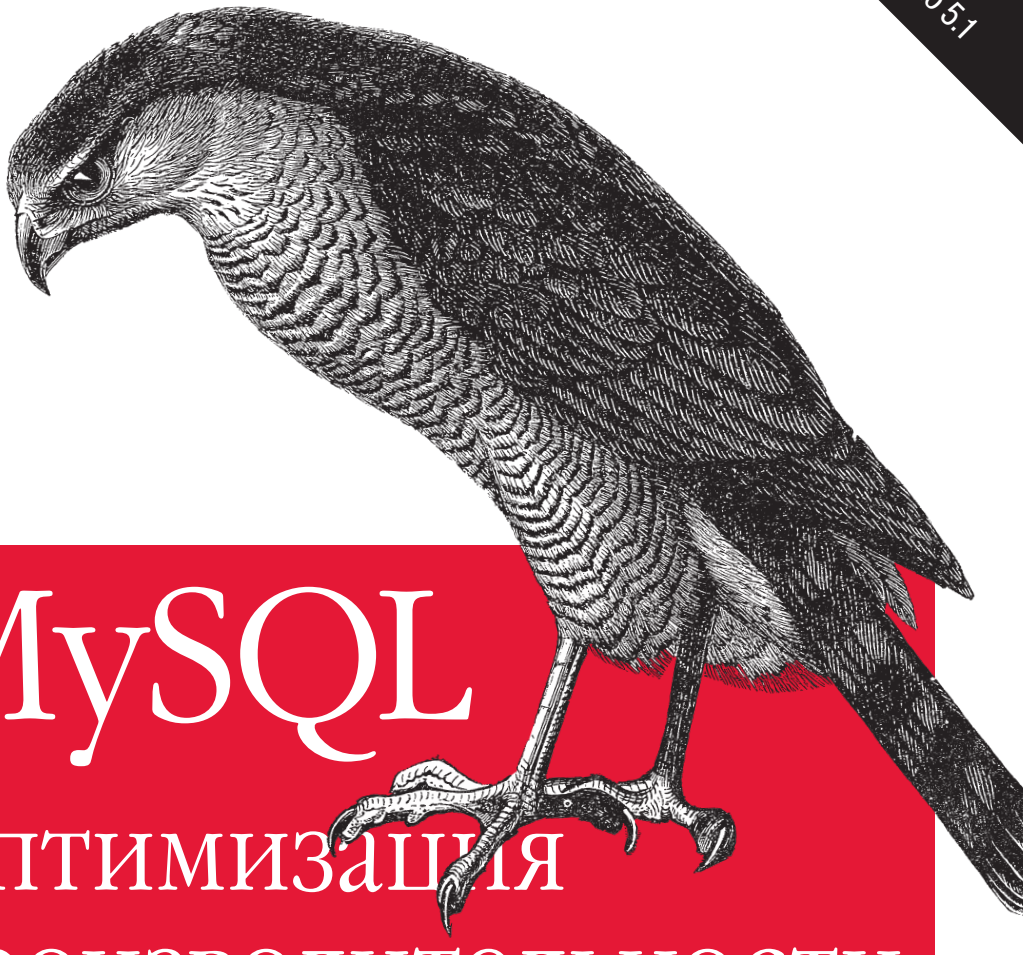
Оптимизация, резервное копирование, репликация и многое другое

2-е издание
Включает версию 5.1



MySQL
Оптимизация производительности

Символ
O'REILLY®

Бэрон Шварц, Петр Зайцев,
Вадим Ткаченко, Джереми Заводны,
Арьен Ленц, Дерек Боллинг

High Performance MySQL

Second Edition

Baron Schwartz, Peter Zaitsev, Vadim Tkachenko,
Jeremy D. Zawodny, Arjen Lentz,
Derek J. Balling

O'REILLY®

MySQL
Оптимизация производительности

Второе издание

Бэрон Шварц, Петр Зайцев, Вадим Ткаченко,
Джереми Д. Заводны, Арьен Ленц,
Дерек Дж. Бэллинг

Санкт-Петербург — Москва
2010

Бэрон Шварц, Петр Зайцев, Вадим Ткаченко,
Джереми Д. Заводны, Арьен Ленц, Дерек Дж. Бэллинг

MySQL. Оптимизация производительности, 2-е издание

Перевод А. Слинкина

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Научный редактор	*А. Рындин*
Редактор	*П. Шалин*
Корректор	*С. Минин*
Верстка	*К. Чубаров*

Шварц Б., Зайцев П., Ткаченко В., Заводны Дж., Ленц А., Бэллинг Д.

MySQL. Оптимизация производительности, 2-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 832 с., ил.

ISBN 978-5-93286-153-0

Авторы этой книги – известные специалисты с многолетней практикой – рассказывают о том, как создавать быстрые и надежные системы на основе MySQL. Ими подробно описываются различные нетривиальные подходы, которые позволят задействовать всю мощь этой СУБД.

Рассматриваются методы проектирования схем, индексов и запросов для достижения максимальной производительности. Предлагаются детальные указания по настройке сервера MySQL, операционной системы и оборудования для полного раскрытия их потенциала. Описаны безопасные способы масштабирования приложений, основанные на репликации и балансировании нагрузки.

Второе издание полностью переработано и существенно дополнено, особое внимание уделено отказоустойчивости, безопасности и обеспечению целостности данных.

Книга рекомендуется как новичкам, так и опытным пользователям, которые хотели бы увеличить производительность своих приложений на базе MySQL.

ISBN 978-5-93286-153-0
ISBN 978-0-596-10171-8 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 15.04.2010. Формат 70×100 $^{1}/_{16}$. Печать офсетная.
Объем 52 печ. л. Тираж 1500 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие

Я давно знаком с Петром, Вадимом и Арьеном. Могу засвидетельствовать, что они используют MySQL как в своих собственных проектах, так и при работе со множеством солидных клиентов. В свою очередь, Бэрон пишет клиентское программное обеспечение, упрощающее использование MySQL.

При подготовке второго издания этой книги был учтен практический опыт авторов по оптимизации, репликации, резервному копированию и другим вопросам. Это не просто книга, которая рассказывает, как оптимизировать работу, чтобы использовать MySQL более эффективно, чем раньше. Помимо всего прочего авторы проделали значительную дополнительную работу, выполнив тесты и опубликовав полученные результаты, подтверждающие их точку зрения. Это позволит читателю, заглянув во внутренние механизмы MySQL, в будущем избежать множества ошибок, приводящих к недостаточной производительности.

Я рекомендую эту книгу как новичкам в MySQL, которые успели немного повозиться с сервером и теперь готовы к написанию своего первого серьезного приложения, так и опытным пользователям, которые уже имеют на своем счету хорошо настроенные приложения на базе MySQL, но хотели бы выжать из них еще капельку производительности.

Майкл Видениус
Март 2008 года

Введение

При написании этой книги мы преследовали несколько целей. Многие из них обязаны нашей давней мечте об «идеальном» пособии по MySQL, которое никто из нас не читал, но которое мы всегда искали на книжных полках. Другие подсказал наш опыт помощи пользователям MySQL.

Мы стремились написать книгу, которая была бы не просто введением в язык SQL. Мы не желали, чтобы в ее названии фигурировал какой-то конкретный интервал времени, например «...за тридцать дней» или «Семь дней для...», и не собирались разговаривать с читателем свысока. Прежде всего, нам хотелось написать книгу, способную повысить квалификацию читателя и помочь ему создавать быстрые, надежные системы на основе MySQL. Такую книгу, которая содержала бы ответы на вопросы из разряда «Как настроить кластер серверов MySQL для обработки миллионов и миллионов запросов и быть уверенным, что он продолжит работать даже при выходе из строя пары серверов?».

Мы решили написать книгу, ориентированную не только на потребности создателей приложений MySQL, но и на жесткие требования администраторов баз данных, которым нужно обеспечить бесперебойную работу системы вне зависимости от того, что разработчики или пользователи запускают на сервере. Мы рассчитываем, что у вас уже есть некоторый опыт работы с MySQL, а в идеале, что вы прочли какое-нибудь введение в MySQL. Мы также предполагаем, что у вас есть небольшой опыт системного администрирования, работы с сетями и операционными системами семейства UNIX.

Переработанное и расширенное второе издание включает в себя более глубокое изложение всех тем, присутствовавших в первом издании, а также множество новых разделов. Частично это является ответом на изменения, произошедшие со времени публикации первого издания: СУБД MySQL стала значительно объемнее, сложнее и, что не менее важно, ее популярность существенно возросла. Сообщество MySQL теперь намного обширнее, а крупные корпорации используют MySQL для своих жизненно важных приложений. Со времени первого издания СУБД MySQL стала рассматриваться как ПО масштаба предприятия[1]. Кроме

[1] Мы рассматриваем эту фразу скорее как маркетинговую уловку, но, похоже, многие люди воспринимают ее серьезно.

того, она все чаще используется в приложениях для Интернета, где простои и другие проблемы нельзя ни допустить, ни скрыть.

В результате второе издание построено несколько иначе, чем первое. Мы придаем надежности и корректности работы такое же значение, как и производительности, отчасти потому, что сами использовали MySQL для решения задач, где от сервера баз данных зависят большие деньги. У нас также есть обширный опыт работы с веб-приложениями, где СУБД MySQL стала очень популярной. Второе издание предназначено для выросшего сообщества MySQL, которое не было таким во времена публикации первого издания.

Структура книги

В этой книге освещено множество сложных тем. Они упорядочены таким образом, чтобы упростить их изучение.

Общий обзор

Глава 1 «Архитектура MySQL» посвящена основам, которые необходимо знать, прежде чем приступать к более сложным темам. Для того чтобы эффективно использовать СУБД MySQL, вы должны понимать, как она устроена. В этой главе рассматривается архитектура MySQL и ключевые особенности ее подсистем хранения. Приводятся сведения об основах реляционных баз данных, включая транзакции. Эта глава также может выступать в роли введения в MySQL, если вы уже знакомы с какой-нибудь другой СУБД, например с Oracle.

Закладка фундамента

В следующих четырех главах приведен материал, к которому вы будете обращаться снова и снова в процессе использования MySQL.

В главе 2 «Поиск узких мест: эталонное тестирование и профилирование» рассказывается об основах эталонного тестирования производительности и профилирования. Здесь приводится методика определения того, какого рода нагрузки способен выдерживать сервер, насколько быстро он может выполнять конкретные задачи и т. п. Тестирование приложения следует выполнять до и после серьезных изменений, чтобы понять, насколько они оказались эффективными. Изменения, кажущиеся полезными, при больших нагрузках могут оказать противоположный эффект, и вы никогда не узнаете причину падения производительности, пока не измерите ее точно.

В главе 3 «Оптимизация схемы и индексирование» мы описываем различные нюансы типов данных, проектирования таблиц и индексов. Правильно спроектированная схема помогает MySQL работать значительно быстрее, а многие вещи, которые мы будем обсуждать в последующих главах, зависят от того, насколько хорошо ваше приложение

использует индексы. Четкое понимание индексов и принципов их использования очень важно для эффективного использования MySQL, поэтому вы, скорее всего, будете неоднократно возвращаться к этой главе.

В главе 4 «Оптимизация производительности запросов» речь пойдет о том, как MySQL выполняет запросы и как можно воспользоваться сильными сторонами оптимизатора запросов. Четкое понимание того, как работает оптимизатор, поможет творить с запросами чудеса и лучше разобраться с индексами. (Индексирование и оптимизация запросов – это что-то вроде проблемы яйца и курицы; возможно, для вас будет полезным перечитать заново третью главу после прочтения четвертой.) В этой главе также приведено много конкретных примеров почти всех типичных запросов, иллюстрирующих оптимальную работу MySQL и показывающих, как преобразовать запросы в такую форму, чтобы получить от СУБД максимум возможностей.

Все упомянутые нами до этого момента темы – таблицы, индексы, данные и запросы – касались любых систем управления базами данных. В главе 5 «Расширенные возможности MySQL» мы выйдем за пределы этих основ и покажем, как работают дополнительные расширенные возможности MySQL. Мы рассмотрим кэш запросов, хранимые процедуры, триггеры, кодировки и прочее. Эти средства реализованы в MySQL иначе, чем в других базах данных, и хорошее их понимание откроет перед вами новые возможности для повышения производительности, о которых вы, быть может, даже не задумывались.

Настройка приложения

В следующих двух главах обсуждается, как вносить изменения, повышающие производительность приложений на основе MySQL.

В главе 6 «Оптимизация параметров сервера» мы обсудим, как настроить MySQL, чтобы извлечь максимум возможного из имеющейся аппаратной конфигурации сервера в применении к конкретному приложению. В главе 7 «Оптимизация операционной системы и оборудования» объясняется, как выжать все, что только можно, из операционной системы и используемого вами оборудования. Мы также предложим аппаратные конфигурации, которые могут обеспечить наилучшую производительность для крупномасштабных приложений.

Вертикальное масштабирование после внесения изменений

Одного сервера бывает достаточно далеко не всегда. В главе 8 «Репликация» мы обсудим репликацию, то есть автоматическое копирование данных на несколько серверов. В сочетании с уроками, посвященными масштабированию, распределению нагрузки и обеспечению высокой доступности в главе 9, озаглавленной «Масштабирование и высокая до-

ступность», вы получите базовые знания для масштабирования приложений до необходимого уровня.

Оптимизация зачастую возможна и на уровне самих приложений, работающих на крупномасштабном сервере MySQL. Можно спроектировать крупное приложение хорошо или плохо. Хотя проектирование и не является основной темой этой книги, мы не хотим, чтобы вы тратили все свое время только на MySQL. Глава 10 «Оптимизация на уровне приложения» поможет выявить наиболее очевидные проблемы общей архитектуры, особенно если это касается веб-приложения.

Обеспечение надежности приложения

Хорошо спроектированная, масштабируемая база данных также должна быть защищена от сбоев электроснабжения, атак злоумышленников, ошибок в приложениях и прочих напастей.

В главе 11 «Резервное копирование и восстановление» мы обсудим различные стратегии резервного копирования и восстановления баз данных MySQL. Эти стратегии помогут минимизировать время простоя в случае выхода из строя оборудования и гарантировать, что данные переживут такую «катастрофу».

Глава 12 «Безопасность» дает ясное представление о некоторых вопросах безопасности сервера MySQL. Но гораздо важнее то, что мы предлагаем целый ряд рекомендаций, позволяющих предотвратить внешние вторжения на сервер. Мы расскажем о некоторых редко освещаемых аспектах безопасности баз данных и покажем, как разные решения влияют на их производительность. Обычно с точки зрения производительности имеет смысл использовать как можно более простые политики безопасности.

Различные полезные темы

В последних нескольких главах и приложениях мы углубимся в вопросы, которые либо не вписываются ни в одну из предыдущих глав, либо так часто упоминаются в других главах, что заслуживают отдельного рассмотрения.

В главе 13 «Состояние сервера MySQL» показано, как исследовать текущий режим работы сервера MySQL. Очень важно знать, как получить информацию о состоянии сервера. Но еще важнее понимать, что эта информация означает. Мы подробно рассмотрим команду `SHOW INNODB STATUS`, поскольку она позволяет детально разобраться в операциях, осуществляемых транзакционной подсистемой хранения InnoDB.

В главе 14 «Инструменты для оптимизации производительности» описаны инструменты, которые можно использовать для более эффективного управления MySQL. В их число входят инструменты мониторинга и анализа, инструменты, помогающие писать запросы, и т. д. В этой

главе описывается созданный Бэроном комплект инструментов Maatkit, который расширяет функциональность MySQL и упрощает жизнь администратору базы данных. В ней также рассказано о написанной Бэроном программе *innotop*, которая обладает удобным графическим интерфейсом и позволяет узнавать о том, что делает сервер MySQL. Она работает подобно команде UNIX *top* и может оказать бесценную помощь на всех этапах процесса настройки, позволяя выяснить, что происходит внутри самого сервера MySQL и подсистем хранения.

В приложении A «Передача больших файлов» говорится о том, как эффективно копировать очень большие файлы, что критически важно при работе со значительными объемами данных. В приложении B «Команда EXPLAIN» показано, как на практике использовать очень полезную команду EXPLAIN. Приложение C «Использование Sphinx совместно с MySQL» представляет собой введение в высокопроизводительную систему полнотекстового поиска Sphinx, которая дополняет собственные возможности СУБД MySQL. И наконец, приложение D «Отладка блокировок» поможет вам выяснить, что происходит, когда запросы вызывают конфликтующие друг с другом блокировки.

Версии программного обеспечения и их доступность

MySQL постоянно меняется. С тех пор как Джереми написал план первого издания этой книги, появилось множество версий MySQL. Когда первое издание готовилось к печати, MySQL 4.1 и 5.0 существовали только в виде альфа-версий. С того момента прошло несколько лет, и они стали основой крупных веб-приложений, эксплуатируемых в промышленном масштабе. На момент окончания подготовки второго издания последними версиями являются MySQL 5.1 и 6.0 (MySQL 5.1 – релиз-кандидат, а 6.0 в стадии альфа-версии).

В этой книге мы не ограничиваемся какой-то конкретной версией, а опираемся на свой обширный опыт работы с MySQL в реальных приложениях. В основном речь идет о версии MySQL 5.0, поскольку именно ее мы считаем «текущей». В большинстве примеров предполагается, что вы используете какую-то относительно зрелую версию MySQL 5.0, например MySQL 5.0.40 или более новую. Мы старались отмечать возможности, которые отсутствуют в более старых версиях или появятся только в следующем семействе 5.1. Однако авторитетным источником информации о возможностях каждой конкретной версии является сама документация по MySQL. Мы надеемся, что в процессе чтения этой книги вы будете время от времени посещать сайт разработчиков СУБД, содержащий всю необходимую информацию (*http://dev.mysql.com/doc/*).

Другим замечательным свойством MySQL является то, что она работает практически на всех современных платформах: Mac OS X, Windows,

GNU/Linux, Solaris, FreeBSD и других! Однако мы предпочитаем GNU/Linux[1] и иные UNIX-подобные операционные системы. Пользователи Windows, вероятно, найдут некоторые различия. Например, пути к файлам записываются совершенно иначе. Мы также ссылаемся на стандартные утилиты командной строки UNIX и предполагаем, что вы знаете соответствующие команды в Windows[2].

Еще одна трудность при работе с MySQL на платформе Windows – отсутствие языка Perl в стандартной поставке операционной системы. В состав дистрибутива MySQL входят несколько полезных утилит, написанных на Perl, а в некоторых главах этой книги представлены примеры Perl-сценариев, которые служат основой для более сложных инструментов, создаваемых уже вами. Комплект Maatkit также написан на Perl. Чтобы использовать эти сценарии, вам потребуется загрузить версию Perl для Windows с сайта компании ActiveState и установить дополнительные модули (`DBI` и `DBD::mysql`) для доступа к MySQL.

Типографские соглашения

В книге применяются следующие типографские соглашения:

Курсив

Таким начертанием выделяются новые термины, URL, адреса электронной почты, имена пользователей и хостов, имена и расширения имен файлов, пути к файлам, имена каталогов, а также команд и утилит UNIX.

`Моноширинный шрифт`

Применяется для фрагментов кода, конфигурационных параметров, имен баз данных и таблиц, имен и значений переменных, имен функций, модулей, содержимого файлов и результатов работы команд.

`Моноширинный полужирный шрифт`

Команды или другой текст, который пользователь должен вводить буквально. Также используется для выделения в результатах работы команды.

`Моноширинный курсив`

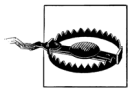
Текст, вместо которого надо подставить значения, вводимые пользователем.

---

[1] Во избежание путаницы мы ссылаемся на Linux, когда пишем о ядре, и на GNU/Linux, когда пишем обо всей инфраструктуре операционной системы, которая поддерживает приложения.

[2] Вы можете найти версии UNIX-утилит для Windows на сайтах *http://unxutils.sourceforge.net* или *http://gnuwin32.sourceforge.net*.

Таким способом выделяются советы, предложения и примечания общего характера.



Таким способом выделяются предупреждения и предостережения.

О примерах кода

Эта книга призвана помочь вам в работе. Поэтому вы можете использовать приведенный в ней код в собственных программах или в создаваемой вами документации. Спрашивать у нас разрешения необязательно, если только вы не собираетесь воспроизводить значительную часть кода. Например, не требуется разрешение, чтобы включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров на компакт-диске нужно получить разрешение. Можно без ограничений цитировать книгу и примеры в ответах на вопросы. Но чтобы включить значительные объемы кода в документацию по собственному продукту, нужно получить разрешение.

Примеры можно найти на сайте *http://www.highperfmysql.com*, где они периодически обновляются. Однако мы не в состоянии обновлять и тестировать код для каждой версии MySQL.

Мы высоко ценим, хотя и не требуем указывать, ссылки на наши издания. В ссылке обычно приводятся название книги, имя автора, издательство и ISBN, например: *«High Performance MySQL: Optimization, Backups, Replication, and More, Second Edition, by Baron Schwartz et al.»* Copyright 2008 O'Reilly Media, Inc., 9780596101718.

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу *permissions@oreilly.com*.

Доступность на Safari

Если на обложке вашей любимой книги присутствует значок Safari® Enabled, это означает, что книга доступна в сетевой библиотеке Safari издательства O'Reilly.

У Safari есть преимущество перед обычными электронными книгами. Это виртуальная библиотека, которая позволяет легко находить тысячи технических изданий, копировать примеры программ, загружать отдельные главы и быстро получать точную и актуальную информацию. Библиотека бесплатна и расположена по адресу *http://safari.oreilly.com*.

Как с нами связаться

Вопросы и замечания по поводу этой книги отправляйте в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США или Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Для этой книги есть веб-страница, на которой выкладываются списки замеченных ошибок, примеры и разного рода дополнительная информация. Адрес страницы:

http://www.oreilly.com/catalog/9780596101718/

Замечания и вопросы технического характера следует отправлять по адресу:

bookquestions@oreilly.com

Дополнительную информацию о наших книгах, конференциях, ресурсных центрах и о сети O'Reilly Network можно найти на сайте:

http://www.oreilly.com

Вы также можете связаться с авторами напрямую. Блог Бэрона находится по адресу *http://www.xaprb.com*.

Петр и Вадим ведут два блога: давно известный и популярный *http://www.mysqlperformanceblog.com* и более новый *http://www.webscalingblog.com*. Сайт их компании, Percona, находится по адресу *http://www.percona.com*.

Сайт компании Арьена, OpenQuery, находится по адресу *http://openquery.com.au*. У Арьена также есть блог *http://arjen-lentz.livejournal.com* и личный сайт *http://lentz.com.au*.

Благодарности ко второму изданию

Разработчик системы Sphinx Андрей Аксенов (Andrew Aksyonov) написал приложение C «Использование Sphinx совместно с MySQL». Мы ставим его на первое место в перечне благодарностей за активное участие в обсуждениях.

В процессе написания этой книги мы получили бесценную помощь от многих людей. Невозможно перечислить всех, кто нам помогал, – мы должны поблагодарить все сообщество MySQL и каждого сотрудника компании MySQL AB. Однако вот список людей, внесших непосредственный вклад в создание настоящего издания (просим извинить, если мы кого-то пропустили): Тобиас Асплунд (Tobias Asplund), Игорь Ба-

баев (Igor Babaev), Паскаль Боргино (Pascal Borghino), Роланд Боуман (Roland Bouman), Рональд Брэдфорд (Ronald Bradford), Марк Каллаган (Mark Callaghan), Джереми Коул (Jeremy Cole), Бритт Кроуфорд (Britt Crawford) и проект HiveDB Project, Василь Димов (Vasil Dimov), Харрисон Фиск (Harrison Fisk), Флориан Хаас (Florian Haas), Дмитрий Жуковский (Dmitri Joukovski) и Zmanda (благодарим за диаграмму, поясняющую мгновенные снимки LVM), Алан Казиндорф (Alan Kasindorf), Шеери Критцер Кабрал (Sheeri Kritzer Cabral), Марко Макела (Marko Makela), Джузеппе Максиа (Giuseppe Maxia), Пол Маккалоги (Paul McCullagh), Б. Кит Мэрфи (B. Keith Murphy), Дирен Пэйтел (Dhiren Patel), Сергей Петруня (Sergey Petrunia), Александр Рубин (Alexander Rubin), Пол Такфилд (Paul Tuckfield), Хейкки Туури (Heikki Tuuri) и Майкл «Монти» Видениус (Michael Widenius).

Отдельная благодарность Энди Ораму (Andy Oram) и Изабель Канкл (Isabel Kunkle), редактору и помощнику редактора нашей книги из издательства O'Reilly, и Рейчел Вилер (Rachel Wheeler), литературному редактору. Благодарим также и остальных сотрудников издательства O'Reilly.

От Бэрона

Я хочу поблагодарить свою жену Линн Рейнвилл и нашего пса по кличке Карбон. Если вам доводилось писать книгу, то уверен, вы знаете, как бесконечно я им благодарен. Я также благодарю Алана Римм-Кауфмана (Alan Rimm-Kaufman) и своих коллег из компании Rimm-Kaufman Group за их поддержку и ободрение в ходе этого проекта. Спасибо Петру, Вадиму и Арьену за то, что они дали мне возможность реализовать свою мечту. И спасибо Джереми и Дереку, которые проложили нам путь.

От Петра

Я годами занимался созданием презентаций, обучением и консультированием по вопросам производительности и масштабирования MySQL и всегда хотел иметь более широкую аудиторию, поэтому был очень взволнован, когда Энди Орам предложил мне поработать над этой книгой. До этого мне не приходилось писать книг, поэтому я не был готов к тому, сколько времени и усилий на это потребуется. Сначала мы собирались лишь обновить первое издание с учетом последних версий MySQL, но оказалось, что надо добавить так много материала, что было решено переписать почти всю книгу.

Это издание является результатом усилий целой команды. Поскольку я был очень загружен раскруткой нашей с Вадимом консультационной компании Percona, а английский для меня не родной язык, то у всех нас были разные роли. Я предложил план и техническое содержание, а затем по мере написания книги занимался пересмотром и расширением материала. Когда к проекту присоединился Арьен (бывший руководитель группы подготовки документации MySQL), мы приступи-

ли к реализации плана. Дело действительно закрутилось, когда к нам присоединился Бэрон, который способен писать высококачественный текст с бешеной скоростью. Вадим оказал огромную помощь в тщательной проверке исходного кода MySQL, а также в подготовке эталонных тестов и проведении других исследований.

По мере работы над книгой мы обнаруживали все больше и больше областей, которые хотели рассмотреть более детально. Многие темы, например репликация, оптимизация запросов, архитектура InnoDB и проектирование, вполне заслуживают отдельных книг, поэтому нам приходилось в какой-то момент останавливаться, оставляя часть материала для возможного следующего издания или для наших блогов, презентаций и статей.

Мы получили неоценимую помощь от наших рецензентов, являющихся одними из ведущих экспертов по MySQL в мире, как в составе, так и вне MySQL AB. В их число входит создатель MySQL Майкл Видениус, автор InnoDB Хейкки Туури, руководитель группы разработчиков оптимизатора MySQL Игорь Бабаев и многие другие.

Я также хочу поблагодарить свою жену Катю Зайцеву и своих детей Ваню и Надежду за то, что они с пониманием отнеслись к необходимости заниматься книгой, из-за чего порой я не мог уделить им достаточно времени. Я также благодарен сотрудникам компании Percona, которые подменяли меня, когда я исчезал для работы над рукописью. Отдельное спасибо хочется сказать Энди Ораму и издательству O'Reilly за то, что издание наконец увидело свет.

От Вадима

Я хочу поблагодарить Петра, вдохновившего меня потрудиться над этой книгой, Бэрона, сыгравшего важную роль в том, что она была написана, и Арьена, работать с которым было очень весело. Спасибо также нашему редактору Энди Ораму, проявившему изрядное терпение по отношению к нам, команде MySQL, создавшей прекрасный продукт, и нашим клиентам, благодаря которым у меня появилась возможность хорошо разобраться в MySQL. И наконец, особая благодарность моей жене Валерии и нашим сыновьям Мирославу и Тимуру, которые всегда поддерживали меня и помогали двигаться вперед.

От Арьена

Я хочу поблагодарить Энди за его мудрость, руководство и терпение. Спасибо Бэрону за то, что он запрыгнул в поезд второго издания, когда тот уже набирал ход, Петру и Вадиму за базовую информацию и тестирование. Спасибо также Джереми и Дереку за написание первого издания.

Хочу поблагодарить всех моих бывших коллег (и нынешних друзей) из MySQL AB, где я получил большую часть знаний на эту тему; в данном

контексте особо хочу упомянуть Монти[1], которого продолжаю считать отцом MySQL даже несмотря на то, что его компания теперь является частью Sun Microsystems. Также я хочу выразить благодарность всем остальным участникам глобального сообщества MySQL.

И напоследок, благодарю свою дочь Фебу, которая на этом этапе своей юной жизни еще не интересуется вещью под названием MySQL. Для некоторых людей неведение является настоящим блаженством, и они показывают нам, что является действительно важным в нашем мире. Для остальных же эта книга может оказаться полезным дополнением на книжной полке. И не забывайте про свою жизнь.

Благодарности к первому изданию

Появление книг, подобных этой, не обходится без участия десятков людей. Без их помощи издание, которое вы держите в руках, было бы, скорее всего, кучей записок, прилепленных к нашим мониторам. В настоящем разделе мы хотим сказать теплые слова о тех людях, которые нам помогали.

Мы не могли бы закончить этот проект без постоянных просьб, подталкивания и поддержки со стороны нашего редактора Энди Орама. Если нужно назвать единственного человека, в наибольшей степени ответственного за то, что вы держите эту книгу в руках, то это Энди. Мы действительно ценим еженедельные придирчивые совещания с ним.

Однако Энди не одинок. В издательстве O'Reilly есть еще много людей, которые принимали участие в превращении наших заметок в книгу, которую вам так хочется прочитать. Мы также хотим поблагодарить людей, занимавшихся производством, иллюстрированием и маркетингом. И, конечно, спасибо Тиму О'Рейли за его постоянную заботу о создании прекрасной документации для популярных программных продуктов с открытым кодом.

Наконец, мы оба хотим сказать большое спасибо нашим рецензентам, которые согласились просматривать черновики этой книги и рассказывали нам о том, что мы делали неправильно. Они потратили часть своего отпуска в 2003 году на просмотр начерно отформатированных версий этого текста, полного опечаток, вводящих в заблуждение выражений и откровенных математических ошибок. Не выделяя никого в отдельности, мы выражаем благодарность Брайану «Krow» Алкеру (Brian «Krow» Aker), Марку «JDBC» Мэтьюсу (Mark «JDBC» Matthews), Джереми «the other Jeremy» Коулу (Jeremy «the other Jeremy» Cole), Майку «VBMySQL.com» Хилльеру (Mike «VBMySQL.com» Hillyer), Реймонду «Rainman» Де Ро (Raymond «Rainman» De Roo), Джеффри «Re-

[1] Имеется в виду Ульф Майкл Видениус (Ulf Michael Widenius) – основной автор оригинальной версии СУБД MySQL и основатель компании MySQL AB. – *Прим. ред.*

gex Master» Фридлу (Jeffrey «Regex Master» Friedl), Джейсону Дехаану (Jason DeHaan), Дену Нелсону (Dan Nelson), Стиву «UNIX Wiz» Фридлу (Steve «UNIX Wiz» Friedl) и Касии «UNIX Girl» Трапзо (Kasia «UNIX Girl» Trapszo).

От Джереми

Я снова хочу поблагодарить Энди, согласившегося взяться за этот проект и неизменно поддерживавшего нас. Помощь Дерека была существенной в процессе подготовки последних 20–30% книги, иначе бы мы не уложились в отведенные сроки. Спасибо ему за согласие принять участие в проекте на его поздней стадии, работу над разделом, посвященным поддержке XML, десятой главой, приложением C и другими частями книги.

Я также хочу поблагодарить своих родителей за то, что они много лет назад подарили мне мой первый компьютер Commodore 64. Они не только терпели первые десять лет, превратившихся в пожизненный компьютерный фанатизм, но и быстро стали помощниками в моих непрекращающихся поисках новых знаний.

Кроме того, я хочу поблагодарить группу людей, от работы с которыми по вербовке приверженцев MySQL на Yahoo! я получал удовольствие последние несколько лет. Джеффри Фридл и Рей Голдбергер вдохновляли и помогали мне на первых этапах этого предприятия. Стив Моррис, Джеймс Харви и Сергей Колычев участвовали в моих постоянных экспериментах с серверами MySQL на Yahoo! Finance, даже когда это отвлекало их от важной работы. Также благодарю бесчисленное множество других пользователей Yahoo!, помогавших мне находить интересные задачи и решения, относящиеся к MySQL. И, что важнее всего, спасибо им за веру в меня, благодаря которой MySQL стала одной из самых важных и заметных частей бизнеса Yahoo!.

Адам Гудман, издатель и владелец *Linux Magazine*, помог мне начать писать для технической аудитории, опубликовав мои статьи по MySQL в 2001 году. Он даже сам не понимает, сколь многому научил меня в плане редактирования и издательского дела. Именно он вдохновил меня не сворачивать с этого пути и продолжать вести ежемесячную колонку в журнале. Спасибо, Адам.

Спасибо Монти и Дэвиду за распространение MySQL в мире. И раз уж я заговорил о компании MySQL AB, спасибо всем прочим замечательным людям, вдохновлявшим меня на написание этой книги: Керри, Лари, Джо, Мартену, Брайану, Полу, Джереми, Марку, Харрисону, Мэтту и всей остальной команде.

Наконец, спасибо всем читателям моего блога за ежедневное неформальное общение о MySQL и обсуждение других технических вопросов. И спасибо Гун Сквад.

От Дерека

Как и Джереми, я должен поблагодарить свою семью, в основном по тем же самым причинам. Я хочу выразить благодарность своим родителям за их постоянное стимулирование меня к написанию книги. Мои бабушка и дедушка снабдили меня деньгами на покупку моего первого компьютера Commodore VIC-20.

Я не могу полностью выразить свою благодарность Джереми за то, что он пригласил меня для совместной работы над этой книгой. Это прекрасный опыт, и я был бы рад снова поработать с ним.

Особое спасибо Реймонду Де Ро (Raymond De Roo), Брайану Вольгемуту (Brian Wohlgemuth), Дэвиду Калафранческо (David Calafrancesco), Тере Доти (Tera Doty), Джею Рубину (Jay Rubin), Биллу Катлану (Bill Catlan), Энтони Хоуву (Anthony Howe), Марку О'Нилу (Mark O'Neal), Джорджу Монтгомери (George Montgomery), Джорджу Барберу (George Barber) и множеству других людей, которые терпеливо выслушивали меня, старались понять, что я хочу сказать, или просто заставляли меня улыбнуться, когда я больше всего нуждался в этом. Без вас я до сих пор писал бы эту книгу и почти наверняка спятил бы в процессе.

1

Архитектура MySQL

Архитектура MySQL очень сильно отличается от архитектур других серверов баз данных и делает эту СУБД эффективной для широкого спектра задач. MySQL не универсальна, но обладает достаточной гибкостью, чтобы отлично работать в очень требовательных средах, например в веб-приложениях. В то же время MySQL может использоваться во встроенных приложениях, хранилищах данных, программном обеспечении индексирования и доставки содержимого, высоконадежных системах с резервированием, системах оперативной обработки транзакций (OLTP) и других системах.

Чтобы максимально эффективно использовать возможности MySQL, нужно понимать ее устройство – чтобы работать с ней, а не против нее. Гибкость MySQL многогранна. Например, ее можно настраивать для взаимодействия с самым различным оборудованием, она поддерживает различные типы данных. Однако самой необычной и важной особенностью MySQL является архитектура подсистем хранения данных, которая отделяет обработку запросов и другие серверные задачи от хранения и извлечения данных. В MySQL 5.1 вы даже можете загружать подсистемы хранения данных как дополнительные модули во время работы. Такое разделение задач позволяет вам выбирать для каждой таблицы способ хранения данных, а также решать, какую производительность, возможности и прочие характеристики вы хотите получить.

В этой главе представлено высокоуровневое описание архитектуры сервера MySQL, рассмотрены основные различия подсистем хранения и объяснено, почему важны эти различия. Мы пытались объяснить архитектуру MySQL на конкретных примерах, стараясь пока не сильно вдаваться в детали. Этот обзор будет полезен как для новичков в обслуживании серверов баз данных, так и для читателей, имеющих серьезный опыт работы с другими аналогичными системами.

Логическая архитектура MySQL

Для того чтобы понять принципы функционирования сервера, необходимо иметь представление о совместной работе различных компонентов MySQL. На рис. 1.1 показана логическая архитектура MySQL.

На самом верхнем уровне содержатся службы, которые не являются уникальными для MySQL. Эти службы необходимы большинству сетевых клиент-серверных инструментов и серверов: они обеспечивают поддержку соединений, идентификацию, безопасность и т. п.

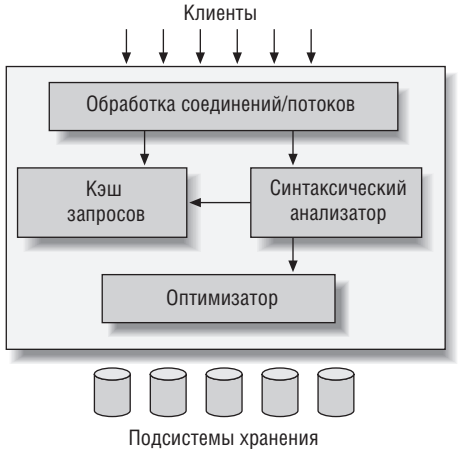


Рис. 1.1. *Логическая архитектура сервера MySQL*

Второй уровень представляет гораздо больший интерес. Здесь сосредоточена значительная часть интеллекта MySQL: синтаксический анализ запросов, оптимизация, кэширование и все встроенные функции (например, функции работы с датами и временем, математические функции, шифрование). На этом уровне реализуется любая независимая от подсистемы хранения данных функциональность, например хранимые процедуры, триггеры и представления.

Третий уровень содержит подсистемы хранения данных. Они отвечают за сохранение и извлечение всех данных, хранимых в MySQL. Подобно различным файловым системам GNU/Linux, каждая подсистема хранения данных имеет свои сильные и слабые стороны. Сервер взаимодействует с ними с помощью *API* (интерфейса прикладного программирования) *подсистемы хранения данных*. Этот интерфейс скрывает различия между подсистемами хранения данных и делает их почти прозрачными на уровне запросов. Кроме того, данный интерфейс содержит пару десятков низкоуровневых функций, выполняющих операции типа «начать транзакцию» или «извлечь строку с таким первичным ключом». Подсистемы хранения не производят синтаксический анализ

кода SQL[1] и не взаимодействуют друг с другом, они просто отвечают на исходящие от сервера запросы.

Управление соединениями и безопасность

Для каждого клиентского соединения выделяется отдельный поток внутри процесса сервера. Запросы по данному соединению исполняются в пределах этого потока, который, в свою очередь, выполняется одним ядром или процессором. Сервер кэширует потоки, так что их не нужно создавать или уничтожать для каждого нового соединения[2].

Когда клиенты (приложения) подключаются к серверу MySQL, сервер должен их идентифицировать. Идентификация основывается на имени пользователя, адресе хоста, с которого происходит соединение, и пароле. Также можно использовать сертификаты X.509 при соединении по протоколу Secure Sockets Layer (SSL). После того как клиент подключился, для каждого запроса сервер проверяет наличие необходимых привилегий (например, разрешено ли клиенту использовать команду `SELECT` применительно к таблице `Country` базы данных `world`). Вопросы, связанные с безопасностью, мы детально обсудим в главе 12.

Оптимизация и выполнение

MySQL осуществляет синтаксический разбор запросов для создания внутренней структуры (дерева разбора), а затем выполняет ряд оптимизаций. В их число входят переписывание запроса, определение порядка чтения таблиц, выбор используемых индексов и т. п. Вы можете повлиять на работу оптимизатора, включив в запрос специальные ключевые слова-подсказки (hints). Также существует возможность попросить сервер объяснить различные аспекты оптимизации. Это позволит вам понять, какие решения принимает сервер, и даст отправную точку для изменения запросов, схем и настроек с целью достижения максимальной эффективности работы. Оптимизатор мы обсудим в главе 4.

Оптимизатор не интересуется тем, в какой подсистеме хранения данных находится каждая таблица, но подсистема хранения данных влияет на то, как сервер оптимизирует запрос. Оптимизатор запрашивает подсистему хранения данных о некоторых ее возможностях и стоимости определенных операций, а также о статистике по содержащимся в таблицах данным. Например, некоторые подсистемы хранения поддерживают типы индексов, которые могут быть полезными для выполнения определенных запросов. Об индексировании и оптимизации схем более подробно написано в третьей главе.

---

[1] Исключением является InnoDB, где происходит синтаксический разбор определений внешних ключей, поскольку в самом сервере MySQL эта функциональность не реализована.

[2] Компания MySQL AB планирует отделить соединения от потоков в будущей версии сервера.

Прежде чем выполнять синтаксический анализ запроса, сервер обращается к кэшу запросов, в котором могут храниться только команды SELECT и соответствующие им результирующие наборы. Если поступает запрос, идентичный уже имеющемуся в кэше, серверу вообще не нужно выполнять анализ, оптимизацию или выполнение запроса – он может просто отправить в ответ на запрос сохраненный результирующий набор! Подробное обсуждение кэша запросов предложено в разделе «Кэш запросов MySQL» главы 5 на стр. 261.

Управление конкурентным доступом

В любой момент может возникнуть ситуация, при которой сразу нескольким запросам требуется одновременно изменить данные, в результате чего возникает задача управления конкурентным доступом. Пока отметим лишь, что MySQL должна решать эту задачу на двух уровнях: сервера и подсистемы хранения данных. Управление конкурентным доступом – обширная тема, которой посвящено много теоретических исследований, но эта книга не о теории и даже не о внутреннем устройстве MySQL. Поэтому мы просто дадим обзор того, что MySQL делает с конкурентными запросами на чтение и запись, чтобы у вас появилось общее представление, которое потребуется вам для понимания оставшейся части этой главы.

Мы будем использовать в качестве примера ящик электронной почты в системе UNIX. Классический формат файла *mbox* очень прост. Сообщения в почтовом ящике *mbox* следуют одно за другим, подряд. В результате читать и анализировать почтовые сообщения очень легко. Это также упрощает доставку почты: достаточно добавить новое сообщение в конец файла.

Но что происходит, если два процесса пытаются одновременно поместить сообщения в один и тот же почтовый ящик? Ясно, что это может привести к повреждению файла, если строки сообщений будут чередоваться. Правильно разработанные почтовые системы используют для предотвращения подобной ситуации блокировку. Если клиент пытается доставить сообщение, когда почтовый ящик заблокирован, ему придется подождать, пока он не сможет сам установить блокировку.

На практике эта схема работает достаточно хорошо, но не поддерживает конкурентный доступ. Поскольку в каждый момент времени только один процесс может изменять содержимое почтового ящика, такой подход вызывает проблемы при работе с почтовыми ящиками большого объема.

Блокировки чтения/записи

Чтение из почтового ящика обычно не вызывает проблем. Ничего страшного, если несколько клиентов считывают информацию из одного и того

же почтового ящика одновременно. Раз они не вносят изменений, ничего плохого не случится. Но что произойдет, если кто-нибудь попытается удалить сообщение номер 25 в тот момент, когда программы осуществляют получение списка электронных писем? Исход подобной ситуации зависит от обстоятельств, но программа чтения может получить поврежденное или неправильное представление о содержимом почтового ящика. Поэтому для безопасности даже считывание из почтового ящика требует специальных предосторожностей.

Если рассматривать почтовый ящик как таблицу базы данных, а каждое почтовое сообщение – как строку, легко увидеть, что в этом контексте сохраняется та же самая проблема. Во многих отношениях почтовый ящик является просто примитивной таблицей базы данных. Модификация строк в такой базе очень похожа на удаление или изменение содержания сообщений в файле почтового ящика.

Решение этой классической проблемы управления совместным доступом довольно простое. В системах, которые имеют дело с совместным доступом на чтение/запись, обычно реализуется набор блокировок, делящихся на два типа. Эти блокировки обычно называют *разделяемыми блокировками* и *монопольными блокировками* или *блокировками чтения* и *блокировками записи*.

Если не вдаваться в подробности, данную концепцию можно описать следующим образом. Блокировки чтения ресурса являются разделяемыми или взаимно не блокирующими: любое количество клиентов может производить считывание из ресурса в одно и то же время, не влияя друг на друга. Блокировки записи, наоборот, являются монопольными. Иными словами, они исключают возможность установки блокировки чтения и других блокировок записи, поскольку единственной безопасной политикой является наличие только одного клиента, осуществляющего запись в данный момент времени, и предотвращение всех операций чтения содержимого ресурса на период выполнения записи.

В мире баз данных блокировки происходят постоянно: MySQL запрещает одному клиенту считывать данные, когда другой клиент их изменяет. Управление блокировками осуществляется с использованием внутренних механизмов СУБД и прозрачно для клиентов.

Детальность блокировок

Одним из способов усовершенствования управления блокировками разделяемого ресурса является увеличение избирательности блокировок. Вместо того чтобы блокировать весь ресурс, можно заблокировать только ту его часть, в которую необходимо внести изменения. Еще лучше заблокировать лишь модифицируемый фрагмент данных. Минимизация объема ресурсов, которые вы блокируете в каждый момент времени, позволяет выполнять одновременные операции с одним и тем же объектом, если эти операции не конфликтуют друг с другом.

Немаловажная проблема заключается в том, что блокировки потребляют ресурсы. Все составляющие их операции – получение блокировки, выяснение, свободна ли блокировка, освобождение блокировки и так далее – вызывают собственные накладные расходы. Если система тратит слишком много времени на управление блокировками вместо того, чтобы расходовать его на сохранение и извлечение данных, то пострадает производительность.

Стратегия блокирования являет собой компромисс между накладными расходами на реализацию блокировок и безопасностью данных, причем этот компромисс оказывает заметное влияние на производительность. Большинство коммерческих серверов баз данных не предоставляет вам большого выбора: вы получаете блокировки таблиц на уровне строки, нередко в сочетании с множеством сложных способов обеспечить хорошую производительность из-за большого количества блокировок.

Со своей стороны СУБД MySQL предлагает такой выбор. Подсистемы хранения данных могут реализовывать собственные политики блокировки и уровни детализации блокировок. Управление блокировками является очень важным решением при проектировании подсистем хранения данных. Фиксация детализации на определенном уровне может дать в ряде случаев лучшую производительность, но сделать эту подсистему менее подходящей для других целей. Поскольку MySQL предлагает несколько подсистем хранения, нет необходимости принимать единственное решение на все случаи жизни. Давайте рассмотрим две самые важные стратегии блокировки.

Табличные блокировки

Основной стратегией блокировки в MySQL, дающей наименьшие накладные расходы, является *табличная блокировка*. Такая блокировка аналогична вышеописанным блокировкам почтового ящика: она блокирует всю таблицу. Когда клиент хочет выполнить запись в таблицу (вставку, удаление, обновление и т. п.), он захватывает блокировку на запись для всей таблицы. Такая блокировка предотвращает все остальные операции чтения и записи. В тот момент, когда никто не производит запись, любой клиент может получить блокировку на чтение и она не будет конфликтовать с другими аналогичными блокировками.

Блокировки таблиц имеют различные модификации для обеспечения высокой производительности в разных ситуациях. Например, блокировки таблицы `READ LOCAL` разрешают некоторые типы одновременных операций записи. Также блокировки записи имеют более высокий приоритет, чем блокировки чтения, поэтому запрос блокировки записи будет помещен в очередь перед уже имеющимися запросами блокировки чтения (блокировки записи могут отодвигать в очереди блокировки чтения, но блокировки чтения не могут отодвигать блокировки записи).

Хотя подсистемы хранения данных могут управлять своими собственными блокировками, сама СУБД MySQL также использует для различных целей разные блокировки, действующие на уровне таблицы. Например, для таких команд, как ALTER TABLE, сервер применяет табличную блокировку вне зависимости от подсистемы хранения данных.

Блокировки строк

Наибольшие возможности совместного доступа (и наибольшие накладные расходы) дают *блокировки строк*. Блокировка на уровне строк доступна, среди прочих, в подсистемах хранения данных InnoDB и Falcon. Блокировки строк реализуются подсистемами хранения данных, а не сервером (взгляните на иллюстрацию логической архитектуры). Сервер ничего не знает о блокировках, реализованных подсистемой хранения данных, и, как вы увидите ниже в этой главе и в остальных разделах книги, все подсистемы хранения данных реализуют блокировки по-своему.

Транзакции

Транзакцией называется *атомарная* группа запросов SQL, т. е. которые рассматриваются как *единое целое*. Если подсистема базы данных может выполнить всю группу запросов, она делает это, но если любой из запросов не может быть выполнен в результате сбоя или по какой-то другой причине, не будет выполнен ни один запрос группы. Все или ничего.

Мало что в этом разделе является специфичным для MySQL. Если вы уже знакомы с транзакциями ACID, можете спокойно перейти к подразделу «Транзакции в MySQL» на стр. 34.

Банковское приложение является классическим примером, показывающим, почему необходимы транзакции. Представьте себе банковскую базу данных с двумя таблицами: checking и savings (текущий и сберегательный счета). Чтобы переместить $200 с текущего счета клиента банка на его сберегательный счет, вам нужно сделать, по меньшей мере, три шага:

1. Убедиться, что остаток на текущем счете больше $200.
2. Вычесть $200 из остатка текущего счета.
3. Добавить $200 к остатку сберегательного счета.

Вся операция должна быть организована как транзакция, чтобы в случае неудачи на любом из этих трех этапов все выполненные ранее шаги были отменены.

Вы начинаете транзакцию командой START TRANSACTION, а затем либо фиксируете изменения командой COMMIT, либо отменяете их командой ROLLBACK. Таким образом, код SQL для нашей транзакции может выглядеть следующим образом:

```
1 START TRANSACTION;
2 SELECT balance FROM checking WHERE customer_id = 10233276;
3 UPDATE checking SET balance = balance - 200.00 WHERE customer_id = 10233276;
4 UPDATE savings SET balance = balance + 200.00 WHERE customer_id = 10233276;
5 COMMIT;
```

Но сами по себе транзакции – это еще не все. Что произойдет в случае сбоя сервера базы данных во время выполнения четвертой строки? Клиент, вероятно, просто потеряет $200. А если другой процесс снимет весь остаток с текущего счета в момент между выполнением строк 3 и 4? Банк предоставит клиенту кредит размером $200, даже не зная об этом.

Транзакций недостаточно, если система не проходит тест ACID. Аббревиатура ACID расшифровывается как Atomicity, Consistency, Isolation и Durability (атомарность, непротиворечивость, изолированность и долговечность). Это тесно связанные критерии, которым должна соответствовать правильно функционирующая система транзакционной обработки:

Атомарность

Транзакция должна функционировать как единая неделимая единица работы таким образом, чтобы вся транзакция была либо выполнена, либо отменена. Когда транзакции являются атомарными, не существует такого понятия, как частично выполненная транзакция: все или ничего.

Непротиворечивость

База данных должна всегда переходить из одного непротиворечивого состояния в последующее. В нашем примере непротиворечивость гарантирует, что сбой между третьей и четвертой строками не приведет к исчезновению $200 с текущего счета. Поскольку транзакция не будет зафиксирована, ни одно из изменений в этой транзакции не будет отражено в базе данных.

Изолированность

Результаты транзакции обычно невидимы другим транзакциям, пока она не закончена. Это гарантирует, что если в нашем примере программа суммирования остатков на банковских счетах будет запущена после третьей строки, но перед четвертой, она по-прежнему увидит $200 на текущем счете. Когда мы будем обсуждать уровни изоляции, вы поймете, почему мы говорим *обычно* невидимы.

Долговечность

Будучи зафиксированы, внесенные в ходе транзакции изменения становятся постоянными. Это означает, что изменения должны быть записаны так, чтобы данные не могли быть потеряны в случае сбоя системы. Долговечность, однако, является несколько расплывчатой концепцией, поскольку на самом деле существует много уровней. Некоторые стратегии обеспечения долговечности предоставляют более сильные гарантии безопасности, чем другие, и ни одна из них

не является надежной на 100%. Мы обсудим, что же на самом деле означает долговечность в MySQL в последующих главах, особенно в разделе «Настройка ввода-вывода InnoDB» главы 6 на стр. 353.

Транзакции ACID гарантируют, что банк не потеряет ваши деньги. Вообще очень сложно или даже невозможно сделать это с помощью логики приложения. Чтобы обеспечить гарантии ACID, ACID-совместимый сервер баз данных должен выполнить множество сложных действий, о которых вы, возможно, даже не догадываетесь.

Как и в случае повышения уровня детализации блокировок, оборотной стороной усиленной безопасности является то, что сервер базы данных должен выполнять больше работы. Сервер базы данных с транзакциями ACID обычно требует большей мощности процессора, объема памяти и дискового пространства, чем без них. Как мы уже упоминали, это тот самый случай, когда архитектура подсистем хранения данных MySQL оказывается благом. Вы можете решить, требует ли ваше приложение использования транзакций. Если они вам на самом деле не нужны, можно добиться большей производительности, выбрав для некоторых типов запросов нетранзакционную подсистему хранения данных. Чтобы установить нужный уровень защиты без использования транзакций, применяется команда `LOCK TABLES`. Все в ваших руках.

Уровни изоляции

Изолированность – более сложное понятие, чем кажется на первый взгляд. Стандарт SQL определяет четыре уровня изоляции с конкретными правилами, устанавливающими, какие изменения видны внутри и вне транзакции, а какие нет. Более низкие уровни изоляции обычно допускают большую степень совместного доступа и вызывают меньше накладных расходов.

Каждая подсистема хранения данных реализует уровни изоляции несколько по-разному, и они не обязательно соответствуют тому, что вы ожидаете, если привыкли к другой СУБД (мы не будем вдаваться сейчас в излишние подробности). Вам следует ознакомиться с руководствами по тем подсистемам хранения данных, которые вы решите использовать.

Давайте вкратце рассмотрим четыре уровня изоляции:

`READ UNCOMMITTED`

На уровне изоляции `READ UNCOMMITTED` транзакции могут видеть результаты незафиксированных транзакций. На этом уровне вы можете столкнуться со множеством проблем, если не знаете абсолютно точно, что делаете. Используйте этот уровень, если у вас есть на то веские причины. На практике `READ UNCOMMITTED` используется редко, поскольку его производительность ненамного выше, чем у других

уровней, имеющих множество преимуществ. Чтение незафиксированных данных еще называют *грязным чтением* (*dirty read*).

READ COMMITTED

Уровнем изоляции по умолчанию для большинства СУБД (но не для MySQL!) является READ COMMITTED. Он удовлетворяет вышеприведенному простому определению изолированности: транзакция увидит только те изменения, которые были уже зафиксированы другими транзакциями к моменту ее начала, а произведенные ею изменения останутся невидимыми для других транзакций, пока текущая транзакция не будет зафиксирована. На этом уровне возможен феномен *невоспроизводимого чтения* (*nonrepeatable read*). Это означает, что вы можете выполнить одну и ту же команду дважды и получить различный результат.

REPEATABLE READ

Уровень изоляции REPEATABLE READ решает проблемы, которые возникают на уровне READ UNCOMMITTED. Он гарантирует, что любые строки, которые считываются в контексте транзакции, будут «выглядеть такими же» при последовательных операциях чтения в пределах одной и той же транзакции, однако теоретически на этом уровне возможен феномен *фантомного чтения* (*phantom reads*). Попросту говоря, фантомное чтение может происходить в случае, если вы выбираете некоторый диапазон строк, затем другая транзакция вставляет новую строку в этот диапазон, после чего вы выбираете тот же диапазон снова. В результате вы увидите новую «фантомную» строку. В InnoDB и Falcon проблема фантомного чтения решается с помощью MVCC (multiversion concurrency control), о чем будет рассказано ниже в этой главе.

REPEATABLE READ является в MySQL уровнем изоляции транзакций по умолчанию. Подсистемы хранения данных InnoDB и Falcon следуют этому соглашению. О том как его изменить, написано в главе 6. Некоторые другие подсистемы хранения данных поступают так же, но выбор остается за конкретной подсистемой.

SERIALIZABLE

Самый высокий уровень изоляции, SERIALIZABLE, решает проблему фантомного чтения, заставляя транзакции выполняться в таком порядке, чтобы исключить возможность конфликта. В двух словах, уровень SERIALIZABLE блокирует каждую строку, которую транзакция читает. На этом уровне может возникать множество задержек и конфликтов при блокировках. На практике данный уровень изоляции применяется достаточно редко, но потребности вашего приложения могут заставить вас использовать его, согласившись с меньшей степенью совместного доступа в пользу стабильности данных.

В табл. 1.1 приведена сводка различных уровней изоляции и недостатки, присущие каждому из них.

Таблица 1.1. Уровни изоляции ANSI SQL

Уровень изоляции	Возможность чернового чтения	Возможность невоспроизводимого чтения	Возможность фантомного чтения	Блокировка чтения
READ UNCOMMITTED	Да	Да	Да	Нет
READ COMMITTED	Нет	Да	Да	Нет
REPEATABLE READ	Нет	Нет	Да	Нет
SERIALIZABLE	Нет	Нет	Нет	Да

Взаимоблокировки

Взаимоблокировка происходит, когда две или более транзакции запрашивают блокировку одних и тех же ресурсов, в результате чего образуется циклическая зависимость. Они также возникают в случае, если транзакции пытаются заблокировать ресурсы в разном порядке. Взаимоблокировки могут происходить, когда несколько транзакций блокируют одни и те же ресурсы. Рассмотрим в качестве примера следующие две транзакции, обращающиеся к таблице `StockPrice`:

Транзакция #1

```
START TRANSACTION;
UPDATE StockPrice SET close = 45.50 WHERE stock_id = 4 and date = '2002-05-01';
UPDATE StockPrice SET close = 19.80 WHERE stock_id = 3 and date = '2002-05-02';
COMMIT;
```

Транзакция #2

```
START TRANSACTION;
UPDATE StockPrice SET high = 20.12 WHERE stock_id = 3 and date = '2002-05-02';
UPDATE StockPrice SET high = 47.20 WHERE stock_id = 4 and date = '2002-05-01';
COMMIT;
```

Если вам не повезет, то каждая транзакция выполнит первый запрос и обновит строку данных, заблокировав ее в процессе обновления. Затем обе транзакции попытаются обновить вторую строку, но обнаружат, что она уже заблокирована. В результате одна транзакция будет до бесконечности ожидать окончания другой, и конфликт не разрешится до тех пор, пока не произойдет какое-то событие, которое снимет взаимную блокировку.

Для разрешения этой проблемы в системах баз данных реализованы различные формы обнаружения взаимоблокировок и тайм-аутов. Такие развитые подсистемы хранения данных, как InnoDB, обнаруживают циклические зависимости и немедленно возвращают ошибку. На самом деле это очень хорошо, иначе взаимоблокировки проявлялись бы как очень медленные запросы. Другие системы в подобных ситуациях откатывают транзакцию по истечении тайм-аута, что не очень хорошо. Способом, которым InnoDB обрабатывает взаимоблокировки, является

откат той транзакции, которая захватила меньше всего монопольных блокировок строк (приблизительный показатель легкости отката).

Поведение и порядок блокировок зависят от конкретной подсистемы хранения данных, так что в некоторых подсистемах при определенной последовательности команд могут происходить взаимоблокировки, хотя в других подсистемах при таких же условиях они не возникают. Взаимоблокировки имеют двойственную природу: некоторые действительно неизбежны из-за характера обрабатываемых данных, другие же вызваны тем, как работает конкретная подсистема хранения.

Взаимоблокировку нельзя разрешить без отката одной из транзакций, частичного либо полного. Существование взаимоблокировок в транзакционных системах – непреложный факт, с учетом которого ваше приложение и нужно проектировать. При возникновении такой ситуации многие приложения могут просто попытаться выполнить транзакцию с самого начала.

Ведение журнала транзакций

Ведение журнала помогает сделать транзакции более эффективными. Вместо обновления таблиц на диске каждый раз, когда происходит какое-либо изменение, подсистема хранения данных может изменить находящуюся в памяти копию данных. Это происходит очень быстро. Затем подсистема хранения запишет сведения об изменениях в журнал транзакции, который хранится на диске и потому долговечен (энергонезависим). Это также относительно быстрая операция, поскольку добавление событий в журнал сводится к операции последовательного ввода-вывода в пределах ограниченной области диска вместо операций случайного ввода-вывода в разных местах. Впоследствии в какой-то более поздний момент времени процесс обновит таблицу на диске. Таким образом, большинство подсистем хранения данных, использующих этот прием (*упреждающую запись в журнал*), сохраняют изменения на диске дважды[1].

Если происходит сбой после того, как внесена соответствующая запись в журнал транзакции, но до того, как обновлены сами данные, подсистема хранения может восстановить изменения после перезапуска сервера. Конкретный метод восстановления различается для каждой подсистемы хранения данных.

Транзакции в MySQL

Компания MySQL AB предоставляет пользователям три транзакционных подсистемы хранения данных: InnoDB, NDB Cluster и Falcon. Су-

[1] Подсистема хранения данных **PBXT за счет хитрых уловок иногда обходится** без упреждающей записи в журнал.

ществует также несколько подсистем от сторонних разработчиков. Наиболее известными сейчас являются solidDB и PBXT. В следующем разделе мы обсудим некоторые свойства каждой из упомянутых выше подсистем.

Режим AUTOCOMMIT

MySQL по умолчанию работает в режиме AUTOCOMMIT. Это означает, что если вы не начали транзакцию явным образом, каждый запрос автоматически выполняется в отдельной транзакции. Вы можете включить или отключить режим AUTOCOMMIT для текущего соединения, установив следующее значение конфигурационной переменной:

```
mysql> SHOW VARIABLES LIKE 'AUTOCOMMIT';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    |   ON  |
+---------------+-------+
1 row in set (0.00 sec)
mysql> SET AUTOCOMMIT = 1;
```

Значения 1 и ON эквивалентны, так же как и 0 и OFF. После отправки запроса в режиме AUTOCOMMIT=0 вы оказываетесь в транзакции, пока не выполните команду COMMIT или ROLLBACK. После этого MySQL немедленно начинает новую транзакцию. Изменение значения переменной AUTOCOMMIT не оказывает влияния на нетранзакционные таблицы, например на таблицы типа MyISAM или Memory, которые по своей природе всегда работают в режиме AUTOCOMMIT.

Определенные команды, будучи выполнены в контексте начатой транзакции, заставляют MySQL зафиксировать ее. Обычно это команды языка определения данных (Data Definition Language – DDL), которые вносят значительные изменения в структуру таблиц, например ALTER TABLE, но LOCK TABLES и некоторые другие директивы также обладают этим свойством. Полный список команд, автоматически фиксирующих транзакцию, вы можете найти в документации к вашей версии MySQL.

MySQL позволяет устанавливать уровень изоляции с помощью команды SET TRANSACTION ISOLATION LEVEL, которая начинает действовать со следующей транзакции. Вы можете настроить уровень изоляции для всего сервера в конфигурационном файле (см. главу 6) или на уровне отдельного сеанса:

```
mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

MySQL распознает четыре стандартных уровня изоляции ANSI, а InnoDB поддерживает их все. Другие подсистемы хранения данных обеспечивают разную степень поддержки уровней изоляции.

Совместное использование различных подсистем хранения данных в транзакциях

MySQL не управляет транзакциями на уровне сервера. Сами транзакции реализуются подсистемами хранения данных. Это означает, что вы не можете надежно сочетать различные подсистемы в одной транзакции. Компания MySQL AB работает над добавлением высокоуровневой службы управления транзакциями на сервере, которая сделает безопасным использование транзакционных таблиц разного типа в одной транзакции. А пока будьте осторожны.

Если вы используете в одной транзакции транзакционные и нетранзакционные таблицы (например, типа InnoDB и MyISAM), то все будет работать хорошо, пока не случится что-то непредвиденное.

Однако если потребуется выполнить откат, то изменения, внесенные в нетранзакционную таблицу, нельзя будет отменить. Это оставляет базу данных в несогласованном состоянии и восстановить ее после такого события будет нелегко, что ставит под сомнение всю идею транзакций в целом. Вот почему так важно выбирать для каждой таблицы подходящую подсистему хранения.

MySQL обычно не выдает ни предупреждений, ни ошибок, если вы выполняете транзакционные операции над нетранзакционной таблицей. Иногда при откате транзакции будет сгенерировано сообщение «Some nontransactional changed tables couldn't be rolled back» (Откат некоторых измененных нетранзакционных таблиц невозможен), но большую часть времени вы не будете знать о том, что работаете с нетранзакционными таблицами.

Явные и неявные блокировки

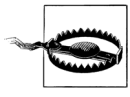
В подсистеме InnoDB используется двухфазный протокол блокировки. Она может устанавливать блокировки в любой момент времени на протяжении всей транзакции, но не снимает их до выполнения команды `COMMIT` или `ROLLBACK`. Все блокировки снимаются одновременно. Описанные ранее механизмы блокировки являются неявными. InnoDB обрабатывает блокировки автоматически в соответствии с уровнем изоляции.

Однако InnoDB поддерживает и явную блокировку, которая в стандарте SQL вообще не упоминается:

- `SELECT ... LOCK IN SHARE MODE`
- `SELECT ... FOR UPDATE`

MySQL также поддерживает команды `LOCK TABLES` и `UNLOCK TABLES`, которые реализуются сервером, а не подсистемой хранения. У них есть свое применение, но они не являются заменой транзакциям. Если вам нужны транзакции, используйте транзакционную подсистему хранения.

Нам часто попадаются приложения, перенесенные из MyISAM в InnoDB, в которых по-прежнему используется команда LOCK TABLES. В транзакционной подсистеме хранения эти команды не нужны благодаря блокировкам на уровне строки и могут вызывать серьезные проблемы с производительностью.



Взаимодействие между командой LOCK TABLES и транзакциями довольно сложное, и в некоторых версиях сервера их поведение непредсказуемо. Поэтому мы рекомендуем использовать команду LOCK TABLES только в рамках транзакции с выключенным режимом AUTOCOMMIT, вне зависимости от того, какой подсистемой хранения вы пользуетесь.

Multiversion Concurrency Control (MVCC)

Большая часть транзакционных подсистем хранения в MySQL, например InnoDB, Falcon и PBXT, используют не просто механизм блокировки строк, а блокировку строк в сочетании с методикой повышения степени конкурентности под названием MVCC (multiversion concurrency control – многоверсионное управление конкурентным доступом). Методика MVCC не является уникальной для MySQL: она используется также в Oracle, PostgreSQL и некоторых других СУБД.

MVCC позволяет во многих случаях вообще отказаться от блокировки и способна значительно снизить накладные расходы. В зависимости от способа реализации она может допускать чтение без блокировок, а блокировать лишь необходимые строки во время операций записи.

Принцип работы MVCC заключается в сохранении мгновенного снимка данных, какими они были в некоторый момент времени. Это означает, что вне зависимости от своей длительности транзакции могут видеть согласованное представление данных. Это также означает, что различные транзакции могут видеть разные данные в одних и тех же таблицах в одно и то же время! Если вы никогда не сталкивались с этим раньше, то наверняка будете удивлены.

Каждая подсистема хранения реализует MVCC по-своему. Некоторые из вариантов включают в себя оптимистическое и пессимистическое управление совместным доступом. Мы проиллюстрируем один из способов работы MVCC, объяснив упрощенную версию поведения InnoDB.

InnoDB реализует MVCC путем сохранения с каждой строкой двух дополнительных скрытых значений, в которых записано, когда строка была создана и когда срок ее хранения истек (или она была удалена). Вместо записи реальных значений момента времени, когда произошли указанные события, строка хранит системный номер версии для этого момента. Данное число увеличивается на единицу в начале каждой

транзакции. Новая транзакция на момент ее начала хранит свою собственную запись текущей версии системы. Любой запрос должен сравнивать номера версий каждой строки с версией транзакции. Давайте посмотрим, как эта методика применяется к конкретным операциям, когда транзакция имеет уровень изоляции REPEATABLE READ:

SELECT

Подсистема InnoDB должна проверить каждую строку, чтобы убедиться, что она отвечает двум критериям:

- InnoDB должна найти версию строки, которая по крайней мере такая же старая, как версия транзакции (то есть ее номер версии должен быть меньше или равен номеру версии транзакции). Это гарантирует, что либо строка существовала до начала транзакции, либо транзакция создала или изменила эту строку.
- Версия удаления строки должна быть не определена или ее значение больше, чем версия транзакции. Это гарантирует, что строка не была удалена до начала транзакции.

Строки, которые проходят обе проверки, могут быть возвращены как результат запроса.

INSERT

InnoDB записывает текущий системный номер версии вместе с новой строкой.

DELETE

InnoDB записывает текущий системный номер версии как идентификатор удаления строки.

UPDATE

InnoDB создает новую копию строки, используя системный номер версии в качестве версии новой строки. Она также записывает системный номер версии как версию удаления старой строки.

Результатом хранения всех этих дополнительных записей является то, что большинство запросов на чтение никогда не ставит блокировки. Они просто считывают данные настолько быстро, насколько можно, обеспечивая выборку только тех строк, которые удовлетворяют заданному критерию. Недостатком подобного подхода является то, что подсистема хранения должна записывать для каждой строки дополнительные данные, выполнять лишнюю работу при проверке строк и производить некоторые дополнительные служебные операции.

Методика MVCC работает только на уровнях изоляции REPEATABLE READ и READ COMMITTED. Уровень READ UNCOMMITTED несовместим с MVCC, поскольку запросы не считывают версию строки, соответствующую их версии транзакции. Они читают самую последнюю версию, несмотря ни на что.

Уровень SERIALIZABLE несовместим с MVCC, поскольку операции чтения блокируют каждую возвращаемую строку.

В табл. 1.2 сведены различные модели блокировки и уровни конкуренции в MySQL.

Таблица 1.2. Модели блокировки и конкуренция в MySQL при использовании уровня изоляции по умолчанию

Стратегия блокировки	Конкуренция	Накладные расходы	Подсистемы хранения
Уровень таблицы	Самая низкая	Самые низкие	MyISAM, Merge, Memory
Уровень строки	Высокая	Высокие	NDB Cluster
Уровень строки с MVCC	Самая высокая	Самые высокие	InnoDB, Falcon, PBXT, solidDB

Подсистемы хранения в MySQL

В этом разделе приведен обзор подсистем хранения MySQL. Мы не станем вдаваться в подробности, поскольку будем обсуждать различные подсистемы хранения и их поведение на протяжении всей книги. Это издание не включает в себя исчерпывающее описание подсистем хранения, так что для принятия решения о том, какую подсистему использовать, вам надо будет почитать документацию по MySQL. Существуют также форумы, посвященные каждой подсистеме хранения MySQL со ссылками на дополнительную информацию и интересные примеры их использования.

Если вы хотите сравнить подсистемы на высоком уровне, то можете сразу обратиться к табл. 1.3.

MySQL хранит каждую базу данных (также именуемую *схемой*), как подкаталог своего каталога данных в файловой системе. Когда вы создаете таблицу, MySQL сохраняет определение таблицы в файле с расширением *.frm* и именем, совпадающим с именем таблицы. Таким образом, определение таблицы с наименованием MyTable сохраняется в файле *MyTable.frm*. Поскольку MySQL использует для хранения имен баз данных и определений таблиц файловую систему, чувствительность к регистру символов зависит от платформы. В Windows имена таблиц и баз данных MySQL не чувствительны к регистру, а в операционных системах семейства UNIX – чувствительны. Каждая подсистема хранения записывает данные таблиц и индексы по-разному, но определение таблицы сервер обрабатывает самостоятельно.

Чтобы определить, какая подсистема хранения используется для конкретной таблицы, используйте команду SHOW TABLE STATUS. Например, чтобы получить информацию о таблице user в базе данных mysql, выполните следующую команду:

```
mysql> SHOW TABLE STATUS LIKE 'user' \G
*************************** 1. row ***************************
           Name: user
         Engine: MyISAM
     Row_format: Dynamic
           Rows: 6
 Avg_row_length: 59
    Data_length: 356
Max_data_length: 4294967295
   Index_length: 2048
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2002-01-24 18:07:17
    Update_time: 2002-01-24 21:56:29
     Check_time: NULL
      Collation: utf8_bin
       Checksum: NULL
 Create_options:
        Comment: Users and global privileges
1 row in set (0.00 sec)
```

Как видно из листинга, это таблица типа MyISAM. Команда выдала еще много дополнительной информации и статистики. Давайте вкратце рассмотрим, что означает каждая строка:

`Name`

Имя таблицы.

`Engine`

Подсистема хранения. В старых версиях MySQL этот столбец назывался `Type`, а не `Engine`.

`Row_format`

Формат строки. Для таблицы MyISAM он может иметь значение `Dynamic`, `Fixed` или `Compressed`. Длина динамических строк варьируется, поскольку они содержат поля переменной длины типа `VARCHAR` или `BLOB`. Строки фиксированного типа имеют один и тот же размер, они состоят из полей с постоянной длиной, например `CHAR` и `INTEGER`. Сжатые строки существуют только в сжатых таблицах, см. подраздел «Сжатые таблицы типа MyISAM» ниже на стр. 44.

`Rows`

Количество строк в таблице. Для нетранзакционных таблиц это число всегда точное. Для транзакционных таблиц оно обычно приблизительное.

`Avg_row_length`

Сколько байтов содержит в среднем каждая строка.

`Data_length`

Объем данных (в байтах), который содержит вся таблица.

`Max_data_length`

Максимальный объем данных, который может хранить эта таблица. Подробности см. в подразделе «Хранение» ниже на стр. 42.

`Index_length`

Какой объем дискового пространства занимают данные индексов.

`Data_free`

Для таблицы типа MyISAM показывает объем выделенного пространства, которое на данный момент не используется. В этом пространстве хранятся ранее удаленные строки. Оно может быть использовано в будущем при выполнении команд `INSERT`.

`Auto_increment`

Следующее значение атрибута `AUTO_INCREMENT`.

`Create_time`

Время создания таблицы.

`Update_time`

Время последнего изменения таблицы.

`Check_time`

Время последней проверки таблицы командой `CHECK TABLE` или утилитой *myisamchk*.

`Collation`

Подразумеваемая по умолчанию кодировка и схема упорядочения для символьных столбцов в этой таблице. Подробное описание см. в разделе «Кодировки и схемы упорядочения» в главе 5.

`Checksum`

Текущая контрольная сумма содержимого всей таблицы, если включен ее подсчет.

`Create_options`

Любые другие параметры, которые были указаны при создании таблицы.

`Comment`

Это поле содержит различную дополнительную информацию. Для таблиц типа MyISAM в нем хранятся комментарии, добавленные при создании таблицы. Если таблица использует подсистему хранения InnoDB, здесь приводится объем свободного места в табличном пространстве InnoDB. Для представлений комментарий содержит текст «VIEW».

Подсистема MyISAM

Будучи подсистемой хранения по умолчанию в MySQL, MyISAM являет собой удачный компромисс между производительностью и функциональностью. Так, она предоставляет полнотекстовое индексирование, сжатие и пространственные функции (для геоинформационных систем – ГИС). MyISAM не поддерживает транзакции и блокировки на уровне строк.

Хранение

MyISAM обычно хранит каждую таблицу в двух файлах: файле данных и индексном файле. Эти файлы имеют расширения соответственно *.MYD* и *.MYI*. Формат MyISAM не зависит от платформы, то есть можно копировать файлы данных и индексов с сервера на базе процессора Intel на сервер PowerPC или Sun SPARC безо всяких проблем.

Таблицы типа MyISAM могут содержать как динамические, так и статические строки (строки фиксированной длины). MySQL самостоятельно решает, какой формат использовать, основываясь на определении таблицы. Количество строк в таблице типа MyISAM ограничено в первую очередь доступным дисковым пространством на сервере базы данных и максимальным размером файла, который позволяет создавать операционная система.

Таблицы MyISAM со строками переменной длины, создаваемые в версии MySQL 5.0, настроены по умолчанию на поддержку 256 Тбайт данных с использованием шестибайтных указателей на записи с данными. В более ранних версиях MySQL указатели по умолчанию были четырехбайтными с максимальным объемом данных 4 Гбайт. Все версии MySQL могут поддерживать размер указателя до восьми байтов. Чтобы изменить размер указателя в таблице MyISAM (уменьшить или увеличить), вы должны задать значения для параметров `MAX_ROWS` и `AVG_ROW_LENGTH`, которые представляют приблизительную оценку необходимого пространства:

```
CREATE TABLE mytable (
a INTEGER NOT NULL PRIMARY KEY,
b CHAR(18) NOT NULL
) MAX_ROWS = 1000000000 AVG_ROW_LENGTH = 32;
```

В этом примере мы сообщаем серверу MySQL, что он должен быть готов хранить в таблице по крайней мере 32 Гбайт данных. Чтобы выяснить, какое решение принял MySQL, просто запросите статус таблицы:

```
mysql> SHOW TABLE STATUS LIKE 'mytable' \G
*************************** 1. row ***************************
           Name: mytable
         Engine: MyISAM
     Row_format: Fixed
           Rows: 0
 Avg_row_length: 0
```

```
    Data_length: 0
Max_data_length: 98784247807
   Index_length: 1024
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2002-02-24 17:36:57
    Update_time: 2002-02-24 17:36:57
     Check_time: NULL
 Create_options: max_rows=1000000000 avg_row_length=32
        Comment:
1 row in set (0.05 sec)
```

Как видите, MySQL запомнил параметры создания точно так, как они были указаны. И показывает возможность хранения 91 Гбайт данных! Вы можете изменить размер указателя позже с помощью команды `ALTER TABLE`, но это приведет к тому, что вся таблица и все ее индексы будут переписаны, а подобная процедура занимает обычно длительное время.

Особенности MyISAM

Как одна из самых старых подсистем хранения, включенных в MySQL, MyISAM обладает многими функциями, которые были разработаны за годы использования СУБД для решения различных задач:

Блокировка и конкуренция

MyISAM блокирует целиком таблицы, но не строки. Запросы на чтение получают разделяемые (на чтение) блокировки всех таблиц, к которым они обращаются. Запросы на запись получают монопольные (на запись) блокировки. Однако вы можете вставлять новые строки в таблицу в момент, когда исполняются запросы на выборку данных из таблицы (конкурентные вставки). Это очень важная и полезная возможность.

Автоматическое исправление

MySQL поддерживает автоматическую проверку и исправление таблиц типа MyISAM. См. раздел «Настройка ввода/вывода в MyISAM» главы 6 на стр. 351.

Ручное исправление

Вы можете использовать команды `CHECK TABLE mytable` и `REPAIR TABLE mytable` для проверки таблицы на предмет наличия ошибок и их устранения. Вы также можете использовать командную утилиту *myisamchk* для проверки и исправления таблиц, когда сервер находится в автономном (offline) режиме.

Особенности индексирования

Вы можете создавать индексы по первым 500 символам столбцов типа `BLOB` и `TEXT` в таблицах MyISAM. MyISAM поддерживает полнотекстовые индексы, которые индексируют отдельные слова для

сложных операций поиска. Дополнительную информацию об индексировании см. в главе 3.

Отложенная запись ключей

Таблицы MyISAM, помеченные при создании параметром `DELAY_KEY_WRITE`, не записывают измененные индексные данные на диск в конце запроса. Вместо этого MyISAM сохраняет изменения в буфере памяти. Сброс индексных блоков на диск происходит при заполнении буфера или закрытии таблицы. Это позволяет увеличить производительность работы с часто изменяемыми таблицами. Однако в случае сбоя сервера или системы индексы наверняка будут повреждены и потребуют восстановления. Это можно сделать с использованием скрипта, запускающего утилиту *myisamchk* перед перезапуском сервера, или при помощи параметров автоматического восстановления (даже если вы не используете параметр `DELAY_KEY_WRITE`, эти меры предосторожности не помешают). Вы можете сконфигурировать отложенную запись ключей как глобально, так и для отдельных таблиц.

Сжатые таблицы MyISAM

Некоторые таблицы, например в приложениях на компакт-дисках или в отдельных встроенных (embedded) средах, после их создания и заполнения данными никогда больше не изменяются. Такие таблицы можно сжать средствами MyISAM.

Для сжатия таблиц существует специальная утилита *myisampack*. Модифицировать сжатые таблицы невозможно (хотя при необходимости их следует распаковать, внести изменения и снова сжать), но при этом они занимают гораздо меньше места на диске. В результате их использования увеличивается производительность, поскольку из-за меньшего размера требуется меньше операций ввода-вывода для поиска на диске необходимых записей. Над сжатыми таблицами MyISAM можно строить индексы, но они также доступны только для чтения.

На современном оборудовании накладные расходы по распаковке данных для чтения в большинстве приложений незначительны. Эти накладные расходы перекрываются значительным выигрышем за счет уменьшения количества операций ввода/вывода. Строки сжимаются по отдельности, так что MySQL не нужно распаковывать всю таблицу (или даже страницу) с целью извлечения одной-единственной строки.

Подсистема MyISAM Merge

Подсистема хранения Merge является вариацией на тему MyISAM. Таблица типа Merge представляет собой объединение нескольких структурно одинаковых таблиц MyISAM в одну виртуальную таблицу. Это особенно полезно, когда вы используете MySQL для протоколирования и организации хранилищ данных. Подробное обсуждение объединен-

ных таблиц см. в разделе «Объединенные таблицы и секционирование» главы 5 на стр. 318.

Подсистема InnoDB

Подсистема хранения InnoDB была разработана для транзакционной обработки, в частности для обработки большого количества кратковременных транзакций, которые значительно чаще благополучно завершаются, чем откатываются. Она остается наиболее популярной транзакционной подсистемой хранения. Высокая производительность и автоматическое восстановление после сбоя делают ее популярной и для нетранзакционных применений.

InnoDB сохраняет данные в наборе из одного или нескольких файлов данных. Этот набор файлов именуется *табличным пространством* (*tablespace*). Табличное пространство в сущности является черным ящиком, которым полностью управляет сама InnoDB. В MySQL 4.1 и более поздних версиях СУБД InnoDB может хранить данные и индексы каждой таблицы в отдельных файлах. Кроме того, данная подсистема может располагать табличные пространства на «сырых» (неформатированных) разделах диска. Подробности см. в разделе «Табличное пространство InnoDB» главы 6 на стр. 363.

Для обеспечения высокой степени конкурентности InnoDB использует MVCC и реализует все четыре стандартных уровня изоляции SQL. Для этой подсистемы уровнем изоляции по умолчанию является REPEATABLE READ, а *стратегия блокировки следующего ключа* позволяет предотвратить фантомные чтения: вместо того чтобы блокировать только строки, затронутые в запросе, InnoDB блокирует также интервалы в индексе, предотвращая вставку фантомных строк.

Таблицы InnoDB строятся на *кластерных индексах*, которые мы подробнее обсудим в главе 3. Структуры индексов в InnoDB очень сильно отличаются от других подсистем хранения. Результатом является более быстрый поиск по первичному ключу. Однако *вторичные индексы* (отличные от индекса по первичному ключу) содержат все столбцы, составляющие первичный ключ, так что если первичный ключ длинный, то все прочие индексы будут большими. Если над таблицей построено много индексов, то первичный ключ нужно делать как можно меньшим. InnoDB не сжимает индексы.

На момент написания этой книги InnoDB не может строить индексы путем сортировки, в отличие от MyISAM. В результате InnoDB загружает данные и создает индексы медленнее, чем MyISAM. Любая операция, которая изменяет структуру таблицы InnoDB, приводит к полной перестройке таблицы, включая все индексы.

Подсистему InnoDB создавали в те времена, когда у большинства серверов были медленные диски, один процессор и ограниченный объем памяти. Сегодня, когда многоядерные серверы с огромным объемом памя-

ти и быстрыми дисками становятся менее дорогими, InnoDB испытывает некоторые проблемы с масштабируемостью.

Разработчики InnoDB занимаются поиском решения данной задачи, но на момент написания этой книги некоторые вопросы все еще остаются открытыми. Информацию о способах достижения высокой конкуренции в InnoDB вы можете найти в разделе «Настройка конкурентного доступа в InnoDB» главы 6 на стр. 370.

Помимо обеспечения высокой конкуренции следующей по популярности особенностью InnoDB являются ограничения целостности типа «внешний ключ», которые в самом сервере MySQL еще не реализованы. InnoDB также обеспечивает очень быстрый поиск по первичному ключу.

В подсистеме InnoDB поддерживаются разнообразные внутренние оптимизации. В их число входят упреждающее интеллектуальное считывание данных с диска, адаптивный хеш-индекс (автоматическое построение хеш-индексов в памяти для обеспечения очень быстрого поиска) и буфер вставок для ускорения операций вставки. Мы будем обсуждать эти вопросы дальше в книге.

Подсистема InnoDB очень сложна, и, если вы используете InnoDB, то мы очень рекомендуем вам прочитать раздел «InnoDB Transaction Model and Locking» (Транзакционная модель и блокировки в InnoDB) в руководстве по MySQL. Существует много сюрпризов и исключений, о которых вы должны знать перед началом создания приложений с использованием InnoDB.

Подсистема Memory

Таблицы типа Memory (раньше называвшиеся таблицами типа HEAP) полезны, когда необходимо осуществить быстрый доступ к данным, которые либо никогда не изменяются, либо нет надобности в их сохранении после перезапуска. Обычно таблицы типа Memory обрабатываются примерно на порядок быстрее, чем таблицы MyISAM. Все их данные хранятся в памяти, поэтому запросам не нужно ждать выполнения операций дискового ввода/вывода. Структура таблицы Memory сохраняется после перезапуска сервера, но данные теряются.

Вот несколько хороших применений для таблиц Memory:

- Для «справочных» таблиц или таблиц «соответствия», например для таблицы, в которой почтовым кодам соответствуют названия регионов
- Для кэширования результатов периодического агрегирования данных
- Для промежуточных результатов при анализе данных

Таблицы Memory поддерживают индексы типа HASH, обеспечивающие очень большую скорость выполнения поисковых запросов. Дополнительная информация об индексах типа HASH приведена в разделе «Хеш-индексы» в главе 3 на стр. 141.

Таблицы Memory работают очень быстро, но не всегда годятся в качестве замены дисковых таблиц. Они используют блокировку на уровне таблицы, что уменьшает конкуренцию при записи, и не поддерживают столбцы типа TEXT и BLOB. Также они допускают использование только строк фиксированного размера, поэтому значения типа VARCHAR сохраняются как значения типа CHAR, что повышает расход памяти.

MySQL внутри себя использует подсистему Memory для хранения промежуточных результатов при обработке запросов, которым требуется временная таблица. Если промежуточный результат становится слишком большим для таблицы Memory или содержит столбцы типа TEXT или BLOB, то MySQL преобразует его в таблицу MyISAM на диске. В следующих главах об этом будет рассказано подробнее.

Многие часто путают таблицы типа Memory с временными таблицами, которые создаются командой CREATE TEMPORARY TABLE. Временные таблицы могут использовать любую подсистему хранения. Это не то же самое, что таблицы типа Memory. Временные таблицы видны только в одном соединении и полностью исчезают при его закрытии.

Подсистема Archive

Подсистема хранения Archive позволяет выполнять только команды INSERT и SELECT и до версии MySQL 5.1 не поддерживала индексирование[1]. Она требует значительно меньше операций дискового ввода/вывода, чем MyISAM, поскольку буферизует записываемые данные и сжимает все вставляемые строки с помощью библиотеки *zlib*. Кроме того, каждый запрос SELECT требует полного сканирования таблицы. По этим причинам таблицы Archive идеальны для протоколирования и сбора данных, когда анализ чаще всего сводится к сканированию всей таблицы, а также в тех случаях, когда требуется обеспечить быстроту выполнения запросов INSERT на главном сервере репликации. Подчиненные серверы репликации могут использовать для той же таблицы другие подсистемы хранения, следовательно существует возможность проиндексировать таблицу на подчиненном сервере для большей производительности при анализе. (Подробнее о репликации рассказано в главе 8.)

Подсистема Archive поддерживает блокировку на уровне строк и специальный системный буфер для вставок с высокой степенью конкурентности. Она обеспечивает согласованные чтения, останавливая выполнение команды SELECT после того, как извлечено столько строк, сколько было в таблице на момент начала выполнения запроса. Она также обеспечивает невидимость результатов пакетной вставки, пока процедура не будет завершена. Эти особенности эмулируют некоторые аспекты по-

[1] **При этом в MySQL 5.1 поддержка индексов довольно существенно ограничена.** – *Прим. науч. ред.*

ведения транзакций и MVCC, но Archive не является транзакционной подсистемой хранения. Она лишь оптимизирована для высокоскоростной вставки и хранения данных в сжатом виде.

Подсистема CSV

Подсистема CSV рассматривает файлы с разделителями-запятыми (*CSV*) как таблицы, но не поддерживает индексы по ним. Она позволяет импортировать и экспортировать данные из CSV-файлов, не останавливая сервер. Если вы экспортируете CSV-файл из электронной таблицы и сохраните в каталоге данных сервера MySQL, то сервер сможет немедленно его прочитать. Аналогично, если вы записываете данные в таблицу CSV, внешняя программа сможет сразу же прочесть его. Таблицы CSV особенно полезны как формат обмена данными и для некоторых типов протоколирования.

Подсистема Federated

Подсистема Federated не хранит данные локально. Каждая таблица типа Federated ссылается на таблицу, расположенную на удаленном сервере MySQL, так что для всех операций она соединяется с удаленным сервером. Иногда ее используют для различных трюков с репликацией.

В текущей реализации этой подсистемы есть много странностей и ограничений. Поэтому мы думаем, что она наиболее полезна для поиска одиночных строк по первичному ключу и для запросов `INSERT`, которые вы хотите выполнить на удаленном сервере. В то же время она плохо подходит для агрегирования, соединений и других базовых операций.

Подсистема Blackhole

В подсистеме Blackhole вообще нет механизма хранения данных. Все команды `INSERT` просто игнорируются. Однако сервер записывает запросы к таблицам Blackhole в журналы как обычно, так что они могут быть реплицированы на подчиненные серверы или просто сохранены в журнале. Это делает подсистему Blackhole полезной для настройки предполагаемых репликаций и ведения журнала аудита.

Подсистема NDB Cluster

Компания MySQL AB приобрела подсистему хранения NDB Cluster у Sony Ericsson в 2003 году. Исходно этот стандарт был создан для высокоскоростной обработки данных (в реальном масштабе времени) с возможностями избыточности и балансировки нагрузки. Хотя журнал велся на диске, все данные хранились в памяти и работа была оптимизирована для поиска по первичному ключу. С тех пор MySQL AB добавила другие методы индексирования и значительно оптимизировала подсистему. Начиная с MySQL 5.1 NDB Cluster позволяет сохранять некоторые столбцы на диске.

Архитектура NDB уникальна: кластер NDB совершенно иной, чем, например, кластер Oracle. Инфраструктура NDB основана на концепции «без разделения ресурсов». Отсутствует какая-либо сеть хранения данных (SAN) или другой подобный централизованный механизм, на котором основаны некоторые прочие типы кластеров. База данных NDB состоит из узлов данных, узлов управления и узлов SQL (экземпляров MySQL). Каждый узел данных хранит сегмент («фрагмент») данных кластера. Фрагменты дублируются, так что система поддерживает несколько копий одних и тех же данных в разных узлах. Для каждого узла обычно выделен один физический сервер с целью обеспечения избыточности и отказоустойчивости. В этом отношении подсистема NDB подобна RAID на уровне сервера.

Узлы управления используются для извлечения централизованной конфигурации, а также для мониторинга и управления узлами кластера. Все узлы данных взаимодействуют друг с другом, а все серверы MySQL (узлы SQL) соединяются со всеми узлами данных. Для NDB Cluster критически важным является минимизация сетевых задержек.

Предупреждение: NDB Cluster – это передовая технология; она, несомненно, заслуживает исследования с целью удовлетворения любопытства, но многие пытаются решать с ее помощью задачи, для которых она не предназначена. По нашему опыту, даже после тщательного изучения данной подсистемы многие все же не понимали, в каких случаях она может быть полезна, по крайней мере, до тех пор, пока не устанавливали ее и не использовали в течение некоторого времени. Обычно это приводило к пустой трате времени, поскольку NDB Cluster просто не предназначена для роли универсального механизма хранения данных.

Обычно вызывает удивление то обстоятельство, что NDB в настоящее время реализует операции соединения (joins) на уровне сервера MySQL, а не на уровне подсистемы хранения. Поскольку все данные для NDB должны извлекаться через сеть, сложные соединения оказываются чрезвычайно медленными. С другой стороны, просмотр одной таблицы может быть очень быстрым, поскольку результат поступает сразу от нескольких узлов, каждый из которых передает свою часть информации. Это только один из многих аспектов, которые следует иметь в виду и хорошо понимать, планируя использовать NDB Cluster в конкретном приложении.

Подсистема NDB Cluster настолько велика и сложна, что мы больше не будем обсуждать ее в дальнейшем. Если вас интересует эта тема, то поищите книгу, специально посвященную ей. Однако скажем, что для большинства традиционных приложений эта подсистема не подходит.

Подсистема Falcon

Джим Старки, один из пионеров в области баз данных, который придумал СУБД Interbase, технологию MVCC и тип столбца BLOB, спро-

ектировал подсистему хранения Falcon. Компания MySQL AB приобрела технологию Falcon в 2006 году, и Джим в настоящее время работает в MySQL AB.

Подсистема Falcon разработана для современного аппаратного обеспечения, в частности для серверов с несколькими 64-разрядными процессорами и большим объемом оперативной памяти, но может функционировать и на более скромной технике. В Falcon используется технология MVCC, причем исполняемые транзакции по возможности целиком хранятся в памяти. Это существенно ускоряет откат и операцию восстановления.

На момент написания книги подсистема Falcon еще не была закончена (например, она пока не синхронизирует фиксацию транзакций с двоичным журналом), поэтому мы не можем писать о ней с достаточной степенью компетентности. Даже проведенные нами предварительные тесты, вероятно, устареют к моменту выхода окончательной версии. Похоже, что у этой подсистемы хороший потенциал для веб-приложений, но окончательные выводы делать пока еще рано.

Подсистема solidDB

Подсистема solidDB, разработанная компанией Solid Information Technology (*http://www.soliddb.com*), предлагает транзакционную обработку данных с использованием технологии MVCC. Она поддерживает как оптимистическое, так и пессимистическое управление конкуренцией, чего нет ни в одной другой подсистеме. Подсистема solidDB для MySQL включает в себя полную поддержку внешних ключей. Она во многом напоминает InnoDB, например в части использования кластерных индексов. В состав solidDB для MySQL включены бесплатные функции оперативного резервного копирования.

Продукт solidDB для MySQL представляет собой законченный пакет, состоящий из подсистем хранения solidDB и MyISAM и самого сервера MySQL. Сопряжение между solidDB и сервером MySQL было представлено в конце 2006 года. Однако базовая технология и код имеют 15-летнюю историю. Компания Solid сертифицирует и поддерживает весь продукт. Он соответствует лицензии GPL и предлагается на коммерческой основе с использованием модели двойного лицензирования, такой же, как и для сервера MySQL.

Подсистема PBXT (Primebase XT)

Подсистема PBXT, разработанная Полом Маккаллоги из гамбургской компании SNAP Innovation GmbH (*http://www.primebase.com*), является транзакционным механизмом хранения данных с уникальным дизайном. Одной из его отличительных характеристик можно назвать способ использования журнала транзакций и файлов данных с целью из-

бежать упреждающей записи в журнал, что значительно уменьшает накладные расходы при фиксации транзакций. Такая архитектура позволяет подсистеме PBXT работать в условиях очень высокой конкуренции на запись, а тесты уже показали, что в некоторых операциях она может быть даже быстрее, чем InnoDB. Подсистема PBXT использует MVCC и поддерживает ограничения внешнего ключа, но не использует кластерные индексы.

PBXT является сравнительно новой технологией, ей еще предстоит проявить себя в реальной работе. Например, реализация по-настоящему долговечных транзакций была закончена только недавно, когда мы писали эту книгу.

Компания SNAP Innovation работает над расширением к PBXT – масштабируемой инфраструктурой «потоковых BLOB» (*blob streaming, http://www.blobstreaming.org*). Она предназначена для эффективного сохранения и извлечения больших блоков двоичных данных.

Подсистема Maria

Maria представляет собой новую подсистему хранения данных, разработанную несколькими ведущими инженерами MySQL, включая Майкла Видениуса, создателя MySQL. Первоначальная версия 1.0 включает в себя только некоторые из запланированных возможностей.

Предполагается, что Maria заменит подсистему хранения MyISAM, которая сейчас применяется в MySQL по умолчанию и используется сервером для выполнения внутренних задач, например для хранения таблицы привилегий и временных таблиц, создаваемых в процессе выполнения запросов. Вот некоторые из планируемых возможностей:

- Выбор транзакционного либо нетранзакционного хранилища на уровне таблицы
- Восстановление после сбоя, даже когда таблицы работают в нетранзакционном режиме
- Блокировка на уровне строк и MVCC
- Улучшенная обработка BLOB

Другие подсистемы хранения

Различные сторонние разработчики предлагают другие (иногда патентованные) подсистемы хранения. Существует множество специализированных и экспериментальных подсистем (например, для запроса к веб-службам). Некоторые из них разрабатываются неофициально, возможно, одним или двумя программистами. Создать подсистему хранения для MySQL сравнительно несложно. Однако большинство таких разработок малоизвестно, отчасти из-за их ограниченной применимости. Мы оставляем вам возможность исследовать их самостоятельно.

Выбор правильной подсистемы хранения

При разработке приложения для MySQL вы должны решить, какую подсистему хранения использовать. Если не задаться этим вопросом на этапе проектирования, то, скорее всего, через какое-то время вы столкнетесь с определенными сложностями. Вы можете обнаружить, что используемая по умолчанию подсистема не обеспечивает нужных возможностей, например транзакций, или может оказаться, что для той совокупности запросов на чтение и запись, которую генерирует ваше приложение, необходимы более детальные блокировки, чем блокировки на уровне таблицы в MyISAM.

Поскольку допустимо выбирать способ хранения данных для каждой таблицы в отдельности, вы должны ясно понимать, как будет использоваться каждая таблица, и какие данные в ней планируется хранить. Это также поможет получить хорошее представление о приложении в целом и о потенциале его роста. Вооружившись этой информацией, вы сможете осознанно выбирать подсистемы хранения данных.

Использование разных подсистем хранения для разных таблиц – не всегда удачное решение. Выбор единого механизма хранения данных для всех таблиц может несколько облегчить вашу жизнь.

Критерии выбора

Хотя на решение о выборе подсистемы хранения могут влиять многие факторы, обычно все сводится к нескольким базовым критериям. Вот основные элементы, которые следует принимать во внимание:

Транзакции

Если вашему приложению требуются транзакции, то InnoDB является наиболее стабильной, хорошо интегрированной, проверенной подсистемой хранения на момент написания книги. Однако мы ожидаем, что со временем появятся транзакционные подсистемы, которые станут сильными соперниками для InnoDB.

MyISAM можно назвать хорошим вариантом, если задача не требует транзакций и в основном предъявляет запросы типа `SELECT` или `INSERT`. Иногда отдельные компоненты приложения (например, протоколирование) попадают в эту категорию.

Конкурентный доступ

Как лучше удовлетворить ваши требования к конкуренции, зависит от рабочей нагрузки. Если вам требуется просто осуществлять в конкурентном режиме операции чтения и вставки, то хотите верьте, хотите нет, MyISAM является прекрасным выбором! Если нужно поддержать совокупность одновременных операций, так чтобы они не искажали результатов друг друга, то хорошо подойдет какая-нибудь подсистема с возможностью блокировок на уровне строки.

Резервное копирование

Необходимость регулярно проводить операции резервного копирования также может повлиять на ваш выбор. Если существует возможность периодически останавливать сервер для выполнения данной процедуры, то подойдет любая подсистема хранения данных. Однако если требуется осуществлять резервное копирование без остановки сервера, выбор уже не так очевиден. Более подробно эта тема обсуждается в главе 11.

Также имейте в виду, что использование различных подсистем хранения увеличивает сложность резервного копирования и настройки сервера.

Восстановление после сбоя

Если объем данных велик, то нужно серьезно оценить, сколько времени займет восстановление базы после сбоя. Таблицы MyISAM обычно чаще оказываются поврежденными и требуют значительно больше времени для восстановления, чем, например, таблицы InnoDB. На практике это одна из самых важных причин, по которым многие используют подсистему InnoDB даже при отсутствии необходимости в транзакциях.

Специальные возможности

Наконец, вы можете обнаружить, что приложению требуются конкретные возможности или оптимизации, которые могут обеспечить только некоторые подсистемы хранения MySQL. Например, многие приложения используют оптимизации кластерных индексов. В настоящее время эту возможность обеспечивают только InnoDB и solidDB. С другой стороны, лишь MyISAM поддерживает полнотекстовый поиск. Если подсистема хранения отвечает одному или нескольким критическим требованиям, но не отвечает другим, то нужно либо найти компромисс, либо выбрать разумное проектное решение. Вы часто можете получить то, что вам нужно, от подсистемы хранения, которая, на первый взгляд, не соответствует вашим требованиям.

Нет необходимости принимать решение немедленно. В оставшейся части книги вы найдете множество материалов, касающихся слабых и сильных сторон каждой подсистемы хранения, а также немало советов по архитектуре и проектированию. В общем, вариантов, вероятно, значительно больше, чем вы можете представить себе сейчас, а дальнейшее чтение поможет вам найти ответ на этот вопрос.

Практические примеры

Все вышесказанное может показаться несколько абстрактным вне контекста реальной практики, так что давайте обратимся к некоторым широко распространенным приложениям баз данных. Мы рассмотрим различные таблицы и определим, какая подсистема хранения лучше

всего отвечает потребностям каждой из них. Итоговую сводку вариантов мы приведем в следующем разделе.

Протоколирование

Предположим, вы хотите использовать MySQL для протоколирования в режиме реального времени всех телефонных звонков с центрального телефонного коммутатора. Или, возможно, вы установили *mod_log_sql* для Apache, чтобы хранить сведения обо всех посещениях веб-сайта прямо в таблице. В таких приложениях обеспечение быстродействия, скорее всего, является самой главной задачей. Вы не хотите, чтобы база данных оказалась узким местом. Механизмы хранения данных MyISAM и Archive подойдут очень хорошо, поскольку у них маленькие накладные расходы, и вы можете осуществлять тысячи операций записи в секунду. Механизм хранения данных PBXT также подходит для задач протоколирования.

Однако все становится гораздо интереснее, когда приходит пора генерировать отчеты на основе накопленных данных. В зависимости от того, какие запросы вы предъявляете, велика вероятность, что сбор данных для отчета значительно замедлит процесс добавления новых записей. Что можно сделать в такой ситуации?

Одним из решений является использование встроенной функции репликации MySQL для клонирования данных на второй (подчиненный) сервер, где затем будут запущены длительные запросы, активно потребляющие ресурсы. Таким образом, главный сервер останется свободным для вставки записей и не нужно будет беспокоиться о том, как создание отчета повлияет на протоколирование в реальном времени.

Можно также запускать запросы в периоды низкой нагрузки, но по мере эволюции приложения эта стратегия может стать неработоспособной.

Другим вариантом является использование объединенной таблицы (подсистема хранения Merge). Вместо того чтобы всегда писать в одну и ту же таблицу, настройте приложение так, чтобы каждый протокол сохранялся в таблице, имя которой составлено из года и названия или номера месяца, например `web_logs_2008_01` или `web_logs_2008_jan`. Затем определите объединенную таблицу, предназначенную для хранения подлежащих агрегированию данных, и используйте ее в запросах. Если требуется агрегировать данные ежедневно или еженедельно, применяйте ту же стратегию, только имена таблиц станут более подробными, например `web_logs_2008_01_01`. Если вы будете обращаться к таблицам, в которые уже не производится запись, то приложение сможет сохранять новые данные протокола в текущую таблицу без помех.

Таблицы только для чтения или в основном для чтения

Таблицы, содержащие данные, которые используются для создания каталога или списка (вакансии, аукционы, недвижимость и т. п.), обычно отличаются тем, что считывание из них происходит значительно

чаще, чем запись. Такие таблицы являются хорошими кандидатами для MyISAM – если забыть о том, что происходит при сбое MyISAM. Постарайтесь избежать недооценки того, насколько это важно. Многие пользователи не понимают, как рискованно использовать подсистему хранения, которая не очень-то и пытается гарантировать надежную запись на диск.

Очень полезно запустить имитацию реальной нагрузки на тестовом сервере, а затем в прямом смысле слова выдернуть вилку электропитания из розетки. Личный опыт восстановления данных после сбоя бесценен. Он убережет от неприятных сюрпризов в будущем.

Только не доверяйте народной мудрости «MyISAM быстрее, чем InnoDB». Категоричность этого утверждения спорна. Мы можем перечислить десятки ситуаций, когда InnoDB повергает MyISAM в прах, особенно в приложениях, где находят применение кластерные индексы или данные целиком размещаются в памяти. В оставшейся части книги вы увидите, какие факторы влияют на производительность подсистемы хранения (размер данных, требуемое количество операций ввода/вывода, первичные ключи и вторичные индексы и т. п.), и какие из них имеют отношение к вашему приложению.

Обработка заказов

При обработке заказов транзакции требуются почти всегда. Законченный наполовину заказ вряд ли обрадует ваших клиентов. Другим важным обстоятельством является то, поддерживает ли подсистема ограничения внешнего ключа. На момент написания этой книги InnoDB, вероятно, была оптимальным выбором для приложений обработки заказов, хотя в качестве кандидата можно рассматривать любую подсистему хранения.

Биржевые котировки

Если вы собираете биржевые котировки для последующего анализа, вам подойдет механизм MyISAM (следует учитывать его обычные тонкие места). Однако если вы используете веб-службу с большим трафиком, которая получает котировки в режиме реального времени и имеет тысячи пользователей, длительное ожидание результатов недопустимо. Многие клиенты будут одновременно пытаться осуществлять чтение из таблицы и запись в нее, поэтому необходима блокировка на уровне строк или проектное решение, минимизирующее количество операций обновления.

Доски объявлений и дискуссионные форумы

Тематические дискуссии являются интересной задачей для пользователей MySQL. Существуют сотни бесплатных систем, написанных на языках PHP и Perl, которые позволяют организовывать на сайтах те-

матические дискуссии. Многие из них созданы без упора на эффективное использование базы данных, в результате чего они имеют обыкновение запускать кучу запросов для каждого обслуживаемого обращения. Некоторые разработаны с намерением обеспечить независимость от типа используемой базы данных, поэтому генерируемые ими запросы не задействуют специфические возможности конкретной СУБД. Нередко подобные системы обновляют счетчики и собирают статистику различных дискуссий. Зачастую они используют для хранения всех своих данных небольшое число монолитных таблиц. В результате несколько центральных таблиц оказываются загруженными операциями записи и чтения, а блокировки, необходимые для обеспечения целостности данных, становятся постоянным источником конфликтов.

Несмотря на недостатки проектирования, большинство таких систем при малых и средних нагрузках работает хорошо. Однако если веб-сайт становится достаточно большим и генерирует значительный трафик, скорость его работы заметно снижается. Очевидным решением этой проблемы является переход на другую подсистему хранения данных, которая может обслуживать больше операций чтения и записи, но иногда пользователи, пытающиеся пойти по этому пути, бывают удивлены тем, что система начинает работать еще медленнее!

Пользователи могут не понимать, что приложение запускает достаточно специфические запросы, например, вот такого вида:

```
mysql> SELECT COUNT(*) FROM table;
```

Проблема заключается в том, что не все подсистемы хранения исполняют такие запросы быстро: MyISAM на это способна, другие – не всегда. Для каждой подсистемы можно найти подобные примеры. В главе 2 вы узнаете, как выявлять и решать подобные проблемы.

Приложения на компакт-дисках

Если вам когда-нибудь потребуется распространять использующие файлы данных MySQL приложения на компакт-дисках, подумайте о применении таблиц типа MyISAM или сжатых таблиц MyISAM, которые можно легко изолировать и скопировать на другой носитель. Сжатые таблицы MyISAM занимают значительно меньше места, чем несжатые, но они предназначены только для чтения. В некоторых приложениях это может вызвать определенные проблемы, но, поскольку данные все равно предназначены для записи на носитель, поддерживающий только чтение, нет оснований избегать использования сжатых таблиц для этой конкретной задачи.

Сводка подсистем хранения

В табл. 1.3 приведены характеристики наиболее популярных подсистем хранения данных в MySQL, относящиеся к транзакциям и блокировкам. В столбце *Версия MySQL* указана минимальная версия MySQL,

Таблица 1.3. Сводка подсистем хранения MySQL

Подсистема хранения	Версия MySQL	Транзакции	Детальность блокировок	Типичные приложения	Противопоказания
MyISAM	Все	Нет	Табличная с конкурентными вставками	SELECT, INSERT, пакетная загрузка	Смешанная нагрузка чтение/запись
MyISAM Merge	Все	Нет	Табличная с конкурентными вставками	Сегментированное архивирование, хранилища данных	Большое количество глобальных поисков
Memory (HEAP)	Все	Нет	Табличная	Промежуточные вычисления, статические справочные данные	Большие наборы данных, постоянное хранение
InnoDB	Все	Да	На уровне строки с MVCC	Транзакционная обработка	Нет
Falcon	6.0	Да	На уровне строки с MVCC	Транзакционная обработка	Нет
Archive	4.1	Нет	На уровне строки	Протоколирование, агрегирование	Потребность в произвольной выборке, обновления, удаления
CSV	4.1	Нет	Табличная	Протоколирование, пакетная загрузка внешних данных	Потребность в произвольной выборке, индексирование
Blackhole	4.1	N/A	N/A	Протоколируемое или реплицируемое архивирование	Все, кроме того, для чего предназначена
Federated	5.0	N/A	N/A	Распределенные источники данных	Все, кроме того, для чего предназначена
NDB Cluster	5.0	Да	На уровне строки	Высокая доступность	Большинство типичных применений
PBXT	5.0	Да	На уровне строки с MVCC	Транзакционная обработка, протоколирование	Необходимость в кластерных индексах
solidDB	5.0	Да	На уровне строки с MVCC	Транзакционная обработка	Нет
Maria (планируется)	6.x	Да	На уровне строки с MVCC	Замена MyISAM	Нет

необходимая для использования соответствующей подсистемы, хотя для некоторых подсистем и версий MySQL может потребоваться скомпилировать свой собственный сервер. Слово «Все» в этом столбце означает все версии, начиная с MySQL 3.23.

Преобразования таблиц

Есть несколько способов преобразования таблицы из одной подсистемы хранения в другую, каждый со своими достоинствами и недостатками. В следующих разделах мы рассмотрим три наиболее общих способа.

ALTER TABLE

Самым простым способом изменить тип таблицы является команда `ALTER TABLE`. Следующая команда преобразует таблицу `mytable` к типу Falcon:

```
mysql> ALTER TABLE mytable ENGINE = Falcon;
```

Этот синтаксис работает для всех подсистем хранения, но есть одна тонкость: использование такого метода может занять много времени. MySQL будет осуществлять построчное копирование старой таблицы в новую. В ходе этой операции, скорее всего, будет задействована вся пропускная способность диска сервера, а исходная таблица окажется заблокированной на чтение. Поэтому будьте осторожны при использовании этого подхода для активно используемых таблиц. Вместо него можно применить один из перечисленных ниже методов, которые сначала создают копию таблицы.

При изменении подсистемы хранения все специфичные для старой подсистемы возможности теряются. Например, после преобразования таблицы InnoDB в MyISAM, а потом обратно будут потеряны все внешние ключи, определенные в исходной таблице InnoDB.

Экспорт и импорт

Чтобы получить больший контроль над процессом преобразования, вы можете сначала экспортировать таблицу в текстовый файл с помощью утилиты *mysqldump*. После этого можно будет просто изменить команду `CREATE TABLE` в этом текстовом файле. Не забудьте отредактировать название таблицы и ее тип, поскольку нельзя иметь две таблицы с одним и тем же именем в одной и той же базе данных, даже если у них разные типы – а *mysqldump* по умолчанию пишет команду `DROP TABLE` перед командой `CREATE TABLE`, так что вы можете потерять свои данные, если не будете осторожны!

Дополнительные советы по эффективному экспорту и загрузке данных вы найдете в главе 11.

Команды CREATE и SELECT

Третий способ преобразования являет собой компромисс между скоростью первого и безопасностью второго. Вместо того чтобы экспортировать всю таблицу или преобразовывать ее за один прием, создайте новую таблицу и используйте команду MySQL `INSERT ... SELECT` для ее заполнения следующим образом:

```
mysql> CREATE TABLE innodb_table LIKE myisam_table;
mysql> ALTER TABLE innodb_table ENGINE=InnoDB;
mysql> INSERT INTO innodb_table SELECT * FROM myisam_table;
```

Этот способ работает хорошо, если данных немного. Но если объем данных велик, то зачастую оказывается гораздо эффективнее заполнять таблицу частями, фиксируя транзакцию после каждой части, чтобы журнал отмены не становился слишком большим. Предполагая, что `id` является первичным ключом, запустите этот запрос несколько раз (каждый раз используя все большие значения `x` и `y`), пока не скопируете все данные в новую таблицу:

```
mysql> START TRANSACTION;
mysql> INSERT INTO innodb_table SELECT * FROM myisam_table
    -> WHERE id BETWEEN x AND y;
mysql> COMMIT;
```

После этого останутся исходная таблица, которую можно удалить, и новая полностью заполненная таблица. Не забудьте заблокировать исходную таблицу, если хотите предотвратить получение несогласованной копии данных!

2

Поиск узких мест: эталонное тестирование и профилирование

Итак, у вас возникла необходимость повысить производительность MySQL. Но что пытаться улучшить? Конкретный запрос? Схему? Оборудование? Единственный способ узнать это – оценить, что именно делает ваша система, и протестировать ее производительность в разных условиях. Вот почему эта глава стала одной из первых в книге.

Наилучшей стратегией являются поиск и усиление самого слабого звена в цепи компонентов вашего приложения. Это особенно полезно, если вы не знаете, что конкретно мешает достижению более высокой производительности сейчас или может помешать в будущем.

Эталонное тестирование (*benchmarking*) и *профилирование* (*profiling*) – вот два важнейших метода определения узких мест. Они связаны друг с другом, но между ними есть разница. Эталонное тестирование измеряет производительность системы. Это в свою очередь позволяет определить пропускную способность, понять, какие изменения существенны, а какие нет, и выяснить, как производительность приложения зависит от характера данных.

В свою очередь, профилирование помогает найти места, где приложение тратит больше всего времени и потребляет больше всего ресурсов. Другими словами, эталонное тестирование отвечает на вопрос «Насколько хороша производительность?», а профилирование – на вопрос «Почему производительность именно такова?»

Мы разбили эту главу на две части: первая посвящена эталонному тестированию, вторая – профилированию. Мы начнем с обсуждения причин и стратегий эталонного тестирования, а затем перейдем к вопросам конкретной тактики. Мы покажем, как спланировать и спроектировать эталонные тесты, позволяющие получить точные результаты, как их выполнять и как анализировать результаты. Мы закончим первую

часть рассмотрением инструментов эталонного тестирования и приведем примеры использования некоторых из них.

В оставшейся части главы будет показано, как профилировать приложения и MySQL. Мы продемонстрируем подробные примеры кода профилирования из реальной жизни, которые мы использовали для анализа производительности приложений. Мы также покажем, как протоколировать запросы MySQL и анализировать журналы, как использовать счетчики состояния MySQL и другие инструменты для выяснения того, что же делают MySQL и ваши запросы.

Почему нужно тестировать производительность?

Во многих средних и крупных компаниях, где используется MySQL, имеются специальные сотрудники, занимающиеся эталонным тестированием. Однако каждый разработчик и администратор баз данных должен быть знаком с основными принципами и методами эталонного тестирования, поскольку они полезны во многих ситуациях. Вот несколько областей применения эталонного тестирования.

- Измерить производительность приложения в текущий момент. Если вы не знаете, насколько быстро оно работает, то не можете быть уверены в полезности внесенных изменений. Вы также можете использовать историю результатов тестирования для диагностики непредвиденных проблем.
- Подтвердить возможность масштабирования системы. Вы можете использовать эталонные тесты для эмуляции гораздо большей нагрузки, чем та, которую испытывает система сейчас. Например, вы можете узнать, что произойдет в случае тысячекратного увеличения количества пользователей.
- Планирование роста. Эталонные тесты помогут оценить, какое оборудование, пропускная способность сети или другие ресурсы потребуются при планируемом увеличении нагрузки. Это поможет уменьшить риски при модернизации или серьезных изменениях в приложении.
- Тестирование способности приложения переносить изменения среды. Например, вы можете выяснить, как работает ваше приложение в случае спорадических всплесков количества параллельно работающих пользователей, с различными конфигурациями серверов или с различным распределением данных.
- Тестирование различных конфигураций оборудования, программного обеспечения и операционной системы. Что лучше для вашей системы – RAID 5 или RAID 10? Как меняется производительность произвольной записи при переходе с дисков ATA на сеть хранения SAN? Лучше ли масштабируется ядро Linux 2.4, чем 2.6? Увеличит ли производительность обновление версии MySQL? Что будет, если

использовать другую подсистему хранения данных? Ответы на все эти вопросы вы можете получить с помощью специальных эталонных тестов.

Вы можете также использовать эталонное тестирование для других целей, например, создать набор ориентированных на конкретное приложение модульных тестов (unit test), но здесь мы сосредоточимся только на аспектах, связанных с производительностью.

Стратегии эталонного тестирования

Существуют две основные стратегии тестирования производительности: тестировать приложение целиком или только аспекты, относящиеся к MySQL. Эти стратегии соответственно называются *полным* и *покомпонентным* тестированием. Есть несколько причин для измерения производительности приложения в целом вместо тестирования только MySQL.

- Вы тестируете все приложение, включая веб-сервер, код приложения и базу данных. При этом вас интересует не только производительность MySQL, а производительность всей системы.
- MySQL не всегда является узким местом приложения и полное тестирование позволяет это обнаружить.
- Вы можете увидеть, как ведет себя кэш каждой части, только в ходе полного тестирования.
- Эталонные тесты хороши лишь в той мере, в какой они отражают реальное поведение приложения, чего трудно добиться при тестировании его по частям.

С другой стороны, эталонные тесты приложения сложно создавать и еще сложнее правильно настраивать. Если вы плохо спроектируете тест, то можете прийти к неверным выводам, поскольку полученные подобным образом результаты не отражают реальности.

Однако иногда нет необходимости собирать информацию обо всем приложении. Вам может потребоваться просто протестировать производительность MySQL, по крайней мере, на начальном этапе. Такое эталонное тестирование полезно, если:

- Вы хотите сравнить различные схемы или запросы
- Вы хотите протестировать конкретную проблему, обнаруженную в приложении
- Вы хотите избежать длительных тестов, ограничившись коротким тестом, который позволит быстро измерить результаты внесенных изменений

Кроме того, эталонное тестирование MySQL полезно, когда вы выполняете характерные для своего приложения запросы на реальном наборе данных. Как сами данные, так и размер набора должны быть реа-

листичными. По возможности используйте мгновенный снимок реальных рабочих данных.

К сожалению, настройка реалистичного эталонного теста может оказаться сложным и длительным делом, поэтому, если вы можете получить копию рабочего набора данных, считайте себя счастливчиком. Конечно, это может оказаться невозможным – например, если вы создаете новое приложение, которым пользуется мало людей и в котором еще недостаточно данных. Если вы хотите знать, как оно будет работать, когда разрастется, то у вас нет другого выбора, кроме как сгенерировать больший объем данных и нагрузку.

Что измерять

Перед началом тестирования нужно определить цели – собственно, это следует сделать даже до начала проектирования тестов. Зная, к чему вы стремитесь, можно будет выбрать инструменты и методики для получения точных, осмысленных результатов. Сформулируйте цели в виде вопросов, например «лучше ли этот процессор, чем тот?» или «будут ли новые индексы работать эффективнее, чем нынешние?».

Возможно, это не покажется вам очевидным, но иногда требуются разные подходы, чтобы измерить разные вещи. Например, для измерения сетевых задержек и пропускной способности нужны различные эталонные тесты.

Рассмотрите следующие показатели и подумайте, как они соответствуют вашим целям в плане увеличения производительности:

Количество транзакций в единицу времени

Это один из классических эталонных тестов приложений баз данных. Стандартизованные тесты типа TPC-C (см. *http://www.tpc.org*) упоминаются очень широко, и многие производители СУБД активно работают над тем, чтобы улучшить характеристики своих продуктов в этих тестах. Упомянутые тесты измеряют производительность оперативной обработки транзакций (OLTP) и лучше всего подходят для интерактивных многопользовательских приложений. Общепринятой единицей измерения является количество транзакций в секунду.

Термином *пропускная способность* обычно обозначают количество транзакций (или других единиц работы) в единицу времени.

Время отклика или задержки

Этот показатель дает представление об общем времени исполнения задачи. В зависимости от приложения может потребоваться измерять время в миллисекундах, секундах или минутах. Отсюда можно определить среднее, минимальное и максимальное время отклика.

Максимальное время отклика редко бывает полезной метрикой, поскольку чем дольше тест работает, тем выше, скорее всего, будет зна-

чение этого показателя. Кроме того, оно наверняка будет варьироваться в разных прогонах теста. По этой причине вместо максимального времени отклика зачастую измеряют время отклика в процентилях. Например, если время отклика составляет 5 миллисекунд с процентилем 95%, значит в 95% случаев задача будет выполнена за время не более 5 миллисекунд.

Обычно имеет смысл представить результаты тестов либо в виде линейного графика (например, среднее и процентиль 95), либо в виде диаграммы разброса, чтобы было видно, как распределены результаты. Эти графики покажут, как будут вести себя тесты при длительных испытаниях.

Предположим, что система каждый час записывает контрольную точку в течение одной минуты. В это время никакие транзакции не завершаются. Время отклика в виде процентиля 95 не покажет всплесков, поэтому результаты скроют проблему. Однако на графике будут видны периодические всплески времени отклика. Рисунок 2.1 иллюстрирует этот момент.

На рис. 2.1 изображено количество транзакций в минуту (NOTPM). Мы видим значительные провалы, которые при анализе среднего времени (пунктирная линия) вообще не заметны. Причиной первого провала является отсутствие данных в кэше сервера (некоторое время требуется на «разогрев» кэша). Другие провалы показывают моменты, когда сервер тратит время на интенсивный сброс «грязных» страниц на диск. Без графика эти отклонения было бы трудно увидеть.

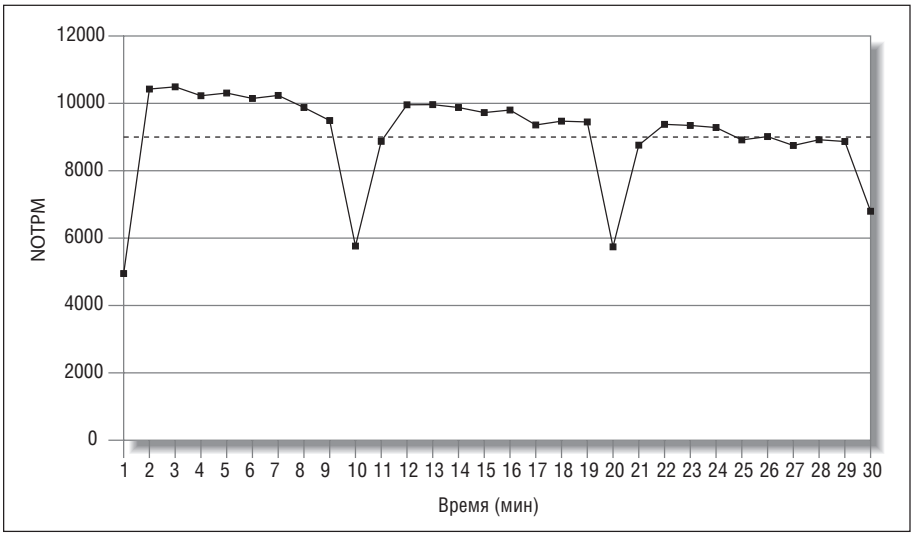


Рис. 2.1. *Результаты 30-минутной работы теста dbt2*

Масштабируемость

Измерение масштабируемости полезно для систем, в которых необходимо поддерживать стабильную производительность даже в условиях меняющейся нагрузки.

«Производительность при переменной нагрузке» – это чрезвычайно абстрактная концепция. Производительность обычно измеряется такими параметрами, как пропускная способность или время отклика, а рабочая нагрузка может меняться в результате изменений размера базы данных, количества одновременных соединений или используемого оборудования.

Измерения масштабируемости хороши для планирования потенциала, поскольку они помогают выявить слабые места вашего приложения, которые нельзя обнаружить с помощью других стратегий. Например, если при проектировании системы выполнялось тестирование времени отклика для одного соединения (плохая стратегия эталонного тестирования), то в случае нескольких одновременных соединений производительность может оказаться низкой. Тесты, проверяющие постоянство времени отклика при увеличении количества соединений, выявят эту ошибку проектирования.

Некоторые действия, такие как пакетные задания для создания сводных таблиц, как раз требуют быстрого времени отклика. Протестировать их на время отклика нужно, но не забудьте продумать, как они будут взаимодействовать с другими сессиями. Пакетные задания могут оказать негативное влияние на интерактивные запросы и наоборот.

Уровень конкуренции

Уровень конкуренции является важным параметром, но нередко его плохо понимают и неправильно используют. Например, принято говорить о количестве пользователей, просматривающих веб-сайт в конкретный момент времени. Однако в протоколе HTTP нет информации о состоянии (stateless), и если большинство пользователей просто читает содержимое окна броузера, это не вызывает конкуренции на веб-сервере. Аналогично, конкуренция на веб-сервере не обязательно означает наличие конкуренции в базе данных. Единственное, с чем конкуренция непосредственно связана, – это объем данных, с которым должен справляться ваш механизм хранения сеансов. Более точная мера измерения конкуренции на веб-сервере – количество запросов в секунду, которые пользователи генерируют в пиковые периоды.

Кроме того, вы можете измерять конкуренцию в различных местах приложения. Более высокая конкуренция на веб-сервере может вызывать более высокую конкуренцию на уровне базы данных, но на нее может повлиять также язык программирования и набор используемых инструментов. Например, язык Java с пулом соединений, вероятно, вызовет меньшее количество одновременных соединений

с сервером MySQL, чем язык PHP с использованием постоянных соединений.

Более важным параметром является количество соединений, в которых исполняются запросы в данный момент времени. Хорошо спроектированное приложение может иметь сотни открытых соединений с сервером MySQL, но только очень небольшая их часть в состоянии поддерживать одновременно исполняющиеся запросы. Таким образом, веб-сайт с «50 000 посетителей одновременно» порой обходится 10 или 15 одновременно выполняющимися запросами к серверу MySQL!

Другими словами, при тестировании в действительности нужно озаботиться *рабочей конкуренцией*, то есть количеством потоков или соединений, выполняющих работу одновременно. Измерьте, не падает ли производительность при увеличении конкуренции. Если да, то ваше приложение вряд ли сможет выдерживать всплески нагрузки.

Нужно либо убедиться, что производительность не падает резко, или спроектировать приложение так, чтобы оно не создавало сильную конкуренцию в тех частях, которые не могут ее выдержать. В общем случае стоит ограничить конкуренцию на сервере MySQL, используя такие методы, как организация очередей на уровне приложений. Подробнее об этом рассказано в главе 10.

Конкуренция – это показатель, совершенно отличный от масштабируемости и времени реакции. По сути это даже не *результат*, а скорее *параметр* теста. Вместо измерения уровня конкуренции, которого достигает ваше приложение, вы измеряете производительность приложения при различных значениях параметра конкуренции.

В итоговом анализе нужно протестировать то, что важно для пользователей. Тесты измеряют производительность, но для разных людей понятие «производительность» имеет несхожий смысл. Соберите различные требования (формально или неформально) к тому, как система должна масштабироваться, каково допустимое время реакции, какой вид конкуренции ожидается, и т. п. Затем попытайтесь спроектировать тесты так, чтобы принять во внимание все требования, не ограничивая угол зрения и не фокусируясь на одних аспектах в ущерб другим.

Тактики эталонного тестирования

Закончив с общими вопросами, давайте перейдем к конкретике – как проектировать и выполнять эталонные тесты. Однако прежде мы предлагаем рассмотреть самые распространенные ошибки, которые могут привести к непригодным или неточным результатам:

- Использование набора данных, имеющего объем, несоизмеримый с рабочими объемами, например использование одного гигабайта данных, когда приложение должно будет обрабатывать сотни гига-

байтов, или использование текущего набора данных, когда вы предполагаете, что приложение будет быстро расти.

- Использование данных с неправильным распределением, например равномерно распределенных, когда в реальных данных будут встречаться «горячие точки». (Вообще, сгенерированные случайным образом данные часто имеют распределение, не имеющее отношения к реальности).
- Использование нереалистично распределенных параметров, например в предположении, что частота просмотра всех профилей пользователей будет одинакова.
- Использование однопользовательского сценария для многопользовательского приложения.
- Тестирование распределенного приложения на единственном сервере.
- Несоответствие реальному поведению пользователя, например неверное время просмотра одной страницы. Реальные пользователи запрашивают страницу, а потом читают ее. Они не щелкают по ссылкам без остановки.
- Выполнение идентичных запросов в цикле. Реальные запросы неодинаковы, так что они могут запрашивать данные не из кэша. Идентичные запросы будут полностью или частично кэшированы на каком-то уровне.
- Отсутствие контроля ошибок. Если результаты теста не имеют смысла – например, медленная операция внезапно очень быстро заканчивается, – ищите ошибку. Вы можете просто протестировать, насколько быстро MySQL может обнаружить синтаксическую ошибку в запросе SQL! Возьмите за правило всегда проверять после выполнения теста журнал ошибок.
- Игнорирование проверки работы системы сразу после ее запуска или перезагрузки. Иногда нужно знать, как быстро ваш сервер наберет «полную мощность» после перезагрузки. Наоборот, если вы предполагаете изучить производительность в нормальном режиме, убедитесь, что на результаты тестирования не повлияет «неразогретый» кэш.
- Использование установок сервера по умолчанию. Оптимизация настроек сервера описана в главе 6.

Избегая этих ошибок, вы значительно сократите путь к достижению высокого качества результатов.

При прочих равных условиях необходимо стараться сделать тесты как можно более реалистичными. Однако иногда имеет смысл использовать немного нереалистичные тесты. Например, предположим, что приложение находится не на том же компьютере, где размещен сервер базы данных. Более реалистичным будет запустить тесты в той же конфигурации, но это добавит факторы, показывающие, например, насколько

быстра сеть и насколько она загружена. Тестирование производительности на одном компьютере обычно проще и в некоторых случаях дает достаточно точные результаты. Но только вы можете решить, когда такой подход допустим.

Проектирование и планирование тестов

Первым шагом в планировании тестов является определение проблемы и цели. Затем нужно решить, использовать ли стандартный тест или разработать свой собственный.

Если вы используете стандартный тест, выберите тот, который соответствует вашим потребностям. Например, не используйте TPC-тесты для тестирования системы электронной коммерции. По словам создателей TPC-тестов, они «ориентированы на системы принятия решений, в которых обрабатываются большие объемы данных». Следовательно, они не подходят для систем оперативной обработки транзакций.[1]

Разработка собственного эталонного теста является сложным итеративным процессом. Для начала сделайте мгновенный снимок рабочего набора данных. Позаботьтесь о том, чтобы была возможность восстановить эти данные для последующих запусков теста.

Затем вам потребуются запросы к данным. Вы можете превратить комплект модульных тестов (unit test suite) в простейший эталонный тест, просто прогнав их несколько раз, но вряд ли это соответствует тому, как используется база данных на самом деле. Лучше записать все запросы на рабочей системе в течение репрезентативного отрезка времени, например в течение часа в период пиковой нагрузки или в течение всего дня. Если вы запишите запросы за малый промежуток времени, то может потребоваться несколько таких периодов. Это покроет все действия системы, например запросы для еженедельного отчета и пакетные задания, которые вы запланировали на часы минимальной загрузки[2].

Вы можете производить запись запросов на нескольких уровнях. Например, если требуется выполнить эталонное тестирование всего стека протоколов, то можно протоколировать HTTP-запросы на веб-сервере. Можно также включить журнал запросов MySQL, но если вы будете воспроизводить запросы из журнала, не забудьте воссоздать отдельные потоки, а не просто последовательно повторяйте запросы один за другим. Также важно создавать для каждого соединения в журнале отдельный поток, а не хаотично выполнять запросы в произвольных потоках. В журнале запросов видно, на каком соединении был выполнен каждый запрос.

---

[1] На самом деле существуют разные тесты TPC: тест TPC-H рассчитан, как и пишет автор, на тестирование систем принятия решений. Однако также существует тест **TPC-C, который предназначен именно для систем оперативной обработки транзакций**. – *Прим. науч. ред.*

[2] Конечно, все это важно, если вы хотите получить идеальные тесты. Реальная жизнь обычно вносит коррективы.

Даже если вы не строите свой собственный эталонный тест, все равно нужно подготовить план тестирования. Вам придется выполнять тесты много раз, поэтому необходимо иметь возможность повторять их в точности. Составьте также план на будущее. В следующий раз тест может запускать кто-то другой, но даже если это будете вы, то не обязательно вспоминать точно, как именно запускали его раньше. Ваш план должен включать в себя тестовые данные, шаги, предпринятые для настройки системы, и план «прогрева» сервера.

Разработайте методику документирования параметров и результатов и тщательно протоколируйте каждый прогон. Метод документирования может быть как очень простым, например запись в электронную таблицу, так и более сложным – с базы данных (имейте в виду, что вы, вероятно, захотите написать какие-то сценарии для упрощения анализа результатов, так что чем проще будет обрабатывать результаты без открытия электронной таблицы и текстовых файлов, тем лучше).

Может оказаться полезным создать тестовый каталог с подкаталогами для записи результатов каждого прогона теста. Вы сможете помещать в каждый такой подкаталог итоги тестирования, файлы настроек и комментарии. Если тест позволяет измерять не только то, что, по вашему мнению, необходимо, все равно записывайте дополнительную информацию. Лучше иметь лишние данные, чем пропустить важные, а в будущем эта информация может оказаться полезной. Старайтесь записывать во время тестирования как можно больше дополнительных сведений, например статистику использования процессора, подсистемы дискового ввода/вывода и сети, счетчики, выведенные командой `SHOW GLOBAL STATUS`, и т. п.

Получение точных результатов

Лучшим способом получить точные результаты является разработка теста таким образом, чтобы он отвечал именно на те вопросы, которые вы хотите задать. Тот ли тест вы выбрали? Собираются ли данные, которые нужны для ответа на поставленный вопрос? Не происходит ли тестирование по неправильным критериям? Например, не запускаете ли вы тест, нагружающий процессор, для прогнозирования производительности приложения, которое заведомо будет ограничено пропускной способностью систем ввода/вывода?

Затем убедитесь в повторяемости результатов тестирования. Попытайтесь приводить систему в одно и то же состояние перед каждым прогоном теста. Если тест особо важен, перезагружайте компьютер каждый раз прежде, чем запустить тест. Если нужно прогнать тест на «прогретом» сервере, что является нормой, то нужно удостовериться, что сервер «прогревался» достаточно долго и что эта процедура воспроизводима. Например, если «прогрев» достигается отправкой случайных запросов, то результаты теста нельзя будет повторить.

Если тест изменяет данные или схему, восстанавливайте их из снимка между прогонами теста. Вставка в таблицу с тысячью строк не даст того же результата, что вставка в таблицу с миллионом строк! Фрагментация и расположение данных на диске также делают результаты невоспроизводимыми. Одним из способов гарантировать, что физическое расположение будет почти таким же, является быстрое форматирование и копирование раздела.

Следите за внешней нагрузкой, системами профилирования и мониторинга, подробной записью в журналы, периодическими заданиями и прочими факторами, которые могут исказить результаты. Типичным сюрпризом является запуск задания *cron* в середине теста, цикл Patrol Read (фоновый поиск плохих кластеров средствами RAID-контроллеров Intel) или запланированная проверка целостности RAID-массива. Убедитесь, что все необходимые для теста ресурсы на время его исполнения выделены только ему. Если сеть загружена еще какой-то работой или тест запущен на SAN-устройстве, которое одновременно используется и другими серверами, то результаты могут оказаться неточными.

Постарайтесь при каждом прогоне теста менять как можно меньше параметров. Если модифицировать сразу несколько факторов, то есть риск что-то упустить. Параметры также могут зависеть друг от друга, поэтому в ряде случаев их нельзя изменять независимо. Иногда не известно даже о существовании такой зависимости, что лишь усложняет дело.[1]

Лучше всего менять параметры тестирования итеративно, а не делать серьезных изменений между прогонами. Например, для подбора подходящего значения параметра сервера используйте технику «разделяй и властвуй» (уменьшение или увеличение значения параметра вдвое на каждой последующей итерации тестирования).

Мы встречали множество эталонных тестов, которые пытаются предсказать производительность после миграции, например с Oracle на MySQL. Зачастую результаты этих тестов ненадежны, поскольку MySQL эффективно работает с совершенно другими типами запросов, чем Oracle. Если вы хотите узнать, насколько хорошо написанное для Oracle приложение будет работать после миграции на MySQL, то вам, скорее всего, придется перепроектировать схему и запросы с учетом специфики MySQL. (Иногда, например, при разработке кроссплатформенного приложения, хочется знать, как одни и те же запросы будут работать на обеих платформах, но это редкий случай.)

В любом случае нельзя получить осмысленные результаты с параметрами настройки MySQL по умолчанию, поскольку они рассчитаны на небольшие приложения, потребляющие очень мало памяти.

---

[1] Это не всегда имеет значение. Например, если вы думаете о переходе с системы Solaris, работающей с оборудованием SPARC, на платформу x86 под управлением GNU/Linux, то нет никакого смысла тестировать систему на платформе Solaris/x86 в качестве промежуточного шага!

Наконец, если вы получите странный результат, не следует просто отклонять его. Разберитесь и постарайтесь выяснить, что произошло. Вы можете обнаружить ценную информацию, серьезную проблему или дефект при проектировании теста.

Прогон теста и анализ результатов

После того как вы все подготовили, пора запускать тест, чтобы начать сбор и анализ данных.

Обычно имеет смысл автоматизировать прогоны теста. Это улучшит результаты и их точность, поскольку не позволит пропустить по забывчивости какие-то шаги и не допустит различий в методике прогона теста. Заодно это поможет документировать весь процесс.

Подойдет любой метод автоматизации, например Makefile или набор сценариев. Выберите, какой язык сценариев вам подходит: shell, PHP, Perl и т. п. Попытайтесь наиболее полно автоматизировать процедуру тестирования, включая загрузку данных, «прогрев» системы, запуск теста и запись результатов.

Если вы все правильно настроите, то тестирование может стать одношаговым процессом. Если вы просто запускаете одноразовый тест, чтобы быстро проверить что-то, нет смысла автоматизировать его.

Обычно эталонный тест выполняют несколько раз. Конкретное количество итераций зависит от методологии оценки результатов и от того, насколько эти результаты важны. Если требуется большая уверенность в итогах тестирования, то нужно прогнать тест большее количество раз. Обычно для оценки применяют такие методики: нахождение лучшего результата, вычисление среднего значения по всем полученным итогам или просто выполнение теста пять раз и вычисление среднего результата по трем лучшим. Степень точности определять вам. Можно применять к результатам статистические методы, найти доверительный интервал и т. д., но обычно такой уровень уверенности не требуется[1]. Если тест отвечает на интересующие вас вопросы, можно просто прогнать его несколько раз и посмотреть, насколько различаются полученные значения. Если различие велико, следует либо увеличить количество прогонов, либо запустить тест на больший период времени, вследствие чего разброс обычно уменьшается.

Собранные результаты надо проанализировать, то есть превратить числа в знания. Целью является получение ответов на вопросы, ради кото-

[1] Если действительно нужны научно достоверные, скрупулезные результаты, то рекомендуем почитать хорошую книгу по разработке и выполнению тестов в контролируемых условиях, поскольку данная тема намного шире, чем мы можем изложить здесь.

рых готовилась тестовая инфраструктура. Идеально было бы сформулировать на выходе утверждения типа «модернизация сервера до четырехпроцессорного увеличит производительность на 50% при той же задержке» или «использование индексов ускорило выполнение запросов».

Методика обработки полученных значений зависит от того, как они собирались. Вероятно, имеет смысл написать сценарии для анализа результатов – и не только для того, чтобы уменьшить объем вашей собственной работы, но и по тем же причинам, по которым следует автоматизировать сам тест: воспроизводимость и документирование.

Инструменты эталонного тестирования

Если нет веских оснований считать, что готовые инструменты не подходят для ваших целей, то и не надо создавать собственную систему тестирования. Существует большое количество готовых решений. О некоторых из них мы расскажем в следующих разделах.

Инструменты полного тестирования

Как мы уже упоминали ранее, существует два типа эталонного тестирования: полное и покомпонентное. Ничего удивительного, что имеются инструменты для тестирования всего приложения, и для тестирования под нагрузкой MySQL и других компонентов по отдельности. Полное тестирование обычно является лучшим способом получить ясное представление о производительности системы. В число существующих инструментов полного тестирования входят:

ab

ab представляет собой широко известный инструмент тестирования производительности сервера HTTP Apache. Он показывает, сколько запросов в секунду способен обслуживать HTTP-сервер. Если вы тестируете веб-приложение, это число демонстрирует, какое количество запросов в секунду может обслужить приложение в целом. Это очень простой инструмент, но полезность его ограничена, поскольку он просто обращается к одному адресу URL настолько быстро, насколько это возможно. Дополнительную информацию об утилите *ab* вы можете найти на странице *http://httpd.apache.org/docs/2.0/programs/ab.html*.

http_load

Этот инструмент концептуально похож на *ab*. Он также предназначен для создания нагрузки на веб-сервер, но при этом является более гибким. Вы можете создать входной файл, включающий много разных адресов URL, а *http_load* будет выбирать их случайным образом. Вы также можете настроить параметры таким образом, что запросы станут отправляться с заданным интервалом, а не с максимально возможной скоростью. Подробности см. на странице *http://www.acme.com/software/http_load/*.

JMeter

JMeter представляет собой приложение на языке Java, которое может загружать другое приложение и измерять его производительность. Эта программа была разработана для тестирования веб-приложений, но ее можно также использовать при тестировании FTP-серверов и для отправки запросов к базе данных через интерфейс JDBC.

Программа JMeter значительно сложнее, чем *ab* и *http_load*. Например, с ее помощью можно более гибко эмулировать поведение реальных пользователей, управляя таким параметром, как время нарастания нагрузки. Эта программа имеет графический интерфейс с интегрированными средствами построения графиков, а также позволяет сохранять результаты и воспроизводить их в автономном режиме. Подробности см. на странице *http://jakarta.apache.org/jmeter/*.

Инструменты покомпонентного тестирования

Существует несколько полезных инструментов для тестирования производительности MySQL и операционной системы, на которой она работает. Примеры тестов с использованием этих инструментов мы приведем в настоящем разделе:

mysqlslap

mysqlslap (*http://dev.mysql.com/doc/refman/5.1/en/mysqlslap.html*) эмулирует нагрузку на сервер и выдает данные хронометража. Эта программа является частью дистрибутива MySQL 5.1, но ее можно использовать и с более ранними версиями, начиная с 4.1. Данный инструмент позволяет настроить количество одновременных соединений и передать программе либо команду SQL в командной строке, либо файл с командами SQL, которые нужно выполнить. Если вы не зададите режим тестирования вручную, программа сама исследует схему базы данных и автоматически сгенерирует команды SELECT.

sysbench

sysbench (*http://sysbench.sourceforge.net*) представляет собой многопотоковый инструмент эталонного тестирования операционной системы. Его задача – дать представление о производительности ОС в терминах факторов, существенных для работы сервера базы данных. Например, вы можете измерить производительность операций файлового ввода/вывода, планировщика операционной системы, выделения памяти и передачи данных, потоков POSIX и самого сервера базы данных. Инструмент *sysbench* поддерживает сценарии на языке Lua (*http://www.lua.org*), что придает ему большую гибкость при тестировании.

Database Test Suite

Инструмент Database Test Suite, разработанный компанией The Open-Source Development Labs (OSDL) и размещенный на сайте SourceForge по адресу *http://sourceforge.net/projects/osdldbt/*, представляет собой

комплект программ для тестирования производительности, аналогичный стандартным промышленным тестам, например опубликованным Советом по производительности обработки транзакций (Transaction Processing Performance Council – TPC). В частности, инструмент *dbt2* представляет собой бесплатную (но несертифицированную) реализацию теста TPC-C OLTP. Он поддерживает подсистемы хранения InnoDB и Falcon. На момент написания этих строк состояние дел с другими транзакционными подсистемами хранения данных MySQL неизвестно.

MySQL Benchmark Suite (sql-bench)

С сервером MySQL распространяется собственный набор инструментов эталонного тестирования, который вы можете использовать для исследования нескольких различных СУБД. Он однопоточный и измеряет скорость выполнения запросов сервером. Результаты показывают, какие типы операций сервер выполняет наилучшим образом.

Главным преимуществом этого набора тестов является то, что он содержит множество готовых программных решений, с его помощью легко сравнивать различные подсистемы хранения или конфигурации. Как высокоуровневый инструмент эталонного тестирования он полезен для сравнения общей производительности двух серверов. Также вы можете запускать подмножество тестов (например, тестировать только производительность команды `UPDATE`). В основном тесты нагружают процессоры, но есть короткие периоды, в течение которых выполняется большой объем операций дискового ввода/вывода.

Среди главных недостатков этого инструмента можно упомянуть то, что он работает в однопользовательском режиме, задействует очень маленький набор исходных значений, не допускает тестирования на данных, характерных именно для ваших условий, а результаты могут отличаться от запуска к запуску. Поскольку он однопоточный и полностью последовательный, с его помощью невозможно оценить выигрыш от наличия нескольких процессоров, но вместе с тем он вполне пригоден для сравнения однопроцессорных серверов.

На машине, где работает тестируемая СУБД, должны быть установлены интерпретатор языка Perl и драйверы DBD. Документацию можно найти по адресу *http://dev.mysql.com/doc/en/mysqlbenchmarks. html/*.

Super Smack

Инструмент Super Smack (*http://vegan.net/tony/supersmack/*) предназначен для эталонного тестирования, тестирования под нагрузкой и создания нагрузки в применении к СУБД MySQL и PostgreSQL. Это мощный, но достаточно сложный инструмент, который позволяет эмулировать несколько пользователей, загружать тестовые данные в базу и заполнять таблицы сгенерированными случайными данными. Тесты содержатся в «smack»-файлах и описываются на простом языке определения клиентов, таблиц, запросов и т. п.

Функция MySQL BENCHMARK()

В MySQL имеется удобная функция BENCHMARK(), которую можно использовать при тестировании скорости выполнения определенных типов операций. Для этого нужно указать количество прогонов и подлежащее выполнению выражение. В качестве последнего может выступать любое скалярное выражение, например скалярный подзапрос или функция. Это удобно для тестирования относительных скоростей некоторых операций, например для выяснения того, какая функция работает быстрее: MD5() или SHA1():

```
mysql> SET @input := 'hello world';
mysql> SELECT BENCHMARK(1000000, MD5(@input));
+---------------------------------+
| BENCHMARK(1000000, MD5(@input)) |
+---------------------------------+
|                               0 |
+---------------------------------+
1 row in set (2.78 sec)
mysql> SELECT BENCHMARK(1000000, SHA1(@input));
+---------------------------------+
| BENCHMARK(1000000, SHA1(@input)) |
+---------------------------------+
|                                0 |
+---------------------------------+
1 row in set (3.50 sec)
```

Возвращаемым значением всегда является нуль. Время исполнения запроса сообщает клиентское приложение. В данном случае похоже, что MD5() быстрее. Однако корректно использовать функцию BENCHMARK() непросто, если вы не знаете, что она делает в действительности. Она просто измеряет, насколько быстро сервер может выполнить выражение, но не сообщает ничего о накладных расходах на синтаксический анализ и оптимизацию. А если выражение не включает в себя пользовательскую переменную, как в нашем примере, то второе и последующие выполнения сервером данного выражения могут оказаться обращением к кэшу[1].

Несмотря на удобство функции BENCHMARK(), мы никогда не используем ее для реального тестирования производительности. Слишком трудно определить, что же она в действительности измеряет, к тому же она слишком узко сфокусирована на малой части общего процесса исполнения.

---

[1] Один из авторов сделал такую ошибку и обнаружил, что 10 000 выполнений определенного выражения заняли почти столько же времени, что и одно выполнение. Это было обращение к кэшу. В общем случае поведение такого рода заставляет предполагать либо обращение к кэшу, либо ошибку.

Примеры эталонного тестирования

В этом разделе мы приведем некоторые примеры реальных эталонных тестов с помощью вышеупомянутых инструментов. Мы не можем дать исчерпывающее описание каждого инструмента, но эти примеры помогут вам решить, какие тесты могут быть полезны для ваших целей, и начать их использование.

http_load

Начнем с простого примера, демонстрирующего использование *http_load* со следующими адресами URL, которые мы сохранили в файле *urls.txt*:

```
http://www.mysqlperformanceblog.com/
http://www.mysqlperformanceblog.com/page/2/
http://www.mysqlperformanceblog.com/mysql-patches/
http://www.mysqlperformanceblog.com/mysql-performance-presentations/
http://www.mysqlperformanceblog.com/2006/09/06/slow-query-log-analyzes-tools/
```

Простейшим способом практического применения инструмента *http_load* является простое извлечение страниц по указанным адресам (URL) в цикле. Программа извлекает их с максимально возможной скоростью:

```
$ http_load -parallel 1 -seconds 10 urls.txt
19 fetches, 1 max parallel, 837929 bytes, in 10.0003 seconds
44101.5 mean bytes/connection
1.89995 fetches/sec, 83790.7 bytes/sec
msecs/connect: 41.6647 mean, 56.156 max, 38.21 min
msecs/first-response: 320.207 mean, 508.958 max, 179.308 min
HTTP response codes:
  code 200 -- 19
```

Результаты вполне ясны – они просто показывают статистическую информацию о запросах. Несколько более сложный сценарий – извлечение адресов URL с максимально возможной скоростью в цикле, но с эмуляцией пяти одновременно работающих пользователей:

```
$ http_load -parallel 5 -seconds 10 urls.txt
94 fetches, 5 max parallel, 4.75565e+06 bytes, in 10.0005 seconds
50592 mean bytes/connection
9.39953 fetches/sec, 475541 bytes/sec
msecs/connect: 65.1983 mean, 169.991 max, 38.189 min
msecs/first-response: 245.014 mean, 993.059 max, 99.646 min
HTTP response codes:
  code 200 -- 94
```

Вместо максимально быстрого извлечения мы можем эмулировать нагрузку для прогнозируемой частоты запросов (например, пять раз в секунду):

```
$ http_load -rate 5 -seconds 10 urls.txt
48 fetches, 4 max parallel, 2.50104e+06 bytes, in 10 seconds
52105 mean bytes/connection
```

```
4.8 fetches/sec, 250104 bytes/sec
msecs/connect: 42.5931 mean, 60.462 max, 38.117 min
msecs/first-response: 246.811 mean, 546.203 max, 108.363 min
HTTP response codes:
code 200 -- 48
```

Наконец, эмулируем еще большую нагрузку с частотой поступления запросов 20 раз в секунду. Обратите внимание, как увеличивается время соединения и отклика с возрастанием нагрузки:

```
$ http_load -rate 20 -seconds 10 urls.txt
111 fetches, 89 max parallel, 5.91142e+06 bytes, in 10.0001 seconds
53256.1 mean bytes/connection
11.0998 fetches/sec, 591134 bytes/sec
msecs/connect: 100.384 mean, 211.885 max, 38.214 min
msecs/first-response: 2163.51 mean, 7862.77 max, 933.708 min
HTTP response codes:
  code 200 -- 111
```

sysbench

С помощью инструмента *sysbench* можно запускать различные эталонные тесты. Он был разработан для тестирования не только производительности базы данных, но и того, насколько хорошо работает система как сервер баз данных. Мы начнем с некоторых тестов, которые не являются специфичными для MySQL и измеряют производительность подсистем, определяющих общие ограничения системы. Затем мы покажем, как измерять производительность СУБД.

Эталонное тестирование процессора с помощью инструмента sysbench

Наиболее очевидным тестом подсистемы является эталонный тест процессора, в котором используются 64-разрядные целые числа для вычисления простых чисел до указанного максимума. Мы запустили этот тест на двух серверах, каждый под управлением GNU/Linux, и сравнили результаты. Вот характеристики оборудования первого сервера данных:

```
[server1 ~]$ cat /proc/cpuinfo
...
model name : AMD Opteron(tm) Processor 246
stepping   : 1
cpu MHz    : 1992.857
cache size : 1024 KB
```

И вот что показал тест производительности:

```
[server1 ~]$ sysbench --test=cpu --cpu-max-prime=20000 run
sysbench v0.4.8: multi-threaded system evaluation benchmark
...
Test execution summary:
  total time: 121.7404s
```

У второго сервера другой процессор:

```
[server2 ~]$ cat /proc/cpuinfo
...
model name : Intel(R) Xeon(R) CPU 5130 @ 2.00GHz
stepping   : 6
cpu MHz    : 1995.005
```

Вот результат его тестирования производительности:

```
[server1 ~]$ sysbench --test=cpu --cpu-max-prime=20000 run
sysbench v0.4.8: multi-threaded system evaluation benchmark
...
Test execution summary:
  total time: 61.8596s
```

Результат просто показывает общее время, требуемое для вычисления простых чисел, что очень легко сравнить. В данном случае второй сервер выполнил тест приблизительно вдвое быстрее первого.

Эталонное тестирование файлового ввода/вывода с помощью инструмента sysbench

Эталонный тест *fileio* измеряет параметры работы операционной системы при различных нагрузках ввода/вывода. Он очень полезен для сравнения жестких дисков, RAID-контроллеров и режимов RAID, а также для настройки подсистемы ввода/вывода.

Первым шагом при выполнении этого теста является подготовка нескольких файлов. Вам нужно сгенерировать значительно больше данных, чем помещается в память. Если данные загрузятся в память, операционная система будет кэшировать большую их часть, а результаты не совсем точно отразят нагрузку на подсистему ввода/вывода. Начнем с создания набора данных:

```
$ sysbench --test=fileio --file-total-size=150G prepare
```

Вторым шагом является запуск теста. Для тестирования производительности при различных типах нагрузки на систему ввода/вывода используются следующие параметры:

`seqwr`
: Последовательная запись

`seqrewr`
: Последовательная перезапись

`seqrd`
: Последовательное чтение

`rndrd`
: Произвольная запись

`rndwr`

Произвольное чтение

`rndrw`

Произвольное чтение в сочетании с произвольной записью

Следующая команда запускает тест ввода/вывода с произвольным чтением/записью:

```
$ sysbench --test=fileio --file-total-size=150G --file-test-mode=rndrw
--init-rng=on --max-time=300 --max-requests=0 run
```

Вот результаты:

```
sysbench v0.4.8: multi-threaded system evaluation benchmark

Running the test with following options:
Number of threads: 1
Initializing random number generator from timer.

Extra file open flags: 0
128 files, 1.1719Gb each
150Gb total file size
Block size 16Kb
Number of random requests for random IO: 10000
Read/Write ratio for combined random IO test: 1.50
Periodic FSYNC enabled, calling fsync( ) each 100 requests.
Calling fsync( ) at the end of test, Enabled.
Using synchronous ввода-вывода mode
Doing random r/w test
Threads started!
Time limit exceeded, exiting...
Done.

Operations performed: 40260 Read, 26840 Write, 85785 Other = 152885 Total
Read 629.06Mb Written 419.38Mb Total transferred 1.0239Gb (3.4948Mb/sec)
  223.67 Requests/sec executed

Test execution summary:
    total time:                          300.0004s
    total number of events:              67100
    total time taken by event execution: 254.4601
    per-request statistics:
        min:                             0.0000s
        avg:                             0.0038s
        max:                             0.5628s
        approx. 95 percentile:           0.0099s

Threads fairness:
    events (avg/stddev):          67100.0000/0.00
    execution time (avg/stddev):  254.4601/0.00
```

Результат работы команды содержит большое количество информации. Наиболее интересны для настройки подсистемы ввода/вывода количество запросов в секунду и общая пропускная способность. В данном случае мы получили 223,67 запросов в секунду и 3,4948 Мбайт в секунду соответственно. Эти значения дают хорошее представление о производительности диска.

Закончив тест, можете удалить файлы, которые программа *sysbench* создала для тестирования:

```
$ sysbench --test=fileio --file-total-size=150G cleanup
```

Эталонное тестирование OLTP-системы с помощью инструмента sysbench

Эталонное тестирование системы оперативной обработки транзакций (OLTP) эмулирует рабочую нагрузку, характерную для обработки транзакций. Мы приведем пример такого тестирования для таблицы, содержащей миллион строк. Первым шагом является подготовка самой таблицы, на которой будет выполняться тестирование:

```
$ sysbench --test=oltp --oltp-table-size=1000000
  --mysql-db=test --mysql-user=root prepare
sysbench v0.4.8: multi-threaded system evaluation benchmark

No DB drivers specified, using mysql
Creating table 'sbtest'...
Creating 1000000 records in table 'sbtest'...
```

Это все, что нужно сделать для подготовки тестовых данных. Затем мы запускаем тест в режиме чтения на 60 секунд с 8 одновременно работающими потоками:

```
$ sysbench --test=oltp --oltp-table-size=1000000 --mysql-db=test
  --mysql-user=root --max-time=60 --oltp-read-only=on --max-requests=0
  --num-threads=8 run
sysbench v0.4.8: multi-threaded system evaluation benchmark

No DB drivers specified, using mysql
WARNING: Preparing of "BEGIN" is unsupported, using emulation
(last message repeated 7 times)
Running the test with following options:
Number of threads: 8

Doing OLTP test.
Running mixed OLTP test
Doing read-only test
Using Special distribution (12 iterations, 1 pct of values are returned in
75 pct
cases)
Using "BEGIN" for starting transactions
Using auto_inc on the id column
```

```
Threads started!
Time limit exceeded, exiting...
(last message repeated 7 times)
Done.

OLTP test statistics:
    queries performed:
        read:                   179606
        write:                  0
        other:                  25658
        total:                  205264
    transactions:               12829  (213.07 per sec.)
    deadlocks:                  0      (0.00 per sec.)
    read/write requests:        179606 (2982.92 per sec.)
    other operations:           25658  (426.13 per sec.)

Test execution summary:
    total time:                 60.2114s
    total number of events:     12829
    total time taken by event execution: 480.2086
    per-request statistics:
        min:                    0.0030s
        avg:                    0.0374s
        max:                    1.9106s
        approx. 95 percentile:  0.1163s

Threads fairness:
    events (avg/stddev):         1603.6250/70.66
    execution time (avg/stddev): 60.0261/0.06
```

Как и раньше, в полученных нами результатах содержится очень много информации. Наибольший интерес представляют следующие данные:

- Счетчик транзакций
- Количество транзакций в секунду
- Статистика по запросам (минимальное, среднее, максимальное время и 95-й процентиль)
- Статистика по равномерности загрузки потоков, которая показывает, насколько справедливо распределялась между ними эмулированная нагрузка.

Другие возможности инструмента sysbench

Инструмент *sysbench* может запускать несколько других эталонных тестов системы, которые непосредственно не измеряют производительность СУБД. Среди них:

`memory`

Выполняет последовательные операции чтения/записи в памяти.

`threads`

Тестирует производительность планировщика потоков. Особенно полезно тестировать поведение планировщика при высоких нагрузках.

`mutex`

Измеряет производительность работы мьютексов, эмулируя ситуацию, когда все потоки большую часть времени работают параллельно, захватывая мьютекс только на короткое время (мьютекс – это структура данных, которая гарантирует взаимно исключающий доступ к некоторому ресурсу, предотвращая возникновение проблем, связанных с одновременным доступом).

`seqwr`

Измеряет производительность последовательной записи (один из типов нагрузки в режиме тестирования fileio). Это очень важно для тестирования практических пределов производительности системы. Тест показывает, насколько хорошо работает кэш RAID-контроллера, и предупреждает, если результаты оказываются необычными. Например, если отсутствует кэш записи с резервным батарейным питанием, а диск обрабатывает 3 000 запросов в секунду, то что-то идет не так и вашим данным определенно угрожает опасность.

Кроме параметра, задающего режим тестирования *(--test)*, инструмент *sysbench* принимает и некоторые другие общие параметры, например *--num-threads*, *--max-requests* и *--maxtime*. Подробное описание этих параметров содержится в соответствующей технической документации.

Инструмент dbt2 из комплекта Database Test Suite

Инструмент *dbt2* из комплекта Database Test Suite представляет собой бесплатную реализацию теста TPC-C. TPC-C – это спецификация, опубликованная организацией TPC и описывающая эмуляцию нагрузки, характерной для сложной оперативной обработки транзакций. Этот инструмент сообщает результаты в транзакциях в минуту (tpmC), а также стоимость каждой транзакции (Price/tpmC). Результаты сильно зависят от оборудования, поэтому опубликованные результаты TPC-C содержат подробные спецификации серверов, использованных при проведении теста.

Тест *dbt2* не является настоящим тестом TPC-C. Он не сертифицирован организацией TPC, и его результаты нельзя непосредственно сравнивать с результатами TPC-C.

Давайте посмотрим, как настраивать и запускать тест *dbt2*. Мы использовали версию *dbt2* 0.37, наиболее свежую из подходящих для MySQL (более поздние версии содержат особенности, которые MySQL поддерживает не полностью). Мы выполнили следующие шаги:

1. Подготовка данных.

   Следующая команда создает данные по 10 складам в указанном каталоге. В целом информация по 10 складам занимает около 700 Мбайт на диске. Требуемое место пропорционально указанному количеству складов, так что можете изменить параметр *-w*, чтобы создать набор данных нужного вам размера.

```
# src/datagen -w 10 -d /mnt/data/dbt2-w10
warehouses = 10
districts = 10
customers = 3000
items = 100000
orders = 3000
stock = 100000
new_orders = 900

Output directory of data files: /mnt/data/dbt2-w10

Generating data files for 10 warehouse(s)...
Generating item table data...
Finished item table data...
Generating warehouse table data...
Finished warehouse table data...
Generating stock table data...
```

2. Загрузка данных в базу MySQL.

   Следующая команда создает базу с названием `dbt2w10` и заполняет ее данными, которые мы сгенерировали на предыдущем шаге (флаг *-d* задает имя базы, а *-f* – каталог со сгенерированными данными):

```
# scripts/mysql/mysql_load_db.sh -d dbt2w10 -f /mnt/data/dbt2-w10 -s /var/
lib/mysql/mysql.sock
```

3. Запуск теста производительности.

   Последний шаг заключается в выполнении следующей команды из каталога *scripts*:

```
# ./run_mysql.sh -c 10 -w 10 -t 300 -n dbt2w10 -u root -o
 /var/lib/mysql/mysql.sock -e
************************************************************************
*                   DBT2 test for MySQL started                        *
*                                                                      *
*            Results can be found in output/9 directory                *
************************************************************************
*                                                                      *
* Test consists of 4 stages:                                           *
*                                                                      *
* 1. Start of client to create pool of databases connections           *
* 2. Start of driver to emulate terminals and transactions generation  *
* 3. Test                                                              *
* 4. Processing of results                                             *
```

```
*                                                                              *
********************************************************************************

 DATABASE NAME:             dbt2w10
 DATABASE USER:             root
 DATABASE SOCKET:           /var/lib/mysql/mysql.sock
 DATABASE CONNECTIONS:      10
 TERMINAL THREADS:          100
 SCALE FACTOR(WARHOUSES):   10
 TERMINALS PER WAREHOUSE:   10
 DURATION OF TEST(in sec): 300
 SLEEPY in (msec)           300
 ZERO DELAYS MODE:          1

 Stage 1. Starting up client...
 Delay for each thread - 300 msec. Will sleep for 4 sec
 to start 10 database connections
 CLIENT_PID = 12962

 Stage 2. Starting up driver...
 Delay for each thread - 300 msec. Will sleep for 34 sec
 to start 100 terminal threads
 All threads has spawned successfuly.

 Stage 3. Starting of the test. Duration of the test 300 sec

 Stage 4. Processing of results...
 Shutdown clients. Send TERM signal to 12962.
  Response Time (s)
  Transaction        %   Average :  90th %   Total   Rollbacks      %
------------     ----- -----------------    ------   ---------  -----
     Delivery     3.53     2.224 : 3.059      1603           0   0.00
    New Order    41.24     0.659 : 1.175     18742         172   0.92
 Order Status     3.86     0.684 : 1.228      1756           0   0.00
      Payment    39.23     0.644 : 1.161     17827           0   0.00
  Stock Level     3.59     0.652 : 1.147      1630           0   0.00

 3396.95 new-order transactions per minute (NOTPM)
 5.5 minute duration
 0 total unknown errors
 31 second(s) ramping up
```

Наиболее важная строка выводится практически в конце распечатки:

```
3396.95 new-order transactions per minute (NOTPM)
```

Она показывает, сколько транзакций может обрабатывать система за одну минуту; чем это значение больше, тем лучше. (Термин «new-order» – это не специальное определение для типа транзакции, он просто обозначает, что тест эмулирует добавление заказа в воображаемом интернет-магазине.)

Для создания различных тестов вы можете менять некоторые параметры:

-c Количество соединений с базой данных. Изменением этого параметра вы эмулируете различные уровни конкуренции и видите, как система масштабируется.

-e Данный параметр активизирует режим с нулевыми задержками между запросами. Это позволяет провести нагрузочное тестирование базы данных, но результат может оказаться нереалистичным, так как живым пользователям нужно некоторое время на размышление перед отправкой очередного запроса.

-t Общая продолжительность теста. Выбирайте этот параметр тщательно, иначе результаты будут бессмысленными. Слишком малое время для тестирования производительности ввода/вывода даст неправильные результаты, поскольку у системы будет недостаточно возможностей для «прогрева» кэша и перехода в нормальный режим работы. С другой стороны, если вы хотите протестировать систему в режиме загрузки процессора, то не стоит устанавливать это значение слишком большим, поскольку набор данных может заметно вырасти, и нагрузка ляжет в основном на подсистему ввода/вывода.

Результаты теста могут дать информацию не только о производительности. Например, если вы увидите слишком много откатов транзакций, значит, что-то в вашей системе работает неправильно.

Комплект инструментов MySQL Benchmark Suite

Комплект инструментов MySQL Benchmark Suite состоит из набора эталонных тестов производительности, написанных на языке Perl, так что для запуска этих тестов вам потребуется установить Perl. Тесты находятся в подкаталоге *sql-bench/* инсталляционного каталога MySQL. На системах Debian GNU/Linux, например, они хранятся в каталоге */usr/share/mysql/sql-bench/*.

Прежде чем приступать к работе, прочитайте файл *README*, в котором объясняется, как использовать этот комплект инструментов, и описываются аргументы командной строки. Для запуска всех тестов используются примерно такие команды:

```
$ cd /usr/share/mysql/sql-bench/
sql-bench$ ./run-all-tests --server=mysql --user=root --log --fast
Test finished. You can find the result in:
output/RUN-mysql_fast-Linux_2.4.18_686_smp_i686
```

Продолжительность работы тестов довольно велика – порой она занимает больше часа, в зависимости от оборудования и конфигурации. Если вы укажете параметр командной строки *--log*, то сможете отслеживать состояние теста во время его выполнения. Каждый тест записывает свои результаты в подкаталог под названием *output*. В свою очередь, каждый файл состоит из последовательности временных меток, соответствую-

щих операциям в тесте. Вот немного переформатированный для удобства печати образец:

```
sql-bench$ tail -5 output/select-mysql_fast-Linux_2.4.18_686_smp_i686
Time for count_distinct_group_on_key (1000:6000):
  34 wallclock secs ( 0.20 usr 0.08 sys + 0.00 cusr 0.00 csys = 0.28 CPU)
Time for count_distinct_group_on_key_parts (1000:100000):
  34 wallclock secs ( 0.57 usr 0.27 sys + 0.00 cusr 0.00 csys = 0.84 CPU)
Time for count_distinct_group (1000:100000):
  34 wallclock secs ( 0.59 usr 0.20 sys + 0.00 cusr 0.00 csys = 0.79 CPU)
Time for count_distinct_big (100:1000000):
  8 wallclock secs ( 4.22 usr 2.20 sys + 0.00 cusr 0.00 csys = 6.42 CPU)
Total time:
  868 wallclock secs (33.24 usr 9.55 sys + 0.00 cusr 0.00 csys = 42.79 CPU)
```

Например, выполнение теста `count_distinct_group_on_key (1000:6000)` заняло 34 секунды. Это общее время, в течение которого длилось тестирование. Преобразование других значений (`usr, sys, cursr, csys`), потребовавшее в общей сложности 0,28 секунды, составляет накладные расходы. Данная величина показывает, сколько времени клиентский код реально работал, а не просто ожидал ответа от сервера MySQL. Таким образом, интересующая нас цифра, показывающая временные затраты на неконтролируемую клиентом обработку, составила 33,72 секунды.

Вместо всего комплекта можно запускать тесты по отдельности. Например, вы можете выполнить только тестирование операций вставки. Это даст более подробную информацию, чем в сводном отчете, который создается при запуске полного набора тестов:

```
sql-bench$ ./test-insert
Testing server 'MySQL 4.0.13 log' at 2003-05-18 11:02:39

Testing the speed of inserting data into 1 table and do some selects on it.
The tests are done with a table that has 100000 rows.

Generating random keys
Creating tables
Inserting 100000 rows in order
Inserting 100000 rows in reverse order
Inserting 100000 rows in random order
Time for insert (300000):
  42 wallclock secs ( 7.91 usr 5.03 sys + 0.00 cusr 0.00 csys = 12.94 CPU)
Testing insert of duplicates
Time for insert_duplicates (100000):
  16 wallclock secs ( 2.28 usr 1.89 sys + 0.00 cusr 0.00 csys = 4.17 CPU)
```

Профилирование

Профилирование показывает, какую долю вносит каждая часть системы в общую стоимость получения результата. Простейшей метрикой стоимости является время, но профилирование может также измерять

количество вызовов функций, операций ввода/вывода, запросов к базе данных и т. д. Важно понять, почему система работает именно так, а не иначе.

Профилирование приложения

Так же как и в случае эталонного тестирования, вы можете профилировать на уровне приложения или отдельного компонента, например сервера MySQL. Профилирование на уровне приложения обычно помогает понять, как оптимизировать его наилучшим образом, и дает более точные результаты, поскольку они включают в себя работу, выполненную всей системой. Например, если вы заинтересованы в оптимизации запросов приложения к MySQL, у вас может возникнуть искушение просто запустить запросы и проанализировать их. Однако если вы поступите таким образом, то упустите много важной информации о запросах, например не получите сведения о той работе, которую выполняет приложение для считывания результатов в память и их обработки[1].

Поскольку веб-приложения являются очень распространенным применением MySQL, мы будем использовать в нашем примере веб-сайт, написанный на языке PHP. Обычно следует профилировать приложение глобально, чтобы увидеть, как загружена система, но вы, вероятно, захотите изолировать некоторые интересующие вас подсистемы, например функцию поиска. Любая ресурсоемкая подсистема является хорошим кандидатом на профилирование в изоляции.

Когда нам нужно оптимизировать использование MySQL веб-сайтом на PHP, мы предпочитаем собирать статистику на уровне объектов (или модулей) в коде PHP. Наша цель – измерить, какая часть времени отклика страницы приходится на операции с базой данных. Доступ к базе данных часто, но не всегда, является узким местом приложения. Узкие места могут быть также вызваны одной из следующих причин:

- Обращения к внешним ресурсам, например веб-сервисам или поисковым системам
- Операции, которые требуют обработки больших объемов данных в приложении, например синтаксический анализ больших файлов XML
- Дорогостоящие операции в циклах, например злоупотребление регулярными выражениями
- Плохо оптимизированные алгоритмы, например наивные (naïve) алгоритмы поиска в списках

Перед тем как начинать анализ запросов к MySQL, вы должны понять реальный источник ваших проблем с производительностью. Профили-

[1] Узкое место можно обнаружить, разобравшись в статистике системы. Если веб-серверы простаивают, а загрузка процессора на сервере MySQL составляет 100%, то можно вообще обойтись без профилирования.

рование приложения поможет найти узкие места, и это важный шаг в мониторинге и увеличении общего быстродействия системы.

Как и что измерять

Время является подходящей метрикой при профилировании большинства приложений, поскольку конечного пользователя больше всего интересует именно время работы. В веб-приложениях мы обычно предусматриваем режим отладки, в котором на каждой странице отображаются выполняемые запросы, а также момент их выполнения и количество возвращенных строк. Далее мы можем запустить команду `EXPLAIN` для медленных запросов (более подробная информация об этой команде приведена в последующих главах). Для более глубокого анализа мы объединяем эти данные с показателями сервера MySQL.

Мы рекомендуем включать код профилирования в *каждый* новый проект. Внедрить код профилирования в существующее приложение может оказаться трудной задачей, но в новые приложения включать его легко. Многие библиотеки содержат специальные средства, упрощающие дело. Например, интерфейс JDBC в языке Java и библиотека *mysqli* в PHP имеют встроенные функции для профилирования доступа к базам данных.

Код профилирования также очень полезен для отслеживания странностей, которые появляются только в рабочей версии приложения и которые невозможно воспроизвести в условиях разработки.

Ваш код профилирования должен собирать и регистрировать, по меньшей мере, следующие сведения:

- Общее время выполнения (в веб-приложениях это полное время генерирования страниц)
- Каждый выполненный запрос и время его исполнения
- Каждый факт открытия соединения с сервером MySQL
- Каждое обращение к внешнему ресурсу, например к веб-сервисам, службе *memcached* и внешним вызываемым сценариям
- Потенциально дорогостоящие вызовы функций, например синтаксический анализ XML
- Процессорное время, потраченное в режиме пользователя и ядра

Эта информация облегчает мониторинг производительности. Она дает понимание, которое вы не сможете получить иными способами, таких аспектов производительности, как:

- Общие проблемы производительности
- Спонтанное увеличение времени отклика
- Узкие места системы, которые могут находиться вне MySQL
- Время на выполнение запросов «невидимых» пользователей, например роботов поисковых систем

Не замедлит ли профилирование работу ваших серверов?

Да. Профилирование и мониторинг увеличивают накладные расходы. Следует ответить на два вопроса: каковы эти накладные расходы и стоит ли овчинка выделки.

Многие люди, которые занимаются проектированием и разработкой высокопроизводительных приложений, полагают, что нужно измерять все, что возможно, и просто принимать стоимость измерений как часть работы приложения. Даже если вы с этим не согласны, стоит хотя бы предусмотреть «облегченное» профилирование, которое может работать постоянно. Мало радости обнаруживать узкие места постфактум просто потому, что вы не встроили в систему средства каждодневного отслеживания изменений производительности. А если вы сталкиваетесь с проблемой, то исторические данные приобретают огромную ценность. Можно также использовать данные профилирования для планирования приобретения оборудования, выделения ресурсов и прогнозирования нагрузки в пиковые периоды или сезоны.

Что мы подразумеваем под «облегченным» профилированием? Хронометраж всех запросов SQL и общего времени исполнения сценария, несомненно, обходится дешево. Кроме того, вам не нужно делать это для каждого просмотра страницы. Если у вас приличный трафик, можно просто профилировать случайную выборку, включая профилирование в конфигурационном файле приложения:

```
<?php
$profiling_enabled = rand(0, 100) > 99;
?>
```

Профилирование только одного процента всех просмотров вашей страницы поможет вам находить самые серьезные проблемы.

При запуске тестов производительности позаботьтесь о подсчете стоимости журналирования, профилирования и измерения, поскольку это может исказить результаты ваших тестов.

Пример профилирования приложения PHP

Чтобы получить представление о том, насколько легким и ненавязчивым может быть профилирование веб-приложения PHP, давайте рассмотрим некоторые примеры кода. Первый пример показывает, что нужно добавить в приложение, как протоколировать запросы и другие данные профилирования в таблицах MySQL и как анализировать результаты.

Чтобы уменьшить влияние протоколирования, мы будем размещать всю регистрируемую информацию в памяти, а затем записывать ее одной строкой после окончания выполнения страницы. Это лучше, чем регистрация каждого запроса в отдельности, поскольку подобный подход удвоил бы количество отправляемых на сервер MySQL запросов. Протоколирование каждого фрагмента профилируемых данных может затруднить поиск узких мест, так как на уровне приложения редко удается собрать достаточно детальные данные.

Мы начнем с кода, который необходим для перехвата профилируемой информации. Вот упрощенный пример класса протоколирования PHP 5 *class.Timer.php*, в котором используются такие встроенные функции, как `getrusage()`, для определения потребления ресурсов сценарием:

```
<?php
/*
* Класс Timer, реализация хронометража в PHP
*/

class Timer {
  private $aTIMES = array( );

  function startTime($point)
  {
    $dat = getrusage( );

    $this->aTIMES[$point][‘start’] = microtime(TRUE);
    $this->aTIMES[$point][‘start_utime’] =
        $dat[“ru_utime.tv_sec”]*1e6+$dat[“ru_utime.tv_usec”];
    $this->aTIMES[$point][,start_stime‘] =
        $dat[“ru_stime.tv_sec”]*1e6+$dat[“ru_stime.tv_usec”];
  }

  function stopTime($point, $comment=‘‘)
  {
    $dat = getrusage( );
    $this->aTIMES[$point][,end‘] = microtime(TRUE);
    $this->aTIMES[$point][,end_utime‘] =
        $dat[“ru_utime.tv_sec”] * 1e6 + $dat[“ru_utime.tv_usec”];
    $this->aTIMES[$point][,end_stime‘] =
        $dat[“ru_stime.tv_sec”] * 1e6 + $dat[“ru_stime.tv_usec”];

    $this->aTIMES[$point][,comment‘] .= $comment;

    $this->aTIMES[$point][,sum‘] +=
        $this->aTIMES[$point][,end‘] - $this->aTIMES[$point][,start‘];
    $this->aTIMES[$point][,sum_utime‘] +=
        ($this->aTIMES[$point][,end_utime‘] -
            $this->aTIMES[$point][,start_utime‘]) / 1e6;
    $this->aTIMES[$point][,sum_stime‘] +=
        ($this->aTIMES[$point][,end_stime‘] -
```

```
38              $this->aTIMES[$point][,start_stime‘]) / 1e6;
39    }
40
41    function logdata( ) {
42
43      $query_logger = DBQueryLog::getInstance(,DBQueryLog‘);
44      $data[,utime‘] = $this->aTIMES[,Page‘][,sum_utime‘];
45      $data[,wtime‘] = $this->aTIMES[,Page‘][,sum‘];
46      $data[,stime‘] = $this->aTIMES[,Page‘][,sum_stime‘];
47      $data[,mysql_time‘] = $this->aTIMES[,MySQL‘][,sum‘];
48      $data[,mysql_count_queries‘] = $this->aTIMES[,MySQL‘][,cnt‘];
49      $data[,mysql_queries‘] = $this->aTIMES[,MySQL‘][,comment‘];
50      $data[,sphinx_time‘] = $this->aTIMES[,Sphinx‘][,sum‘];
51
52      $query_logger->logProfilingData($data);
53
54    }
55
56    // Эта служебная функция реализует паттерн Singleton
57    function getInstance( ) {
58      static $instance;
59
60      if(!isset($instance)) {
61         $instance = new Timer( );
62      }
63
64      return($instance);
65    }
66 }
67 ?>
```

Использование класса `Timer` в приложении сложности не представляет. Нужно просто окружить обращениями к таймеру потенциально дорогие (или интересующие вас по другой причине) вызовы. Например, вот как это сделать с каждым запросом к MySQL. Новый интерфейс PHP `mysqli` позволяет расширить базовый класс `mysqli` и переобъявить метод `query`:

```
68 <?php
69 class mysqlx extends mysqli {
70   function query($query, $resultmode) {
71     $timer = Timer::getInstance( );
72     $timer->startTime(‘MySQL’);
73     $res = parent::query($query, $resultmode);
74     $timer->stopTime(‘MySQL’, “Query: $query\n”);
75     return $res;
76   }
77 }
78 ?>
```

Эта методика требует очень незначительных изменений кода. Можно просто изменить `mysqli` на `mysqlx` глобально, и тогда приложение начнет

протоколировать все запросы. Этот подход используется для измерения времени доступа к любому внешнему ресурсу, например к системе полнотекстового поиска Sphinx:

```
$timer->startTime('Sphinx');
$this->sphinxres = $this->sphinx_client->Query ( $query, "index" );
$timer->stopTime('Sphinx', "Query: $query\n");
```

Теперь посмотрим, как протоколировать собранные данные. Это как раз тот случай, когда разумно использовать подсистему MyISAM или Archive. Любая из них является хорошим кандидатом для хранения журналов. При добавлении строк в журналы мы используем команду `INSERT DELAYED`, так что вставка будет выполнена в фоновом потоке сервера базы данных. Это означает, что запрос вернет управление мгновенно и не окажет заметного влияния на время отклика. (Даже если не использовать `INSERT DELAYED`, вставки будут выполняться параллельно с другими операциями, если мы не отключим этот режим явно, поэтому посторонние команды `SELECT` не будут блокировать протоколирование). Наконец, мы реализуем схему секционирования по дате, ежедневно создавая новую таблицу для хранения протоколов.

Вот как выглядит команда `CREATE TABLE` для создания нашей таблицы:

```
CREATE TABLE logs.performance_log_template (
    ip                  INT UNSIGNED NOT NULL,
    page                VARCHAR(255) NOT NULL,
    utime               FLOAT NOT NULL,
    wtime               FLOAT NOT NULL,
    mysql_time          FLOAT NOT NULL,
    sphinx_time         FLOAT NOT NULL,
    mysql_count_queries INT UNSIGNED NOT NULL,
    mysql_queries       TEXT NOT NULL,
    stime               FLOAT NOT NULL,
    logged              TIMESTAMP NOT NULL
                        default CURRENT_TIMESTAMP on update CURRENT_
TIMESTAMP,
    user_agent          VARCHAR(255) NOT NULL,
    referer             VARCHAR(255) NOT NULL
) ENGINE=ARCHIVE;
```

На самом деле мы не будем вставлять в эту таблицу какие-либо данные. Это просто шаблон для команд `CREATE TABLE LIKE`, которые мы используем для создания ежедневных таблиц.

Более подробно об этом будет рассказано в главе 3, а пока только заметим, что имеет смысл использовать тип данных минимального размера, достаточного для хранения нужных записей. Чтобы сохранить IP-адреса, мы используем беззнаковое целое. Для хранения адреса страницы и адреса страницы-источника перехода мы используем символьный тип длиной 255 знаков. Длина строки может превышать это значение, но обычно первых 255 символов достаточно для наших целей.

Завершающей частью является протоколирование результатов по окончании выполнения страницы. Вот код PHP, необходимый для записи данных в таблицу:

```
<?php
// Начало выполнения страницы
$timer = Timer::getInstance( );
$timer->startTime('Page');
// ... другой код ...
// Конец выполнения страницы
$timer->stopTime('Page');
$timer->logdata( );
?>
```

В классе `Timer` используется вспомогательный класс `DBQueryLog`, который отвечает за протоколирование в базу данных и ежедневное создание новой таблицы. Вот его код:

```
<?php
/*
* Класс DBQueryLog записывает данные профилирования в базу
*/
class DBQueryLog {

  // конструктор и т. д...

  /*
   * Запись данных, создание таблицы-протокола, если ее нет. Заметьте,
   * что дешевле полагать, что таблица существует и перехватывать ошибку,
   * если это не так, чем проверять ее существование при каждом запросе.
   */
  function logProfilingData($data) {
    $table_name = "logs.performance_log_" . @date("ymd");

    $query = "INSERT DELAYED INTO $table_name (ip, page, utime,
           wtime, stime, mysql_time, sphinx_time, mysql_count_queries,
           mysql_queries, user_agent, referer) VALUES (.. data ..)";

    $res = $this->mysqlx->query($query);
    // Обработка ошибки «таблица не найдена» - создание таблицы для нового дня
    if ((!$res) && ($this->mysqlx->errno == 1146)) { // 1146 - таблица не найдена
       $res = $this->mysqlx->query(
         "CREATE TABLE $table_name LIKE logs.performance_log_template");
       $res = $this->mysqlx->query($query);
     }
  }
}
?>
```

После того как данные собраны, можно анализировать журнал. Прелесть использования MySQL для протоколирования заключается в том, что свойственная SQL гибкость вполне пригодна для анализа, так что

вы легко можете написать запрос с целью получения любого нужного отчета на основе данных журнала. Например, найти семь страниц, время исполнения которых превышает 10 секунд в первый день февраля 2007 года:

```
mysql> SELECT page, wtime, mysql_time
    -> FROM performance_log_070201 WHERE wtime > 10 LIMIT 7;
+-----------------------------------------+---------+------------+
| page                                    | wtime   | mysql_time |
+-----------------------------------------+---------+------------+
| /page1.php                              | 50.9295 |   0.000309 |
| /page1.php                              | 32.0893 |   0.000305 |
| /page1.php                              | 40.4209 |   0.000302 |
| /page3.php                              | 11.5834 |   0.000306 |
| /login.php                              | 28.5507 |    28.5257 |
| /access.php                             | 13.0308 |    13.0064 |
| /page4.php                              | 32.0687 |   0.000333 |
+-----------------------------------------+---------+------------+
```

(Обычно мы выбираем больше данных, но здесь в целях иллюстрации сократили запрос.)

Сравнив `wtime` (полное время исполнения страницы) и время запроса, вы увидите, что запрос MySQL долго обрабатывался только на двух из семи страниц. Поскольку вместе с данными профилирования мы сохраняли сами запросы, их можно извлечь для проверки:

```
mysql> SELECT mysql_queries
    -> FROM performance_log_070201 WHERE mysql_time > 10 LIMIT 1\G
*************************** 1. row ***************************
mysql_queries:
Query: SELECT id, chunk_id FROM domain WHERE domain = 'domain.com'
Time: 0.00022602081298828
Query: SELECT server.id sid, ip, user, password, domain_map.id as chunk_id
FROM server JOIN domain_map ON (server.id = domain_map.master_id) WHERE
domain_map.id = 24
Time: 0.00020599365234375
Query: SELECT id, chunk_id, base_url,title FROM site WHERE id = 13832
Time: 0.00017690658569336
Query: SELECT server.id sid, ip, user, password, site_map.id as chunk_id
FROM server JOIN site_map ON (server.id = site_map.master_id) WHERE site_
map.id = 64
Time: 0.0001990795135498
Query: SELECT from_site_id, url_from, count(*) cnt FROM link24.link_in24
FORCE INDEX (domain_message) WHERE domain_id=435377 AND message_day IN (...)
GROUP BY from_site_id ORDER BY cnt desc LIMIT 10
Time: 6.3193740844727
Query: SELECT revert_domain, domain_id, count(*) cnt FROM art64.link_out64
WHERE from_site_id=13832 AND message_day IN (...) GROUP BY domain_id ORDER
BY cnt desc LIMIT 10
Time: 21.3649559021
```

Таким образом, мы обнаружили два проблематичных запроса со временем выполнения 6,3 и 21,3 секунды, которые нужно оптимизировать.

Протоколирование всех запросов подобным способом обходится дорого, поэтому обычно мы включаем протоколирование либо для части страниц, либо только в режиме отладки.

Как можно определить, есть ли узкое место в той части системы, которая не подвергалась профилированию? Самый простой способ – взглянуть на «потерянное время». Полное время выполнения страницы (`wtime`) представляет собой сумму времени работы в режиме пользователя, в режиме ядра, времени исполнения SQL-команды и другого времени, которое вы можете измерить, плюс «потерянное время», которое вы измерить не можете. Существует некоторое наложение, например процессорное время, необходимое для выполнения кодом PHP обработки SQL-запросов, но оно обычно незначительно. На рис. 2.2 показано, как может распределяться полное время исполнения.

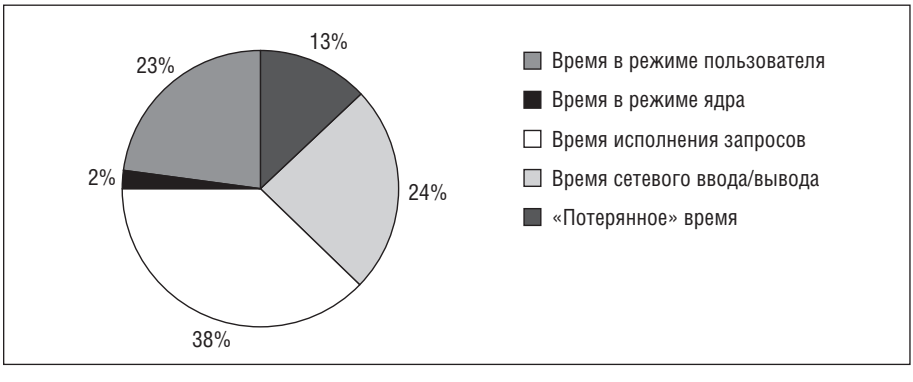


Рис. 2.2. *Потерянное время представляет собой разницу между полным временем выполнения и временем, которое вы можете подсчитать*

В идеале «потерянное время» должно быть как можно меньше. Если после вычитания из `wtime` всего, что вы измерили, все равно останется большое значение, значит, на время работы сценария влияет какой-то неучтенный фактор. Возможно, это время, необходимое для генерации страницы, а может быть, где-то происходит ожидание[1].

Существуют два типа ожиданий: ожидание в очереди к процессору и ожидание ресурсов. Процесс ждет готовности к выполнению, но все процессоры заняты. Обычно невозможно вычислить, сколько времени процесс проводит в очереди к процессору, но это, как правило, не проблема. Более вероятно, что вы обратились к какому-то внешнему ресурсу, забыв поставить профилирование.

[1] Предполагается, что веб-сервер буферизует результат, так что вы не измеряете время, требуемое для отправки результата клиенту.

При условии достаточно полного профилирования вы легко сможете обнаружить проблему. Здесь нет никаких хитростей: если в общем времени выполнения вашего сценария большую часть составляет процессорное время, вам, вероятно, нужно подумать об оптимизации кода PHP. Однако иногда одни измерения маскируют другие. Например, активное использование процессорного времени может быть обусловлено ошибкой в программе, которая делает вашу систему кэширования неэффективной и заставляет приложение отправлять слишком много SQL-запросов.

Как демонстрирует этот пример, профилирование на уровне приложения является наиболее гибким и полезным приемом. Когда возможно, имеет смысл вставлять профилирование в любое приложение, в котором требуется выявлять узкие места.

Напоследок мы должны упомянуть, что продемонстрировали только простейшие приемы профилирования. Нашей целью в этом разделе было научить вас определять, не является ли MySQL источником проблем. Также можно профилировать сам код приложения. Например, решив, что нужно оптимизировать код PHP, поскольку он потребляет слишком много процессорного времени, вы можете воспользоваться такими инструментами, как *xdebug*, *Valgrind* и *cachegrind*.

Некоторые языки программирования имеют встроенную поддержку профилирования. Например, можно профилировать код Ruby с помощью параметра командной строки *-r*, а Perl – следующим образом:

```
$ perl -d:DProf <script file>
$ dprofpp tmon.out
```

Поиск в Интернете стоит начинать с запроса «профилирование *<язык>*» (или на английском profiling *<language>*).

Профилирование MySQL

В этом разделе мы более глубоко рассмотрим профилирование MySQL, поскольку оно меньше зависит от конкретного приложения. Иногда требуется профилировать одновременно и приложения, и сервер. Хотя профилирование приложения может дать более полную картину производительности всей системы, профилирование MySQL предоставляет много информации, которая недоступна, когда вы смотрите на приложение в целом. Например, профилирование PHP-кода не покажет, сколько строк MySQL просмотрел при выполнении запроса.

Как и в случае профилирования приложения, в данном случае основной целью является установить, на что MySQL тратит большую часть времени. Мы не будем профилировать исходный код MySQL. Хотя данная процедура иногда бывает полезной при нестандартной инсталляции СУБД, ее описание выходит за рамки настоящей книги. Вместо этого мы покажем некоторые приемы, которые можно использовать для получения и анализа информации о различных операциях, кото-

рые MySQL выполняет при формировании результатов пользовательского запроса.

Можно работать на любом уровне детализации, который отвечает вашим целям: профилировать сервер целиком либо исследовать отдельные запросы или пакеты запросов, чтобы собрать следующую информацию:

- К каким данным MySQL обращается чаще всего
- Какие типы запросов MySQL выполняет чаще всего
- В каких состояниях преимущественно находятся потоки (threads) MySQL
- Какие подсистемы MySQL чаще всего использует для выполнения запросов
- Какие виды обращения к данным встречаются наиболее часто
- Сколько различных видов действий, например просмотра индексов, выполняет MySQL

Мы начнем на самом высоком уровне – профилировании всего сервера – а дальше будем углубляться в детали.

Протоколирование запросов

В MySQL есть два типа журналов запросов: *общий журнал* и *журнал медленных запросов*. Оба предназначены для протоколирования запросов, но на разных этапах их выполнения. В общий журнал каждый запрос заносится в момент поступления серверу, поэтому там присутствуют даже те запросы, которые не были выполнены из-за возникших ошибок. Здесь регистрируются все запросы, а также некоторые события, например установление и разрыв соединения. Вы можете включить этот журнал с помощью одного конфигурационного параметра:

```
log = <имя_файла>
```

Естественно, общий журнал не содержит сведений о времени выполнения и иной информации, доступной только после завершения запроса. Напротив, журнал медленных запросов включает только данные о выполненных запросах. Точнее, здесь протоколируются запросы, выполнение которых заняло время, превышающее установленный порог. Для профилирования могут быть полезны оба журнала, но журнал медленных запросов является основным инструментом, позволяющим выявить проблемные запросы. Обычно мы рекомендуем включать его.

В следующем примере конфигурации этот журнал включается, и в нем регистрируются все запросы, выполнение которых занимает больше двух секунд, а также запросы, для обработки которых не были задействованы индексы. Кроме того, будут протоколироваться медленные административные запросы, например `OPTIMIZE TABLE`:

```
log-slow-queries                = <имя_файла>
long_query_time                 = 2
```

```
log-queries-not-using-indexes
log-slow-admin-statements
```

Вам нужно подстроить эти параметры под себя в конфигурационном файле сервера *my.cnf*. Более подробно о конфигурировании сервера рассказывается в главе 6.

Значением по умолчанию для параметра `long_query_time` является 10 секунд. В большинстве случаев это слишком много, поэтому мы обычно задаем две секунды. Однако во многих случаях и одной секунды более чем достаточно. В следующем разделе мы покажем пример тонкой настройки протоколирования в MySQL.

В версии MySQL 5.1 глобальные системные переменные `slow_query_log` и `slow_query_log_file` обеспечивают контроль над журналом медленных запросов во время выполнения, но в MySQL 5.0 невозможно включить или выключить этот журнал без перезапуска сервера. Традиционным обходным путем для MySQL 5.0 является использование переменной `long_query_time`, которую можно изменять динамически. Следующая команда на самом деле не отключает регистрацию медленных запросов, но имеет практически тот же эффект (если выполнение каких-то запросов занимает более 10 000 секунд, вам в любом случае надо их оптимизировать!):

```
mysql> SET GLOBAL long_query_time = 10000;
```

Сопутствующая конфигурационная переменная `log_queries_not_using_indexes` заставляет сервер записывать в журнал медленных запросов все запросы, которые не используют индексы, вне зависимости от того, насколько быстро они выполняются. Включение журнала медленных запросов обычно лишь немного увеличивает накладные расходы на выполнение «медленного» запроса. С другой стороны, запросы, не нуждающиеся в индексах, могут поступать часто и выполняться достаточно быстро (например, сканирование очень маленьких таблиц), поэтому их протоколирование способно замедлить работу сервера и даже привести к быстрому росту объема журнала.

К сожалению, в MySQL 5.0 невозможно включать или выключать протоколирование таких запросов с помощью динамически устанавливаемой переменной. Придется отредактировать конфигурационный файл, а затем перезапустить MySQL. Один из способов решения проблемы без перезапуска – сделать файл журнала символической ссылкой на */dev/null*, когда вы хотите отключить его (на самом деле этот фокус применим к любому файлу-протоколу). После такого изменения нужно выполнить команду `FLUSH LOGS`, тогда MySQL гарантированно закроет текущий дескриптор файла журнала и заново откроет его, перенаправив в */dev/null*.

В отличие от MySQL 5.0, версия MySQL 5.1 позволяет включать и выключать протоколирование во время исполнения и использовать для хранения журнала таблицы, к которым можно обращаться на языке SQL. Это большое усовершенствование.

Тонкая настройка протоколирования

Журнал медленных запросов в MySQL 5.0 и более ранних версиях имел ряд ограничений, которые делали его бесполезным для решения некоторых задач. Одной из подобных проблем являлось то, что минимальное значение переменной `long_query_time` в MySQL 5.0 было равно одной секунде, а шаг изменения тоже равнялся секунде. Для большинства интерактивных приложений это слишком долго. Разрабатывая высокопроизводительное веб-приложение, вы, вероятно, захотите, чтобы *вся страница* генерировалась меньше, чем за одну секунду. А ведь страница за время своего формирования, скорее всего, отправит много запросов. В этом контексте запрос, выполнение которого занимает 150 миллисекунд, будет, конечно, считаться очень медленным.

Другой проблемой является то, что вы не можете протоколировать все выполняемые сервером запросы в журнале медленных запросов (в частности, запросы от подчиненного сервера (slave thread's queries) не протоколируются). В общий журнал заносятся все запросы, но до того, как произведен синтаксический разбор, поэтому там нет информации о времени исполнения, времени блокировки и количестве просмотренных строк. Только журнал медленных запросов содержит такого рода сведения.

Наконец, при включенном режиме `log_queries_not_using_indexes` журнал медленных запросов может оказаться забитым записями о быстрых, эффективных запросах, которые выполняют полное сканирование таблицы. Например, если генерируется список штатов с помощью команды `SELECT * FROM STATES`, то этот запрос попадет в журнал, поскольку он выполняет полное сканирование таблицы.

При профилировании с целью оптимизации производительности вы ищете запросы, на которые приходится большая часть работы сервера MySQL. К таковым далеко не всегда относятся медленные запросы. Например, запрос продолжительностью 10 миллисекунд, который запускается 1000 раз в секунду, нагрузит сервер сильнее, чем десятисекундный запрос, запускаемый один раз в секунду. Чтобы выявить такую проблему, требуется протоколировать все запросы и проанализировать результаты.

Обычно имеет смысл смотреть как на медленные запросы (даже если они выполняются нечасто), так и на запросы, которые в общей сложности потребляют большую часть ресурсов сервера. Это поможет выявить различные проблемы, в частности определить запросы, вызывающие недовольство пользователей.

Мы разработали заплату для сервера MySQL, основанную на работе Георга Рихтера (Georg Richter), которая позволяет задавать пороговое значение для медленных запросов в миллисекундах вместо секунд. Она также позволяет записывать в журнал медленных запросов *все* запросы путем установки `long_query_time=0`. Эту заплату можно скачать со стра-

ницы *http://www.mysqlperformanceblog.com/mysql-patches/*. Ее главный недостаток заключается в неоходимости перекомпилировать MySQL самостоятельно, поскольку заплата не включена в официальный дистрибутив MySQL до версии 5.1.

На момент работы над этой книгой версия заплаты, включенная в MySQL 5.1, изменяет только точность задания времени. Новая редакция, еще не включенная в официальную поставку MySQL, добавляет больше полезной функциональности. Она включает идентификатор соединения, по которому поступил запрос, а также информацию о кэше запросов, типе операции соединения, временных таблицах и сортировке. Также включена статистика InnoDB: сведения о поведении системы ввода-вывода и ожиданиях блокировок.

Новая заплата позволяет протоколировать запросы, выполняемые потоком SQL на подчиненном сервере, что очень важно, если имеется проблема с запаздыванием репликации на подчиненных серверах (о том как «подстегнуть» подчиненные серверы, см. раздел «Слишком большое отставание репликации» главы 8 на стр. 494). Она также позволяет выборочно протоколировать только некоторые сеансы. Этого обычно достаточно для целей профилирования, и мы думаем, что это хорошая практика.

Данная заплата относительно новая, так что, если вы накладываете ее самостоятельно, будьте осторожны. Мы полагаем, что она вполне безопасна, но все же ее не подвергали такому тщательному тестированию, как остальную часть сервера MySQL. Если вас беспокоит стабильность сервера после наложения заплаты, то не обязательно запускать «залатанную» версию каждый раз. Можно загрузить ее на несколько часов, запротоколировать некоторые запросы, а затем вернуться к оригинальной версии.

При профилировании имеет смысл протоколировать все запросы с параметром `long_query_time=0`. Если большая часть нагрузки приходится на очень простые запросы, то об этом надо знать. Протоколирование всех подобных запросов окажет некоторое влияние на производительность и потребует существенно больше места на диске – это еще одна причина, по которой не стоит регистрировать все запросы постоянно. К счастью, вы можете изменить параметр `long_query_time` без перезапуска сервера, так что несложно получить выборку всех запросов за небольшое время, а затем вернуться к протоколированию только очень медленных запросов.

Как читать журнал медленных запросов

Вот пример из журнала медленных запросов:

```
1 # Time: 030303 0:51:27
2 # User@Host: root[root] @ localhost []
3 # Query_time: 25 Lock_time: 0 Rows_sent: 3949 Rows_examined: 378036
4 SELECT ...
```

Строка 1 показывает, когда запрос был зарегистрирован, а строка 2 – кто его исполнил. Строка 3 демонстрирует, сколько секунд заняло выполнение запроса, как долго он ждал блокировки таблицы на уровне сервера MySQL (не на уровне подсистемы хранения), сколько строк запрос вернул и сколько строк он просмотрел. Все эти строки закомментированы, так что они не будут выполняться, если вы загрузите журнал в клиент MySQL. Последняя строка – это собственно запрос.

Вот пример с сервера MySQL 5.1:

```
1 # Time: 070518 9:47:00
2 # User@Host: root[root] @ localhost []
3 # Query_time: 0.000652 Lock_time: 0.000109 Rows_sent: 1 Rows_examined: 1
4 SELECT ...
```

Информация в основном одна и та же, не считая того, что время в строке 3 имеет большую точность. В более новой версии заплаты содержится еще больше информации:

```
 1 # Time: 071031 20:03:16
 2 # User@Host: root[root] @ localhost []
 3 # Thread_id: 4
 4 # Query_time: 0.503016 Lock_time: 0.000048 Rows_sent: 56 Rows_examined: 1113
 5 # QC_Hit: No Full_scan: No Full_join: No Tmp_table: Yes Disk_tmp_table: No
 6 # Filesort: Yes Disk_filesort: No Merge_passes: 0
 7 #   InnoDB_IO_r_ops: 19 InnoDB_IO_r_bytes: 311296 InnoDB_IO_r_wait: 0.382176
 8 #   InnoDB_rec_lock_wait: 0.000000 InnoDB_queue_wait: 0.067538
 9 #   InnoDB_pages_distinct: 20
10 SELECT ...
```

Строка 5 показывает, был ли запрос обслужен из кэша, было ли выполнено полное сканирование таблицы, имела ли место операция соединения без индексов, использовалась ли временная таблица, а если да, то была ли она создана на диске. Строка 6 информирует нас о том, выполнял ли запрос файловую сортировку, и если да, была ли она произведена на диске и сколько проходов слияния было выполнено.

Строки 7, 8 и 9 появятся, если в запросе использовались таблицы типа InnoDB. Строка 7 демонстрирует, сколько операций чтения страниц InnoDB запланировала при выполнении запроса, с соответствующим значением в байтах. Последним значением в строке 7 является время, потребовавшееся подсистеме InnoDB для чтения данных с диска. Строка 8 показывает, как долго запрос ждал блокировок строк и как долго он ожидал входа в ядро InnoDB[1].

Строка 9 содержит сведения о приблизительном количестве уникальных страниц InnoDB, к которым обращался запрос. Чем больше это число, тем меньше его точность. Данную информацию можно, в частности, применить для оценки количества страниц в рабочем набора запроса;

[1] Информацию о ядре InnoDB см. в разделе «Настройка конкурентного доступа для InnoDB» в главе 6 на стр. 370.

оно характеризует кэширование данных в пуле буферов InnoDB. Эта величина также показывает, насколько в действительности полезны имеющиеся кластерные индексы. Если строки запроса хорошо кластеризованы, они поместятся в меньшем количестве страниц. Подробнее см. раздел «Кластерные индексы» главы 3 на стр. 152.

Использование журнала медленных запросов для разрешения проблем с долго выполняющимися запросами не всегда столь очевидно. Хотя журнал содержит много полезной информации, один очень важный момент упущен: *почему* все-таки запрос оказался медленным. Иногда это понятно сразу. Если журнал говорит, что было просмотрено 12 000 000 строк, а клиенту отправлено 1 200 000, вы знаете, почему запрос выполняется медленно – он очень большой! Однако так просто проблема решается далеко не всегда.

Не слишком доверяйтесь журналу медленных запросов. Если вы видите в нем один и тот же запрос, повторяющийся много раз, есть большая вероятность, что он медленный и нуждается в оптимизации. Но один лишь факт, что запрос появляется в этом журнале, не означает, что это плохой запрос, и даже не обязательно медленный. Вы можете найти медленный запрос, запустить его самостоятельно и выяснить, что он выполняется за долю секунды. Появление в журнале просто говорит о том, что *в тот момент* он обрабатывался долго, но отнюдь не означает, что это повторится сейчас или в будущем. Есть много причин, почему запрос может быть в одних случаях медленным, а в других – быстрым.

- Таблица могла быть заблокирована, поэтому запрос был вынужден ждать. Величина `Lock_time` показывает, как долго запрос ждал освобождения блокировки.
- Данные или индексы могли к тому моменту еще отсутствовать в кэше. Это обычный случай, когда сервер MySQL только запущен или не настроен должным образом.
- Мог идти ночной процесс резервного копирования, из-за чего все операции дискового ввода/вывода замедлялись.
- Сервер мог обрабатывать в тот момент другие запросы, поэтому данный выполнялся медленнее.

Журнал медленных запросов не дает полной картины имевших место событий. Вы можете использовать его для построения списка «подозреваемых», но потом необходимо провести более глубоко исследование.

Заплаты для журнала медленных запросов специально разрабатывались, чтобы помочь вам понять, почему запрос работает медленно. В частности, если вы используете InnoDB, очень полезной может оказаться статистика InnoDB: она демонстрирует, ждал ли запрос дискового ввода/вывода, долго ли он стоял в очереди InnoDB и т. п.

Инструменты анализа журналов

Теперь, когда запросы запротоколированы, пришло время проанализировать результаты. Общая стратегия заключается в том, чтобы найти запросы, которые влияют на сервер больше всего, проверить их планы выполнения с помощью команды `EXPLAIN` и настроить нужным образом. Повторите полный анализ после настройки, поскольку сделанные изменения могут повлиять на другие запросы. Например, следует ожидать, что новые индексы ускорят запросы `SELECT`, но замедлят запросы `INSERT` и `UPDATE`.

В общем случае при исследовании журналов следует обращать внимание на следующие три вещи:

Долго выполняющиеся запросы

Периодические выполняемые пакетные задания действительно могут запускать долго выполняющиеся запросы, но обычные запросы не должны занимать много времени.

Запросы, больше всего нагружающие сервер

Ищите запросы, которые потребляют большую часть времени сервера. Напомним, что короткие запросы, выполняемые очень часто, тоже могут занимать много времени.

Новые запросы

Ищите запросы, которых вчера не было в первой сотне, а сегодня они появились. Это могут быть новые запросы или запросы, которые обычно выполнялись быстро, а теперь замедлились из-за изменившейся схемы индексации. Либо произошли еще какие-то изменения.

Если журнал медленных запросов достаточно мал, это легко сделать вручную, но если вы протоколируете все запросы (как мы предлагаем), то понадобятся специальные инструменты. Вот некоторые из наиболее распространенных приложений, предназначенных для этой цели:

mysqldumpslow

Программа *mysqldumpslow* поставляется с сервером MySQL. Это сценарий на языке Perl, который умеет агрегировать журнал медленных запросов и показывать, сколько раз каждый запрос появляется в журнале. Таким образом, вы не будете тратить время, пытаясь оптимизировать 30-секундный медленный запрос, который запускается раз в день, когда есть много других, более коротких медленных запросов, запускающихся тысячи раз в день.

Преимуществом программы *mysqldumpslow* является то, что она уже имеется в комплекте с СУБД. Недостаток ее в том, что она несколько менее гибкая, чем некоторые другие инструменты. Кроме того, она плохо документирована и не понимает журналы, записанные на серверах, где установлена заплата, благодаря которой время выражается в микросекундах.

mysql_slow_log_filter

Этот инструмент, который можно скачать со страницы *http://www.mysqlperformanceblog.com/files/utils/mysql_slow_log_filter*, понимает формат времени с микросекундной точностью. Его можно использовать для извлечения запросов, которые исполняются дольше определенного порога или просматривают больше указанного количества строк. Он хорош для «урезания» файла журнала в случае, когда установлена микросекундная заплата, вызывающая слишком быстрый рост объема журнала без фильтрации. Вы можете запустить его на некоторое время с высокими порогами, оптимизировать худшие запросы, затем изменить параметры для выявления следующих запросов и продолжить настройку.

Следующая команда показывает запросы, которые либо выполняются дольше половины секунды, либо просматривают больше тысячи строк:

```
$ tail -f mysql-slow.log | mysql_slow_log_filter -T 0.5 -R 1000
```

mysql_slow_log_parser

Это еще один инструмент, доступный на странице *http://www.mysqlperformanceblog.com/files/utils/mysql_slow_log_parser*, который может агрегировать микросекундный журнал медленных запросов. В дополнение к агрегированию и созданию отчетов он показывает минимальное и максимальное значения времени выполнения, количество проанализированных строк, выводит запрос в каноническом виде и демонстрирует реальный запрос, к которому можно применить команду `EXPLAIN`. Вот пример распечатки:

```
### 3579 Queries
### Total time: 3.348823, Average time: 0.000935686784017883
### Taking 0.000269 to 0.130820 seconds to complete
### Rows analyzed 1 - 1
SELECT id FROM forum WHERE id=XXX;
SELECT id FROM forum WHERE id=12345;
```

mysqlsla

Программа MySQL Statement Log Analyzer, доступная на странице *http://hackmysql.com/mysqlsla*, может анализировать не только журнал медленных запросов, но и общий журнал, а также журналы, содержащие команды SQL с разделителями. Подобно *mysql_slow_log_parser* она позволяет выводить запрос в каноническом виде и подводить итоги. Данная программа также может применять к запросам команду `EXPLAIN` (она переписывает многие отличные от `SELECT` запросы так, чтобы они были пригодны для работы `EXPLAIN`) и генерировать сложные отчеты.

Иногда статистику журналов медленных запросов используют для прогнозирования величины, на которую можно уменьшить потребление

ресурсов сервера. Предположим, вы сделали выборку запросов за один час (3 600 секунд) и обнаружили, что суммарное время выполнения всех запросов в журнале составляет 10 000 секунд (суммарное время больше, чем реально истекшее, поскольку запросы выполняются параллельно). Если анализ журнала покажет, что на худший запрос ушло 3000 секунд, значит, этот запрос ответственен за 30% нагрузки. Теперь вы знаете, насколько удастся уменьшить потребление ресурсов сервера путем оптимизации этого запроса.

Профилирование сервера MySQL

Один из лучших способов профилировать сервер – то есть увидеть, на что он тратит большую часть времени, заключается в использовании команды `SHOW STATUS`, которая возвращает много информации о состоянии. Здесь мы упомянем только некоторые из выводимых ею переменных.

Команда `SHOW STATUS` ведет себя несколько каверзно и это может привести к плохим результатам в MySQL 5.0 и более новых версиях. Подробнее о поведении команды `SHOW STATUS` и связанных с ней проблемах рассказано в главе 13.

Чтобы увидеть в режиме, близком к реальному времени, как работает ваш сервер, периодически запускайте команду `SHOW STATUS` и сравнивайте результат с предыдущим запуском. Вы можете делать это с помощью следующей команды:

```
mysqladmin extended -r -i 10
```

Некоторые из переменных не являются строго возрастающими счетчиками, так что вы можете увидеть странные результаты типа отрицательного значения `Threads_running`. Не беспокойтесь по этому поводу, просто счетчик уменьшился с момента предыдущей выборки.

Поскольку информации выводится много, имеет смысл пропустить ее через утилиту *grep*, чтобы отфильтровать переменные, которые вас не интересуют. В качестве альтернативы для просмотра результатов можно использовать программу *innotop* или какой-нибудь другой инструмент из описанных в главе 14. Вот некоторые наиболее полезные переменные:

`Bytes_received` и `Bytes_sent`

Количество байтов, соответственно полученных и отправленных сервером.

`Com_*`

Команды, которые сервер выполняет

`Created_*`

Временные таблицы и файлы, созданные во время выполнения запроса

`Handler_*`

Операции подсистемы хранения

`Select_*`

Различные типы планов выполнения операции соединения

`Sort_*`

Разнообразная информация о сортировке

Вы можете использовать этот подход для мониторинга внутренних операций MySQL, например количества обращений к ключу (key accesses), считываний ключа с диска для таблиц типа MyISAM, частоты доступа к данным, считываний данных с диска для таблиц типа InnoDB, и т. д. Это поможет определить, где находятся реальные или потенциальные узкие места вашей системы, даже не глядя на отдельный запрос. Вы также можете использовать инструменты, которые анализируют результаты, выданные командой `SHOW STATUS`, для получения мгновенного снимка общего состояния сервера, например *mysqlreport*.

Здесь мы не будем углубляться в подробную семантику переменных состояния, а объясним их использование на примерах, так что не беспокойтесь о том, что вы не знаете, что они означают.

Еще одним хорошим способом профилирования сервера MySQL является использование команды `SHOW PROCESSLIST`. Она позволяет видеть не только типы выполняющихся запросов, но и состояние соединений. Некоторые вещи, например большое количество соединений в состоянии `Locked`, являются явными указаниями на узкие места. Как и в случае команды `SHOW STATUS`, вывод `SHOW PROCESSLIST` достаточно подробен. Обычно гораздо удобнее использовать такие инструменты, как *innotop*, чем разбирать его вручную.

Профилирование запросов с помощью команды SHOW STATUS

Комбинация команд `FLUSH STATUS` и `SHOW SESSION STATUS` помогает увидеть что происходит, когда MySQL исполняет запрос или пакет запросов. Это отличный способ осуществить их оптимизацию.

Давайте на примере посмотрим, как интерпретировать результаты запроса. Сначала с помощью команды `FLUSH STATUS` сбросим значения переменных состояния в нуль, чтобы вы могли увидеть, что делает MySQL для выполнения запроса:

```
mysql> FLUSH STATUS;
```

Затем запустите запрос. Мы добавили параметр `SQL_NO_CACHE`, чтобы MySQL не обслуживал его из кэша:

```
mysql> SELECT SQL_NO_CACHE film_actor.actor_id, COUNT(*)
    -> FROM sakila.film_actor
```

```
    ->   INNER JOIN sakila.actor USING(actor_id)
    -> GROUP BY film_actor.actor_id
    -> ORDER BY COUNT(*) DESC;
...
200 rows in set (0.18 sec)
```

Запрос вернул 200 строк, но что он делал на самом деле? Команда `SHOW SESSION STATUS` может дать некоторое представление об этом. Сначала посмотрим, какой план выполнения запроса выбрал сервер:

```
mysql> SHOW SESSION STATUS LIKE 'Select%';
+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| Select_full_join       | 0     |
| Select_full_range_join | 0     |
| Select_range           | 0     |
| Select_range_check     | 0     |
| Select_scan            | 2     |
+------------------------+-------+
```

Похоже, MySQL выполнил полное сканирование таблицы (указано даже два сканирования, но это особенность команды `SHOW SESSION STATUS`, к которой мы вернемся ниже). Если в запросе участвует более одной таблицы, то несколько переменных могут иметь значения больше нуля. Например, если сервер использовал поиск по диапазону в последующей (в порядке соединения) таблице, то переменная `Select_full_range_join` примет отличное от нуля значение. Можно пойти еще дальше, взглянув на низкоуровневые операции подсистемы хранения, которые были произведены при выполнении запроса:

```
mysql> SHOW SESSION STATUS LIKE 'Handler%';
+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| Handler_commit             | 0     |
| Handler_delete             | 0     |
| Handler_discover           | 0     |
| Handler_prepare            | 0     |
| Handler_read_first         | 1     |
| Handler_read_key           | 5665  |
| Handler_read_next          | 5662  |
| Handler_read_prev          | 0     |
| Handler_read_rnd           | 200   |
| Handler_read_rnd_next      | 207   |
| Handler_rollback           | 0     |
| Handler_savepoint          | 0     |
| Handler_savepoint_rollback | 0     |
| Handler_update             | 5262  |
| Handler_write              | 219   |
+----------------------------+-------+
```

Высокие показатели операций «read» (чтение) демонстрируют, что MySQL пришлось просканировать для удовлетворения запроса более одной таблицы. Обычно, если MySQL читает только одну таблицу с полным сканированием, мы видим большие значения переменной `Handler_read_rnd_next`, а переменная `Handler_read_rnd` равна нулю.

В данном случае многочисленные ненулевые значения указывают на то, что MySQL, должно быть, использовал временную таблицу для обработки фраз `GROUP BY` и `ORDER BY`. Вот почему у переменных `Handler_write` и `Handler_update` ненулевые показатели: MySQL, вероятно, произвел запись во временную таблицу, просканировал ее с целью сортировки, а затем просканировал еще раз, чтобы вывести результаты в отсортированном порядке. Давайте посмотрим, что делал сервер для упорядочивания полученных данных:

```
mysql> SHOW SESSION STATUS LIKE 'Sort%';
+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| Sort_merge_passes | 0     |
| Sort_range        | 0     |
| Sort_rows         | 200   |
| Sort_scan         | 1     |
+-------------------+-------+
```

Как мы и предполагали, MySQL отсортировал результат путем сканирования временной таблицы, содержащей все отобранные строки. Если бы значение было больше, чем 200 строк, мы предположили бы, что сортировка происходила в какой-то другой точке выполнения запроса. Мы также можем увидеть, сколько временных таблиц создал MySQL для этого запроса:

```
mysql> SHOW SESSION STATUS LIKE 'Created%';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| Created_tmp_disk_tables | 0     |
| Created_tmp_files       | 0     |
| Created_tmp_tables      | 5     |
+-------------------------+-------+
```

Приятно видеть, что запросу не потребовалось использовать диск для временных таблиц, поскольку это значительно снизило бы скорость. Однако есть одна странность – неужели MySQL создал пять временных таблиц только для обработки одного этого запроса?

Фактически запросу требуется лишь одна временная таблица. Это та самая особенность, о которой мы уже упоминали ранее. Что происходит? Мы запускали пример в MySQL 5.0.45, а в MySQL 5.0 команда `SHOW SESSION STATUS` в действительности выбирает данные из таблиц

INFORMATION_SCHEMA, что является «платой за наблюдение»[1]. Это несколько искажает результат, в чем вы можете убедиться, еще раз запустив SHOW STATUS:

```
mysql> SHOW SESSION STATUS LIKE 'Created%';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| Created_tmp_disk_tables | 0     |
| Created_tmp_files       | 0     |
| Created_tmp_tables      | 6     |
+-------------------------+-------+
```

Обратите внимание, что значение снова увеличилось. Переменная Handler и другие переменные поменялись подобным же образом. Ваши результаты могут отличаться в зависимости от версии MySQL.

Можно применить тот же самый подход для профилирования в MySQL 4.1 и более старых версиях: выполнить команду FLUSH STATUS, запустить запрос и затем выполнить директиву SHOW STATUS. Нужно только делать это на неиспользуемом сервере, поскольку в более старых версиях счетчики глобальны и могут быть изменены другими процессами.

Лучшим способом компенсировать «плату за наблюдение», вызванную запуском команды SHOW STATUS, является вычисление стоимости путем двукратного запуска команды и вычитания второго результата из первого. В дальнейшем вы сможете вычитать это число из SHOW STATUS, чтобы получить реальную стоимость запроса. Чтобы получить точные результаты, нужно знать область видимости переменных, это позволит понять, за наблюдение каких переменных приходится «платить стоимость наблюдения». Некоторые переменные являются сеансовыми, другие – глобальными. Вы можете автоматизировать этот сложный процесс с помощью инструмента *mk-query-profiler*.

Этот способ автоматического профилирования можно внедрить в код подключения к базе данных вашего приложения. Когда профилирование включено, код подключения может автоматически сбрасывать состояние перед каждым запросом и затем протоколировать различия. В качестве альтернативы можно вести профилирование по страницам, а не по запросам. Обе стратегии полезны для определения того, какую работу MySQL выполняет во время запроса.

Команда SHOW PROFILE

Команда SHOW PROFILE представляет собой заплату, которую Джереми Коул (Jeremy Cole) предложил для «общественной» (Community) вер-

[1] Проблема «платы за наблюдения» исправлена в MySQL 5.1 для команды SHOW SESSION STATUS.

сии MySQL 5.0.37[1]. Профилирование по умолчанию отключено, но его можно включить на уровне сеанса. При этом сервер MySQL будет собирать информацию о ресурсах, использованных при выполнении запроса. Чтобы начать сбор статистики, присвойте переменной `profiling` значение 1:

```
mysql> SET profiling = 1;
```

Теперь запустим запрос:

```
mysql> SELECT COUNT(DISTINCT actor.first_name) AS cnt_name, COUNT(*) AS cnt
    -> FROM sakila.film_actor
    -> INNER JOIN sakila.actor USING(actor_id)
    -> GROUP BY sakila.film_actor.film_id
    -> ORDER BY cnt_name DESC;
...
997 rows in set (0.03 sec)
```

Данные о профилировании этого запроса сохранены в сеансе. Чтобы посмотреть запросы, которые были профилированы, воспользуемся командой `SHOW PROFILES`:

```
mysql> SHOW PROFILES\G
*************************** 1. row ***************************
 Query_ID: 1
 Duration: 0.02596900
    Query: SELECT COUNT(DISTINCT actor.first_name) AS cnt_name,...
```

Вы можете извлечь сохраненные данные профилирования с помощью команды `SHOW PROFILE`. При запуске без аргументов она показывает каждое состояние и время нахождения в нем для последней выполненной команды:

```
mysql> SHOW PROFILE;
+----------------------+----------+
| Status               | Duration |
+----------------------+----------+
| (initialization)     | 0.000005 |
| Opening tables       | 0.000033 |
| System lock          | 0.000037 |
| Table lock           | 0.000024 |
| init                 | 0.000079 |
| optimizing           | 0.000024 |
| statistics           | 0.000079 |
| preparing            | 0.00003  |
| Creating tmp table   | 0.000124 |
| executing            | 0.000008 |
| Copying to tmp table | 0.010048 |
| Creating sort index  | 0.004769 |
```

[1] На момент написания этих строк команда `SHOW PROFILE` все еще не включена в версии Enterprise MySQL, даже в более новые, чем 5.0.37.

```
| Copying to group table | 0.0084880 |
| Sorting result         | 0.001136  |
| Sending data           | 0.000925  |
| end                    | 0.00001   |
| removing tmp table     | 0.00004   |
| end                    | 0.000005  |
| removing tmp table     | 0.00001   |
| end                    | 0.000011  |
| query end              | 0.00001   |
| freeing items          | 0.000025  |
| removing tmp table     | 0.00001   |
| freeing items          | 0.000016  |
| closing tables         | 0.000017  |
| logging slow query     | 0.000006  |
+------------------------+-----------+
```

Каждая строка описывает одно состояние процесса и указывает, как долго он находился в этом состоянии. Столбец `Status` соответствует столбцу `State` в выводе команды `SHOW FULL PROCESSLIST`. Значения берутся из структуры `thd->proc_info`, то есть непосредственно из внутренней структуры данных MySQL. Поля этой структуры документированы в руководстве по MySQL, хотя большинство из них имеет интуитивно понятные имена, и разобраться в них нетрудно.

Вы можете указать запрос, который следует профилировать, передав его идентификатор `Query_ID`, полученный от команд `SHOW PROFILES`. Кроме того, можно попросить вывести дополнительные столбцы. Например, чтобы увидеть, сколько времени процессор работал в режиме пользователя и ядра, используйте следующую команду:

```
mysql> SHOW PROFILE CPU FOR QUERY 1;
```

Команда `SHOW PROFILE` позволяет разобраться в том, что делает сервер при выполнении запроса, и помогает понять, на что в реальности уходит время. К числу ее ограничений можно отнести неспособность видеть и профилировать запросы, выполненные другим соединением, а также накладные расходы, вызванные профилированием.

Другие способы профилирования MySQL

В этой главе мы достаточно подробно показали, как использовать информацию о внутреннем состоянии MySQL, чтобы узнать, что происходит внутри сервера. Можно заниматься профилированием, наблюдая за другими отчетами о состоянии MySQL. В частности, полезны команды `SHOW INNODB STATUS` и `SHOW MUTEX STATUS`. Об этих и других командах мы будем более подробно говорить в главе 13.

Когда не удается добавить код профилирования

Иногда вы не можете ни добавить код профилирования, ни наложить заплату на сервер, ни даже изменить его конфигурацию. Однако обыч-

но все же есть способ выполнить хоть какое-то профилирование. Попробуйте использовать следующие идеи.

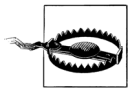
- Настройте журналы своего веб-сервера, чтобы они записывали общее время выполнения и процессорное время, потребленное каждым запросом.
- Используйте анализаторы трафика (снифферы) для перехвата и хронометража запросов (включая сетевые задержки), передаваемых по сети. В число бесплатных мониторов входят *mysqlsniffer* (*http://hackmysql.com/mysqlsniffer*) и *tcpdump*. Примеры использования программы *tcpdump* можно найти по адресу *http://forge.mysql.com/snippets/view.php?id=15*.
- Используйте прокси-серверы, такие как MySQL Proxy, для перехвата и хронометража запросов.

Профилирование операционной системы

Иногда бывает полезно получить статистику операционной системы и попытаться выяснить, что же делают ОС и оборудование. Это может помочь не только при профилировании приложения, но и в процессе поиска неполадок.

Признаемся, что в этом разделе мы уклонились в сторону UNIX-подобных системных платформ, поскольку именно с ними работаем чаще всего. Однако те же приемы можно использовать и в других операционных системах при условии, что они предоставляют статистику.

Чаще всего мы используем инструменты *vmstat*, *iostat*, *mpstat* и *strace*. Каждый из них позволяет под разными углами взглянуть на сочетание процесса, памяти, процессора и операций ввода/вывода. Эти инструменты имеются в большинстве UNIX-подобных систем. Примеры того, как их использовать, мы покажем на протяжении книги, особенно в конце главы 7.



Будьте осторожны при использовании программы *strace* в GNU/Linux на рабочих серверах. Похоже, она не всегда правильно работает с многопоточными процессами, и нам доводилось видеть, как ее использование приводило к аварийной остановке сервера.

Поиск неполадок, относящихся к процессам и соединениям MySQL

Мы нигде еще не обсуждали подробно инструменты для мониторинга сетевой активности и простейшего поиска неполадок. В качестве примера покажем, каким образом можно проследить соединение с сервером MySQL в обратном направлении до источника на другом сервере (например, до сервера Apache).

Начнем с вывода команды SHOW PROCESSLIST в MySQL и обратим внимание на столбец Host. Мы будем использовать следующий пример:

```
*************************** 21. row ***************************
     Id: 91296
   User: web
   Host: sargon.cluster3:37636
     db: main
Command: Sleep
   Time: 10
  State:
   Info: NULL
```

Столбец Host показывает, откуда пришло соединение, и, что не менее важно, порт TCP, с которого оно пришло. Вы можете использовать эту информацию для выяснения того, какой процесс открыл соединение. Если у вас есть доступ к хосту sargon с правами пользователя *root*, вы можете использовать программу *netstat* и номер порта, чтобы узнать, какой процесс открыл соединение:

```
root@sargon# netstat -ntp | grep :37636
tcp 0 0 192.168.0.12:37636 192.168.0.21:3306 ESTABLISHED 16072/apache2
```

Номер и название процесса указываются в последнем поле: это соединение инициировал процесс 16072 на сервере Apache. Зная идентификатор процесса, вы можете получить о нем множество сведений, например, какими еще сетевыми соединениями процесс владеет:

```
root@sargon# netstat -ntp | grep 16072/apache2
tcp 0 0 192.168.0.12:37636 192.168.0.21:3306 ESTABLISHED 16072/apache2
tcp 0 0 192.168.0.12:37635 192.168.0.21:3306 ESTABLISHED 16072/apache2
tcp 0 0 192.168.0.12:57917 192.168.0.3:389 ESTABLISHED 16072/apache2
```

Похоже, что рабочий процесс Apache открыл два соединения с MySQL (порт 3306) и какое-то соединение с портом 389 на другой машине. Что такое порт 389? Многие программы используют стандартизованные номера портов, например для MySQL портом по умолчанию является 3306. Список служб часто находится в файле */etc/services*, так что посмотрим, что там имеется:

```
root@sargon# grep 389 /etc/services
ldap            389/tcp # Lightweight Directory Access Protocol
ldap            389/udp
```

Нам известно, что этот сервер использует аутентификацию по протоколу LDAP, так что упоминание о LDAP имеет смысл. Давайте посмотрим, что еще можно выяснить о процессе 16072. Что делает процесс, легко узнать с помощью утилиты *ps*. Примененный нами шаблон для программы *grep* позволяет кроме строки процесса 16072 посмотреть на первую строку распечатки, в которой выводятся заголовки столбцов:

```
root@sargon# ps -eaf | grep 'UID\|16072'
UID PID PPID C STIME TTY TIME CMD
```

```
apache 16072 22165 0 09:20 ? 00:00:00 /usr/sbin/apache2 -D DEFAULT_VHOST...
```

Потенциально вы можете использовать эту информацию для поиска других проблем. Не удивляйтесь, например, если обнаружите, что обращения из Apache к службам LDAP или NFS приводят к медленной генерации страниц.

Помимо этого можно просмотреть список файлов, открытых процессом, с помощью команды *lsof*. Она позволяет получить самую разнообразную информацию, поскольку в UNIX все является файлами. Полную распечатку мы здесь приводить не будем, поскольку она чрезвычайно подробна, но чтобы узнать, какие файлы открыл интересующий нас процесс, достаточно выполнить команду *lsof | grep 16072*. Можно также использовать *lsof* для поиска сетевых соединений, если *netstat* отсутствует. Например, ниже мы воспользовались *lsof* для получения примерно той же информации, колторую дает *netstat*. Вывод команды мы слегка переформатировали:

```
root@sargon# lsof -i -P | grep 16072
apache2 16072 apache 3u IPv4 25899404 TCP *:80 (LISTEN)
apache2 16072 apache 15u IPv4 33841089 TCP sargon.cluster3:37636->
                                           hammurabi.cluster3:3306
(ESTABLISHED)
apache2 16072 apache 27u IPv4 33818434 TCP sargon.cluster3:57917->
                                           romulus.cluster3:389
(ESTABLISHED)
apache2 16072 apache 29u IPv4 33841087 TCP sargon.cluster3:37635->
                                           hammurabi.cluster3:3306
(ESTABLISHED)
```

В GNU/Linux еще одним ценным помощником при поиске неполадок является файловая система */proc*. Каждый процесс имеет внутри */proc* свой каталог, в котором присутствует много информации о нем, например путь к текущей рабочей папке, сведения об использовании памяти и т. п.

Сервер Apache имеет функцию, аналогичную команде UNIX *ps*: URL */server-status/*. Например, если в вашей внутренней сети сервер Apache доступен по адресу *http://intranet/*, то вы можете набрать в броузере адрес *http://intranet/server-status/*, чтобы увидеть, что делает Apache. Это может быть полезно, когда нужно выяснить, какой URL обслуживает процесс. Страница содержит описание с пояснениями к выведенной информации.

Более развитые средства профилирования и поиска неполадок

Если вам нужно глубже разобраться в том, что происходит, например, понять, почему процесс ушел в спячку и это состояние не удается прервать, вы можете использовать команды *strace -p* и/или *gdb -p*. Они умеют показывать системные вызовы и обратную трассировку стека, а это

позволит вам узнать, что делал процесс перед тем как завис. Причин может быть множество, например, аварийно завершившая свою работу служба блокировок NFS, обращение к удаленному веб-сервису, который не отвечает, и т. д.

Существует возможность более детального профилирования системы или части систем с целью выяснить, что они делают. Если вам действительно нужна высокая производительность и у вас начинаются проблемы, в ваших силах даже профилировать внутреннюю работу MySQL. Хотя это может показаться не вашей заботой (это дело разработчиков MySQL, не правда ли?), тем не менее, перепрофилирование – один из способов изолировать часть системы, которая вызывает проблемы. Возможно, вы не сможете или не захотите исправить их, но, по крайней мере, сумеете спроектировать свое приложение так, чтобы обойти это место. Вот некоторые полезные инструменты:

OProfile

Программа OProfile (*http://oprofile.sourceforge.net*) – это системный профилировщик для Linux. Он состоит из драйвера ядра и демона для получения выборки данных, а также нескольких инструментов, которые помогут проанализировать собранные данные. Он профилирует весь код, включая обработчики прерывания, ядро, модули ядра, приложения и совместно используемые библиотеки. Если приложение скомпилировано с символами отладки, то OProfile может аннотировать исходный код, но это не обязательно. Вы можете профилировать систему, ничего не перекомпилируя. У этого профилировщика относительно низкие накладные расходы, обычно в пределах нескольких процентов.

gprof

gprof представляет собой профилировщик GNU, который порождает профили исполнения программ, скомпилированных с параметром *-pg*. Он вычисляет время, проведенное в каждой подпрограмме. *gprof* может создавать отчеты по частоте и продолжительности вызовов функции, генерировать граф вызовов и аннотированный исходный код.

Прочие инструменты

Существует много других инструментов, включая специализированные и/или коммерческие. В их число входят Intel VTune, Sun Performance Analyzer (часть Sun Studio) и DTrace для Solaris и других ОС.

3

Оптимизация схемы и индексирование

Оптимизация неудачно спроектированной или плохо проиндексированной схемы может увеличить производительность на порядки. Если вам требуется высокое быстродействие, вы должны разработать схему и индексы под те конкретные запросы, которые будете запускать. Кроме того, следует оценить требования к производительности для различных типов запросов, поскольку изменения в одном из них или в одной части схемы могут иметь последствия в других местах. Оптимизация часто требует компромиссов. Например, добавление индексов для ускорения выборки данных замедляет их изменение. Аналогично денормализованная схема ускоряет некоторые типы запросов, но замедляет другие. Добавление таблиц счетчиков и сводных таблиц является хорошим способом оптимизации запросов, но их поддержка может стоить дорого.

Иногда приходится выходить из роли разработчика и исследовать требования бизнеса, который вам поручено автоматизировать. Люди, не являющиеся экспертами в области баз данных, часто формулируют бизнес-требования, не понимая, как они повлияют на производительность. Если вы сможете объяснить, что незначительная, на первый взгляд, функция удвоит требования к оборудованию, они могут решить, что без нее вполне можно обойтись.

Оптимизация схемы и индексирование потребуют как взгляда на картину в целом, так и внимания к деталям. Вам нужно понимать всю систему, чтобы разобраться, как каждая ее часть влияет на остальные. Эта глава начинается с обсуждения типов данных, далее речь идет о стратегии индексирования и нормализации, а в конце приведено несколько замечаний о подсистемах хранения.

Вам, вероятно, потребуется заново просмотреть эту главу после прочтения раздела об оптимизации запросов. Многие из обсуждаемых здесь тем – особенно индексирование – нельзя рассматривать в изоляции. Чтобы принимать правильные решения об индексировании, вам нужно иметь представление об оптимизации запросов и настройке сервера.

Выбор оптимальных типов данных

MySQL поддерживает множество типов данных. Выбор правильного типа для хранения вашей информации критичен с точки зрения увеличения производительности. Следующие простые рекомендации помогут вам выбрать лучшее решение вне зависимости от типа сохраняемых данных.

Меньше – обычно лучше

Нужно стараться использовать типы данных минимального размера, достаточного для их правильного хранения и представления. Меньшие по размеру типы данных обычно быстрее, поскольку занимают меньше места на диске, в памяти и в кэше процессора. Кроме того, для их обработки обычно требуется меньше процессорного времени.

Избегайте недооценки диапазона возможных значений данных, поскольку увеличение размерности типа данных во многих местах схемы может оказаться болезненным и длительным процессом. Если вы сомневаетесь, какой тип данных выбрать, отдайте предпочтение самому короткому при условии, что его размера хватит. (Если система не очень загружена, хранит не очень много значений или вы находитесь на раннем этапе процесса проектирования, то легко сможете изменить решение позже).

Просто значит хорошо

Для выполнения операций с более простыми типами данных обычно требуется меньше процессорного времени. Например, сравнение целых чисел дешевле сравнения символов, поскольку из-за различных кодировок и схем упорядочения (правил сортировки) сравнение символов усложняется. Вот два примера: следует хранить значения даты и времени во встроенных типах данных MySQL, а не в строках. Для IP-адресов имеет смысл использовать целочисленные типы данных. Мы еще вернемся к обсуждению этой темы.

При возможности избегайте значений NULL

Всюду, где это возможно, определяйте столбцы как `NOT NULL`. Очень часто в таблицах встречаются поля, допускающие хранение `NULL` (отсутствие значения), хотя приложению это совершенно не нужно, – просто потому, что такой режим выбирается по умолчанию. Однако не объявляйте столбец как `NOT NULL`, если у хранящихся в нем данных могут отсутствовать значения.

Для MySQL оптимизация запросов, содержащих допускающие `NULL` столбцы, вызывает дополнительные сложности, поскольку из-за них усложняются индексы, статистика индексов и сравнение значений. Столбец, допускающий `NULL`, занимает больше места на диске и требует специальной обработки внутри MySQL. Когда такой столбец проиндексирован, ему требуется дополнительный байт для каждой записи, а в MyISAM даже может возникнуть ситуация, когда при-

дется преобразовать индекс фиксированного размера (например, индекс по одному целочисленному столбцу) в индекс переменного размера.

Даже когда требуется представить в таблице факт отсутствия значения, можно обойтись без использования NULL. Вместо этого иногда можно использовать нуль, специальное значение или пустую строку.

Повышение производительности в результате замены столбцов NULL на NOT NULL обычно невелико, так что не делайте их поиск и изменение в существующих схемах приоритетом, если не уверены, что именно они вызывают проблемы. Однако если вы планируете индексировать столбцы, по возможности определяйте их как NOT NULL.

Первым шагом в принятии решения, какой тип данных использовать для столбца, является определение общего класса типов: числовые, строковые, временные и т. п. Обычно никаких сложностей при этом не возникает, однако бывают ситуации, когда выбор неочевиден.

Следующим шагом является выбор конкретного типа. Многие типы данных MySQL позволяют хранить данные одного и тот же вида, но с разным диапазоном значений, точностью или требуемым физическим пространством (на диске или в памяти). Некоторые типы обладают специальным поведением или свойствами.

Например, в столбцах DATETIME и TIMESTAMP можно хранить один и тот же тип данных: дату и время, с точностью до секунды. Однако тип TIMESTAMP требует вдвое меньше места, позволяет работать с часовыми поясами и обладает специальными средствами автоматического обновления. С другой стороны, диапазон допустимых значений для него намного уже.

Здесь мы обсудим основные типы данных. В целях совместимости MySQL поддерживает различные псевдонимы, например INTEGER, BOOL и NUMERIC. Все это – именно псевдонимы одного и того же типа данных. Данный факт может сбить с толку, но не оказывает влияния на производительность.

Целые числа

Существуют два типа чисел: целые и вещественные (с дробной частью). Для хранения целых чисел используйте один из следующих целочисленных типов данных: TINYINT, SMALLINT, MEDIUMINT, INT или BIGINT. Их размеры соответственно равны 8, 16, 24, 32 и 64 бита. Целочисленный тип данных длиной N бит позволяет хранить значения от $-2^{(N-1)}$ до $2^{(N-1)}-1$.

Целые типы данных могут иметь необязательный атрибут UNSIGNED, запрещающий отрицательные значения и приблизительно вдвое увеличивающий верхний предел положительных значений. Например, тип TINYINT UNSIGNED позволяет хранить значения от 0 до 255, а не от –128 до 127.

Знаковые и беззнаковые типы требуют одинакового пространства и обладают одинаковой производительностью, так что используйте тот тип, который больше подходит для диапазона ваших данных.

Ваш выбор определяет то, как СУБД MySQL *хранит* данные, в памяти или на диске. Однако для *вычислений* с целыми числами обычно используются 64-разрядные целые типа `BIGINT`, даже на машинах с 32-разрядной архитектурой (исключение составляют некоторые агрегатные функции, использующие для вычислений тип `DECIMAL` или `DOUBLE`).

СУБД MySQL позволяет указывать для целых чисел «размер», например `INT(11)`. Для большинства приложений это не имеет значения: диапазон возможных значений этим не ограничивается. Однако данный параметр говорит некоторым интерактивным инструментам MySQL (например, клиенту командной строки), сколько позиций необходимо зарезервировать для вывода числа. С точки зрения хранения и вычисления `INT(1)` и `INT(20)` идентичны.

Подсистема хранения данных Falcon отличается от других подсистем, предоставляемых компанией MySQL AB, тем, что сохраняет целые числа в своем собственном внутреннем формате. Пользователь не может управлять реальным размером хранимых данных. Подсистемы хранения сторонних производителей, например Brighthouse, также имеют собственные форматы хранения и схемы сжатия.

Вещественные числа

Вещественные числа – это числа, имеющие дробную часть. Однако в типе данных `DECIMAL` также можно хранить большие числа, не помещающиеся в типе `BIGINT`. MySQL поддерживает как «точные» (exact), так и «неточные» (inexact) типы.

Типы `FLOAT` и `DOUBLE` допускают приближенные математические вычисления с плавающей точкой. Если вам нужно точно знать, как вычисляются результаты с плавающей точкой, изучите, как это реализовано на имеющейся у вас платформе.

Тип `DECIMAL` предназначен для хранения точных дробных чисел. В MySQL версии 5.0 и более поздних `DECIMAL` обеспечивает точные вычисления. MySQL 4.1 и более ранние версии для работы с `DECIMAL` использовали вычисления с плавающей точкой, которые иногда давали странные результаты из-за потери точности. В этих версиях MySQL тип `DECIMAL` предназначался только для хранения.

Начиная с версии MySQL 5.0 математические операции с типом `DECIMAL` реализуются самим сервером баз данных, поскольку процессоры не поддерживают такие вычисления непосредственно. Операции с плавающей точкой выполняются несколько быстрее, чем точные вычисле-

ния с DECIMAL, так как процессор выполняет их естественным для него образом.

Как типы с плавающей запятой, так и тип DECIMAL позволяют задавать требуемую точность. Для столбца типа DECIMAL вы можете указать максимально разрешенное количество цифр до и после десятичной запятой. Это влияет на объем пространства, требуемого для хранения данных столбца. MySQL 5.0 и более новые версии упаковывают цифры в двоичную строку (девять цифр занимают четыре байта). Например, DECIMAL(18, 9) будет хранить девять цифр с каждой стороны десятичной точки, используя в общей сложности девять байтов: четыре для цифр перед десятичным разделителем, один для самой десятичной точки и четыре для цифр после нее.

Число типа DECIMAL в MySQL 5.0 и более новых версиях может содержать до 65 цифр. Более ранние версии MySQL имели предел 254 цифры и хранили значения в виде неупакованных строк (один байт на цифру). Однако эти версии СУБД на деле не умели использовать такие большие числа в вычислениях, поскольку тип DECIMAL был просто форматом хранения. При выполнении каких-либо операций значения DECIMAL преобразовывались в тип DOUBLE.

Существуют способы указать желаемую точность для столбца чисел с плавающей точкой так, что MySQL незаметно для пользователя выберет другой тип данных или будет округлять значения при сохранении. Эти спецификаторы точности нестандартны, поэтому мы рекомендуем задавать желаемый тип, но не точность.

Типы с плавающей точкой обычно используют для хранения одного и того же диапазона значений меньше пространства, чем тип DECIMAL. Столбец типа FLOAT задействует всего лишь четыре байта. Тип DOUBLE требует восемь байтов и имеет большую точность и больший диапазон значений. Как и в случае целых чисел, вы выбираете тип только для хранения. Для вычислений с плавающей точкой MySQL использует тип DOUBLE.

Ввиду дополнительных требований к пространству и стоимости вычислений тип DECIMAL стоит использовать только тогда, когда нужны точные результаты при вычислениях с дробными числами, – например, при хранении финансовых данных.

Строковые типы

MySQL поддерживает большой диапазон строковых типов данных с различными вариациями. Эти типы претерпели значительные изменения в версиях 4.1 и 5.0, что сделало их еще более сложными. Начиная с версии MySQL 4.1, каждый строковый столбец может иметь свою собственную кодировку и соответствующую *схему упорядочения* (эта тема подробнее обсуждается в главе 5). Данное обстоятельство может значительно повлиять на производительность.

Типы VARCHAR и CHAR

Двумя основными строковыми типами являются VARCHAR и CHAR, предназначенные для хранения символьных значений. К сожалению, весьма непросто объяснить, как эти значения хранятся в памяти и на диске, поскольку конкретная реализация зависит от выбранной подсистемы хранения (например, Falcon использует собственные форматы хранения почти для каждого типа данных). Мы будем исходить из того, что вы применяете InnoDB или MyISAM. В противном случае вам следует прочитать документацию по используемой подсистеме хранения.

Давайте посмотрим, как обычно записываются на диск значения VARCHAR и CHAR. Имейте в виду, что подсистема хранения может представлять CHAR или VARCHAR в памяти не так, как на диске, и что сервер может преобразовывать данные в другой формат, когда извлекает их из подсистемы хранения. Вот общее сравнение этих двух типов:

VARCHAR

Тип VARCHAR хранит символьные строки переменной длины и является наиболее общим строковым типом данных. Строки этого типа могут занимать меньше места, чем строки фиксированной длины. Происходит это потому, что в VARCHAR используется лишь столько места, сколько действительно необходимо (например, для хранения более коротких строк требуется меньше пространства, чем в случае CHAR). Исключением являются таблицы типа MyISAM, созданные с параметром ROW_FORMAT=FIXED, когда для каждой строки на диске отводится область фиксированного размера, поэтому место может расходоваться впустую.

В типе VARCHAR используется один или два дополнительных байта для хранения длины строки: один байт, если максимальная длина строки в столбце не превышает 255 байт, и два байта в случае более длинных строк. В предположении, что используется кодировка *latin1*, тип VARCHAR(10) может занимать до 11 байт. Тип VARCHAR(1000) занимает до 1002 байт, поскольку в данном случае для хранения информации о длине строки требуется два байта.

VARCHAR увеличивает производительность за счет меньшего потребления места на диске. Однако поскольку строки имеют переменную длину, они способны увеличиваться при обновлении, что вызывает дополнительную работу. Если строка становится длиннее и больше не помещается в ранее отведенное для нее место, то ее дальнейшее поведение зависит от подсистемы хранения. Например, MyISAM может фрагментировать строку, а InnoDB, возможно, придется расщепить страницу. Другие подсистемы хранения могут вообще не обновлять данные в месте их хранения.

Обычно имеет смысл использовать тип VARCHAR при соблюдении хотя бы одного из следующих условий: максимальная длина строки

в столбце значительно больше средней; обновление поля выполняется редко, так что фрагментация не представляет проблемы; либо используется сложная кодировка, например UTF-8, в которой для хранения одного символа используется переменное количество байтов.

Начиная с версии MySQL 5.0, при сохранении и извлечении текстовых строк MySQL сохраняет пробелы в конце строки. В версиях до 4.1 включительно MySQL удаляет пробелы в конце строки.

CHAR

Тип CHAR имеет фиксированную длину: MySQL всегда выделяет место для указанного количества символов. При сохранении значения CHAR MySQL удаляет все пробелы в конце строки[1] (это справедливо также для типа VARCHAR в MySQL 4.1 и более ранних версий – CHAR и VARCHAR были логически идентичны и отличались только форматом хранения). Для удобства сравнения значения дополняются пробелами (по сути сравнение происходит без учета пробелов в конце строки).

Тип CHAR полезен, когда требуется сохранять очень короткие строки или все значения имеют приблизительно одинаковую длину. Например, CHAR является хорошим выбором для хранения MD5-сверток паролей пользователей, которые всегда имеют одинаковую длину. Тип CHAR также имеет преимущество над VARCHAR для часто меняющихся данных, поскольку строка фиксированной длины не подвержена фрагментации. В случае очень коротких столбцов тип CHAR также эффективнее, чем VARCHAR. Если тип CHAR(1) применяется для хранения значений Y и N, то в однобайтовой кодировке[2] он займет только один байт, тогда как для типа VARCHAR(1) потребуется два байта из-за наличия дополнительного байта длины строки.

Это поведение может несколько сбивать с толку, поэтому приведем пример. Сначала мы создали таблицу с одним столбцом типа CHAR(10) и сохранили в нем какие-то значения:

```
mysql> CREATE TABLE char_test( char_col CHAR(10));
mysql> INSERT INTO char_test(char_col) VALUES
    -> ('string1'), (' string2'), ('string3 ');
```

При извлечении значений пробелы в конце строки удаляются:

```
mysql> SELECT CONCAT("'", char_col, "'") FROM char_test;
+----------------------------+
| CONCAT("'", char_col, "'") |
+----------------------------+
```

[1] При сохранении данных в поле типа CHAR MySQL автоматически дополняет значение пробелами до указанной размерности, а в момент извлечения, наоборот, эти пробелы удаляются

[2] Не забывайте, что длина указывается в символах, а не в байтах. Для многобайтовых кодировок может потребоваться больше одного байта на символ.

```
| 'string1'                  |
| '  string2'                |
| 'string3'                  |
+----------------------------+
```

Если мы сохраним те же значения в столбце типа VARCHAR(10), то получим после извлечения следующий результат:

```
mysql> SELECT CONCAT("'", varchar_col, "'") FROM varchar_test;
+------------------------------+
| CONCAT("'", varchar_col, "'") |
+------------------------------+
| 'string1'                    |
| ' string2'                   |
| 'string3 '                   |
+------------------------------+
```

Как именно записываются данные, зависит от подсистемы хранения, и не все подсистемы обрабатывают значения фиксированной и переменной длины одинаково. В подсистеме Memory используются строки фиксированной длины, поэтому ей приходится выделять максимально возможное место для каждого значения, даже если это поле переменной длины. С другой стороны, в подсистеме Falcon используются столбцы переменной длины даже для полей типа CHAR. Однако поведение при дополнении и удалении пробелов одинаково для всех подсистем хранения, поскольку эту работу выполняет сам сервер MySQL.

Родственными типами для CHAR и VARCHAR являются BINARY и VARBINARY, предназначенные для хранения двоичных строк. Двоичные строки очень похожи на обычные, но вместо символов в них содержатся байты. Метод дополнения пробелами также отличается: MySQL добавляет

Щедрость не всегда разумна

Сохранение значения 'hello' требует одинакового пространства и в столбце типа VARCHAR(5), и в столбце типа VARCHAR(200). Есть ли преимущество в использовании более короткого столбца?

Оказывается, преимущество есть, и большое. Для столбца большей размерности может потребоваться намного больше памяти, поскольку MySQL часто выделяет для внутреннего хранения значений участки памяти фиксированного размера. Это особенно плохо для сортировки или операций, использующих временные таблицы в памяти. То же самое происходит при файловых сортировках, использующих временные таблицы на диске.

Наилучшей стратегией является выделение такого объема памяти, который действительно нужен.

в строки типа BINARY нулевой байт \0 вместо пробелов и не удаляет дополненные байты при извлечении[1].

Эти типы полезны, когда нужно сохранять двоичные данные, и вы хотите, чтобы MySQL сравнивал значение как байты, а не как символы. Отличием побайтового сравнения является не только различие в обработке строк разного регистра . MySQL сравнивает строки BINARY побайтно в соответствии с числовым значением каждого байта. В результате двоичное сравнение может оказаться значительно проще, а значит, и быстрее символьного.

Типы BLOB и TEXT

Строковые типы BLOB и TEXT предназначены для хранения больших объемов двоичных или символьных данных соответственно.

Фактически это семейства типов данных: к символьным относятся типы TINYTEXT, SMALLTEXT, TEXT, MEDIUMTEXT и LONGTEXT, к двоичным – TINYBLOB, SMALLBLOB, BLOB, MEDIUMBLOB и LONGBLOB. BLOB является синонимом для SMALLBLOB, а TEXT – синонимом для SMALLTEXT.

В отличие от всех остальных типов данных, MySQL обрабатывает значения BLOB и TEXT как отдельные объекты. Подсистемы хранения зачастую взаимодействуют с ними особым образом; InnoDB может помещать их в отдельную «внешнюю» область хранения, если они имеют большой размер. Каждому значению такого типа требуется от одного до четырех байтов в самой строке и достаточное место во внешнем хранилище для хранения фактического значения.

Единственное различие между семействами BLOB и TEXT заключается в том, что типы BLOB хранят двоичные данные без учета схемы упорядочения и кодировки, а с типами TEXT ассоциированы схемы упорядочения и кодировка.

СУБД MySQL сортирует столбцы BLOB и TEXT иначе, чем столбцы других типов: вместо сортировки строк по всей длине хранимых данных, она сортирует только по первым max_sort_length байтам. Если нужна сортировка только по нескольким первым символам, то можно либо уменьшить значение серверной переменной max_sort_length, либо использовать конструкцию ORDER BY SUBSTRING(*column, length*).

MySQL не может индексировать данные этих типов по полной длине и не может использовать для сортировки индексы (подробнее см. ниже в этой главе).

---

[1] Будьте осторожны с типом BINARY, если значение после извлечения должно оставаться неизменным. **MySQL дополняет его до требуемой длины нулевыми байтами.**

Как избежать создания временных таблиц на диске

Поскольку подсистема хранения Memory не поддерживает типы BLOB и TEXT, для запросов, в которых используются столбцы такого типа и которым нужна неявная временная таблица, придется использовать временные таблицы MyISAM на диске, даже если речь идет всего о нескольких строках. Это может привести к серьезной потере производительности. Даже если вы настроите в MySQL хранение временных таблиц на виртуальном диске, все равно потребуется много дорогостоящих вызовов операционной системы (подсистема хранения Maria должна смягчить эту проблему путем кэширования в памяти всего, а не только индексов).

Лучше всего не использовать типы BLOB и TEXT, если можно без них обойтись. Если же избежать этого не удается, можно использовать конструкцию ORDER BY SUBSTRING(*column, length*) для преобразования значений в символьные строки, которые уже могут храниться во временных таблицах в памяти. Только выбирайте достаточно короткие подстроки, чтобы временная таблица не вырастала до объемов, превышающих значения переменных max_heap_table_size или tmp_table_size, поскольку тогда MySQL преобразует таблицу в таблицу MyISAM на диске.

Если столбец Extra в распечатке команды EXPLAIN содержит слова «Using temporary», значит, для запроса использована неявная временная таблица.

Использование типа ENUM вместо строкового типа

Иногда вместо обычных строковых типов можно использовать тип ENUM. В столбце типа ENUM можно хранить до 65 535 различных строковых значений. MySQL сохраняет их очень компактно, упаковывая в один или два байта в зависимости от количества значений в списке. MySQL воспринимает каждое значение как целое число, представляющее позицию значения в списке значений поля, и отдельно хранит в *frm*-файле «справочную таблицу», определяющую соответствие между числом и строкой. Вот пример:

```
mysql> CREATE TABLE enum_test(
    ->   e ENUM('fish', 'apple', 'dog') NOT NULL
    -> );
mysql> INSERT INTO enum_test(e) VALUES('fish'), ('dog'), ('apple');
```

Во всех трех строках таблицы в действительности хранятся целые числа, а не строки. Убедиться в двойственной природе значений можно, если извлечь их в числовом контексте:

```
mysql> SELECT e + 0 FROM enum_test;
+-------+
| e + 0 |
+-------+
|     1 |
|     3 |
|     2 |
+-------+
```

Эта двойственность может привести к путанице, если указать в качестве констант столбца ENUM числа, например ENUM('1', '2', '3'). Мы не рекомендуем так поступать.

Другим сюрпризом является то, что поля типа ENUM сортируются по внутренним целочисленным значениям, а не по самим строкам:

```
mysql> SELECT e FROM enum_test ORDER BY e;
+-------+
| e     |
+-------+
| fish  |
| apple |
| dog   |
+-------+
```

Обойти это неудобство можно, введя значения для столбца ENUM в желаемом порядке сортировки. Можно также использовать функцию FIELD() с целью принудительного задания порядка сортировки в запросе, но это не позволит MySQL использовать для сортировки индекс:

```
mysql> SELECT e FROM enum_test ORDER BY FIELD(e, 'apple', 'dog', 'fish');
+-------+
| e     |
+-------+
| apple |
| dog   |
| fish  |
+-------+
```

Главным недостатком столбцов типа ENUM является то, что список строк фиксирован, а для их добавления или удаления необходимо использовать команду ALTER TABLE. Таким образом, если предполагается изменение списка возможных значений в будущем, то обращение к типу ENUM для представления строк может быть не такой уж и хорошей идеей. MySQL использует тип ENUM в своих собственных таблицах привилегий с целью хранения значений Y и N.

Поскольку MySQL сохраняет каждое значение как целое число и вынуждена выполнять просмотр таблицы соответствий для преобразования числа в строковое представление, то со столбцами типа ENUM связаны некоторые накладные расходы. Обычно это компенсируется их малым размером, но не всегда. В частности, соединение столбца типа

CHAR или VARCHAR со столбцом типа ENUM может оказаться медленнее, чем с другим столбцом типа CHAR или VARCHAR.

Для иллюстрации мы протестировали, насколько быстро MySQL выполняет такое соединение с таблицей, в одном из наших приложений. В таблице определен довольно длинный первичный ключ:

```
CREATE TABLE webservicecalls (
    day date NOT NULL,
    account smallint NOT NULL,
    service varchar(10) NOT NULL,
    method varchar(50) NOT NULL,
    calls int NOT NULL,
    items int NOT NULL,
    time float NOT NULL,
    cost decimal(9,5) NOT NULL,
    updated datetime,
    PRIMARY KEY (day, account, service, method)
) ENGINE=InnoDB;
```

Таблица содержит около 110 000 строк и имеет объем порядка 10 Мбайт, поэтому она целиком помещается в памяти. Столбец service содержит 5 различных значений со средней длиной 4 символа, а столбец method — 71 значение со средней длиной 20 символов.

Мы сделали копию этой таблицы и преобразовали столбцы service и method в тип ENUM следующим образом:

```
CREATE TABLE webservicecalls_enum (
    ... опущено ...
    service ENUM(...значения опущены...) NOT NULL,
    method ENUM(...значения опущены...) NOT NULL,
    ... опущено ...
) ENGINE=InnoDB;
```

Затем мы измерили производительность соединения таблиц по столбцам первичного ключа. Вот использованный нами запрос:

```
mysql> SELECT SQL_NO_CACHE COUNT(*)
    -> FROM webservicecalls
    ->    JOIN webservicecalls USING(day, account, service, method);
```

Мы варьировали этот запрос, соединяя столбцы типа VARCHAR и ENUM в различных комбинациях. Результаты показаны в табл. 3.1.

Таблица 3.1. Скорость соединения столбцов типа VARCHAR и ENUM

Тест	Запросов в секунду
Соединение VARCHAR с VARCHAR	2.6
Соединение VARCHAR с ENUM	1.7
Соединение ENUM с VARCHAR	1.8
Соединение ENUM с ENUM	3.5

Соединение становится быстрее после преобразования всех столбцов в тип ENUM, но соединение столбцов типа ENUM со столбцами типа VARCHAR происходит медленнее. В данном случае преобразование имеет смысл, если нет необходимости соединять эти столбцы со столбцами типа VARCHAR.

Однако имеется и другая польза от преобразования столбцов: команда SHOW TABLE STATUS в столбце Data_length показывает, что после преобразования двух столбцов к типу ENUM таблица стала приблизительно на треть меньше. В некоторых случаях это может дать выигрыш, даже несмотря на необходимость соединения столбцов типа ENUM со столбцами типа VARCHAR. Кроме того, сам первичный ключ после преобразования уменьшается приблизительно вдвое. Поскольку это таблица InnoDB, то если в ней есть и другие индексы, уменьшение размера первичного ключа приведет к уменьшению их размера тоже. Мы объясним данный эффект ниже в этой главе.

Типы Date и Time

СУБД MySQL включает много различных типов значений даты и времени (темпоральных типов), например YEAR и DATE. Минимальной единицей времени, которую может хранить MySQL, является одна секунда. Однако *вычисления* с темпоральными данными выполняются с точностью до одной микросекунды, и мы покажем, как обойти ограничение хранения.

Большинство темпоральных типов не имеет альтернатив, поэтому вопрос о том, какой лучше, не возникает. Нужно лишь решить, что делать, когда требуется сохранять и дату, и время. Для этих целей MySQL предлагает два очень похожих типа данных: DATETIME и TIMESTAMP. Большинству приложений подходят оба, но в некоторых случаях один работает лучше, чем другой. Давайте более подробно рассмотрим каждый из них:

DATETIME

Этот тип позволяет хранить значения в большом диапазоне, с 1001 до 9999 года, с точностью в одну секунду. Дата и время упаковываются в целое число в формате YYYYMMDDHHMMSS независимо от часового пояса. Под значение отводится восемь байт.

По умолчанию MySQL показывает данные типа DATETIME в точно определенном, допускающем сортировку формате: 2008-01-16 22:37:08. Этот способ представления даты и времени согласуется со стандартом ANSI.

TIMESTAMP

Как следует из названия, в типе TIMESTAMP хранится количество секунд, прошедших с полуночи первого января 1970 года (по гринвичскому времени) – так же, как во временной метке UNIX. Для хранения типа TIMESTAMP используется только четыре байта, поэтому он позволяет представить значительно меньший диапазон дат, чем тип DATETIME: с 1970 года до некоторой даты в 2038 году. В MySQL имеют-

ся функции FROM_UNIXTIME() и UNIX_TIMESTAMP(), служащие для преобразования временной метки UNIX в дату и наоборот.

В новых версиях MySQL значения типа TIMESTAMP форматируются точно так же, как значения DATETIME, но в старых версиях они отображаются без разделителей между составными частями. Это различие проявляется только при выводе, формат хранения значений TIMESTAMP одинаков во всех версиях MySQL.

Отображаемое значение типа TIMESTAMP зависит также от часового пояса. Часовой пояс определен для сервера MySQL, операционной системы и клиентского соединения. Таким образом, если в поле типа TIMESTAMP хранится значение 0, то для часового пояса Eastern Standard Time, отличающегося от гринвичского времени на пять часов, будет выведена строка 1969-12-31 19:00:00.

Тип TIMESTAMP имеет также специальные свойства, которых нет у типа DATETIME. По умолчанию, если вы не указали значение для столбца, MySQL вставляет в первый столбец типа TIMESTAMP текущее время[1]. Кроме того, по умолчанию MySQL также изменяет значение первого столбца типа TIMESTAMP при обновлении строки, если ему явно не присвоено значение в команде UPDATE. Вы можете настроить поведение при вставке и обновлении для каждого столбца типа TIMESTAMP. Наконец, столбцы типа TIMESTAMP по умолчанию создаются в режиме NOT NULL, в отличие от всех остальных типов данных.

Несмотря на эти особенности, мы рекомендуем пользоваться типом TIMESTAMP, если это возможно, поскольку с точки зрения занимаемого места на диске он гораздо эффективнее, чем DATETIME. Иногда временные метки UNIX сохраняют в виде целых чисел, но обычно это не дает никаких преимуществ. Поскольку такое представление часто неудобно, мы не советуем так поступать.

Что если нужно сохранять значение даты и времени с точностью большей, чем одна секунда? На данный момент в MySQL для этих целей нет соответствующего типа данных, но вы можете использовать свой собственный формат хранения, скажем, сохранять временную метку с микросекундной точностью в типе данных BIGINT либо воспользоваться типом DOUBLE и поместить дробную часть секунды после десятичной точки. Оба подхода вполне работоспособны.

Битовые типы данных

В MySQL есть несколько типов данных, использующих для компактного хранения значений отдельные биты. Все эти типы с технической

[1] Правила поведения типа TIMESTAMP сложны и неодинаковы в различных версиях MySQL, поэтому вам нужно убедиться, что это поведение соответствует вашим представлениям. Обычно имеет смысл после внесения изменений в столбцы TIMESTAMP проверить вывод команды SHOW CREATE TABLE.

точки зрения являются строковыми вне зависимости от внутреннего формата хранения и манипуляций:

`BIT`

До версии MySQL 5.0 ключевое слово `BIT` было просто синонимом `TINYINT`. Но начиная с версии MySQL 5.0, это совершенно иной тип данных с особыми характеристиками. Ниже мы обсудим его поведение.

Вы можете использовать столбец типа `BIT` для хранения одного или нескольких значений true/false в одном столбце. `BIT(1)` определяет поле, содержащее один бит, `BIT(2)` – два бита и т. д. Максимальная длина столбца типа `BIT` равна 64 битам.

Поведение типа `BIT` зависит от подсистемы хранения. MyISAM объединяет битовые столбцы, поэтому для хранения 17 отдельных столбцов типа `BIT` требуется только 17 бит (в предположении, что ни в одном из столбцов не разрешено значение `NULL`). При вычислении размера места для хранения MyISAM округлит это число до трех байтов. Другие подсистемы, например Memory и InnoDB, представляют каждый столбец как наименьший целочисленный тип, достаточно большой для размещения всех битов, поэтому сэкономить пространство не получится.

MySQL рассматривает `BIT` как строковый тип, а не числовой. Когда вы извлекаете значение типа `BIT(1)`, результатом является строка, но ее содержимое представляет собой двоичное значение 0 или 1, а не значение «0» или «1» в кодировке ASCII. Однако если вы извлечете значение в числовом контексте, результатом будет число, в которое преобразуется битовая строка. Имейте это в виду, когда захотите сравнить результат с другим значением. Например, если вы сохраните значение `b'00111001'` (являющееся двоичным эквивалентом числа 57) в столбце типа `BIT(8)` и затем извлечете его, вы получите строку, содержащую код символа 57. Это код символа в кодировке ASCII для цифры «9». Но в числовом контексте вы получите число 57:

```
mysql> CREATE TABLE bittest(a bit(8));
mysql> INSERT INTO bittest VALUES(b'00111001');
mysql> SELECT a, a + 0 FROM bittest;
+------+-------+
| a    | a + 0 |
+------+-------+
| 9    |    57 |
+------+-------+
```

Такое поведение может сбивать с толку, так что мы советуем использовать тип `BIT` с осторожностью. Для большинства приложений лучше его вообще избегать.

Если возникла необходимость сохранять значение true/false в одном бите, просто создайте столбец типа `CHAR(0)` с возможностью хранения

NULL. Подобный столбец может представить как отсутствие значения (NULL), так и значение нулевой длины (пустая строка).

SET

Если нужно сохранять много значений true/false, попробуйте объединить несколько столбцов в один столбец типа SET. В MySQL его внутренним представлением является упакованный битовый вектор, эффективно использующий пространство. MySQL содержит такие функции, как FIND_IN_SET() и FIELD(), которые можно легко использовать в запросах. Главным недостатком является стоимость изменения определения столбца: эта процедура выполняется с помощью команды ALTER TABLE, которая для больших таблиц обходится очень дорого (но ниже в этой главе мы рассмотрим обходной путь). Кроме того, в общем случае при поиске в столбцах типа SET не используются индексы.

Побитовые операции над целочисленными столбцами

Альтернативой типу SET является использование целого числа как упакованного набора битов. Например, вы можете поместить восемь бит в тип TINYINT и выполнять с ним побитовые операции. Для упрощения работы можно определить именованные константы для каждого бита в коде приложения.

Главным преимуществом такого подхода по сравнению с использованием типа SET является то, что вы можете изменить «нумерацию» представляемого поля без обращения к команде ALTER TABLE. Недостаток же заключается в том, что запросы труднее писать и понимать (что означает, если бит 5 установлен?). Некоторые люди легко ориентируются в побитовых операциях, но многие их не любят, поэтому использование данного метода является делом вкуса.

Примером применения битовых векторов является список управления доступом (ACL), в котором хранятся разрешения. Каждый бит или элемент SET представляет значение, например CAN_READ, CAN_WRITE или CAN_DELETE. Если вы используете столбец типа SET, вы позволяете MySQL сохранять в определении столбца соответствие между битами и значениями. Если же используется целочисленный столбец, то такое соответствие устанавливается в коде приложения. Вот как могут выглядеть запросы со столбцом типа SET:

```
mysql> CREATE TABLE acl (
    -> perms SET('CAN_READ', 'CAN_WRITE', 'CAN_DELETE') NOT NULL
    -> );
mysql> INSERT INTO acl(perms) VALUES ('CAN_READ,CAN_DELETE');
mysql> SELECT perms FROM acl WHERE FIND_IN_SET('CAN_READ', perms);
+---------------------+
| perms               |
+---------------------+
| CAN_READ,CAN_DELETE |
+---------------------+
```

При использовании целочисленного столбца этот пример записывается следующим образом:

```
mysql> SET @CAN_READ := 1 << 0,
    -> @CAN_WRITE := 1 << 1,
    -> @CAN_DELETE := 1 << 2;
mysql> CREATE TABLE acl (
    -> perms TINYINT UNSIGNED NOT NULL DEFAULT 0
    -> );
mysql> INSERT INTO acl(perms) VALUES(@CAN_READ + @CAN_DELETE);
mysql> SELECT perms FROM acl WHERE perms & @CAN_READ;
+-------+
| perms |
+-------+
|     5 |
+-------+
```

Мы задействовали для определения столбцов переменные, но вы можете использовать в своем коде и константы.

Выбор типа идентификатора

Выбор типа данных для столбца идентификатора имеет очень большое значение. Велика вероятность, что этот столбец будет сравниваться с другими значениями (например, в соединениях) и использоваться для поиска чаще, чем другие столбцы. Возможно также, что вы будете применять идентификаторы как внешние ключи в других таблицах, поэтому выбор типа столбца идентификатора, скорее всего, определит и типы столбцов в связанных таблицах (как мы уже демонстрировали в этой главе, имеет смысл использовать в связанных таблицах одни и те же типы данных, поскольку они, скорее всего, будут задействованы для соединения).

При выборе типа данных для столбца идентификатора нужно принимать во внимание не только тип хранения, но и то, как MySQL выполняет вычисления и сравнения с этим типом. Например, MySQL хранит типы `ENUM` и `SET` как целые числа, но при выполнении сравнения в строковом контексте преобразует их в строки.

Сделав выбор, убедитесь, что вы используете один и тот же тип во всех связанных таблицах. Типы должны совпадать в точности, включая такие свойства как `UNSIGNED`[1]. Смешение различных типов данных может

[1] При использовании подсистемы хранения **InnoDB вообще невозможно создать** внешний ключ, если типы данных не совпадают в точности. Сообщение об ошибке «ERROR 1005 (HY000): Can't create table» может сбивать с толку в зависимости от контекста, а вопросы на эту тему часто появляются в списках рассылок MySQL (как ни странно, разрешается создавать внешние ключи между столбцами `VARCHAR` разной длины).

вызвать проблемы с производительностью, и, даже если этого не произойдет, неявные преобразования типов в процессе сравнения нередко являются причиной труднонаходимых ошибок. Они могут проявиться значительно позже, когда вы уже давно забудете, что сравниваете данные разных типов.

Имеет смысл выбирать самый маленький размер поля, способный вместить требуемый диапазон значений, и при необходимости оставлять место для дальнейшего роста. Например, если есть столбец `state_id`, в котором хранятся названия штатов США, вам не нужны тысячи или миллионы значений, поэтому не используйте тип `INT`. Типа `TINYINT`, который на три байта короче, вполне достаточно. Если вы используете это значение как внешний ключ в других таблицах, три байта могут иметь большое значение.

Целые типы

Целые типы обычно лучше всего подходят для идентификаторов, поскольку они работают быстро и допускают автоматический инкремент (`AUTO_INCREMENT`).

`ENUM` *и* `SET`

Типы `ENUM` и `SET` обычно не годятся для идентификаторов, зато могут быть хороши для статических «таблиц определений», содержащих значения состояния или «типа». Столбцы `ENUM` и `SET` подходят для хранения такой информации, как состояние заказа, вид продукта или пол человека.

Например, если вы используете поле `ENUM` для определения вида продукта, вам может потребоваться справочная (lookup) таблица с первичным ключом по полю точно такого же типа `ENUM` (вы можете поместить в справочную таблицу столбцы с текстом описания, чтобы сгенерировать глоссарий или показать осмысленные названия в элементах выпадающего меню на веб-сайте). В этом случае можно использовать тип `ENUM` для идентификатора, но, как правило, этого лучше избегать.

Строковые типы

По возможности избегайте задания для идентификаторов строковых типов, поскольку они занимают много места и обычно обрабатываются медленнее, чем целочисленные типы. Особенно осторожным следует быть при использовании строковых идентификаторов в таблицах MyISAM. Подсистема хранения MyISAM по умолчанию применяет для строк упакованные индексы, поиск по которым значительно медленнее. В наших тестах мы обнаружили, что в процессе работы с упакованными индексами MyISAM производительность падает чуть ли не в шесть раз.

Следует также быть очень внимательными при работе со «случайными» строками, например сгенерированными функциями `MD5()`, `SHA1()` или `UUID()`. Созданные с их помощью значения распределены случай-

ным образом в большом пространстве, что может замедлить работу команд INSERT и некоторых типов SELECT[1]:

- Они замедляют запросы INSERT, поскольку вставленное значение должно быть помещено в случайное место в индексах. Это приводит к расщеплению страниц, произвольному доступу к диску и к фрагментации кластерного индекса в подсистемах хранения, которые его поддерживают.
- Они замедляют запросы SELECT, так как логически соседние строки оказываются разбросаны по всему диску и памяти.
- Случайные значения приводят к ухудшению работы кэша для запросов всех типов, поскольку нарушают принцип локальности ссылок, лежащий в основе работы кэша. Если весь набор данных одинаково «горячий», то не имеет смысла сохранять какие-то части информации в кэше, а если рабочее множество не помещается в памяти, то будут иметь место частые непопадания и вытеснения из кэша.

Если вы сохраняете значения UUID, то нужно удалить тире или, что еще лучше, преобразовать значения UUID в 16-байтные числа с помощью функции UNHEX() и сохранить их в столбце типа BINARY(16). Вы можете извлекать значения в шестнадцатеричном формате с помощью функции HEX().

Значения, сгенерированные с помощью функции UUID(), имеют характеристики, отличающиеся от характеристик значений, сгенерированных криптографическими хеш-функциями типа SHA1(): данные UUID распределены неравномерно и являются в некоторой степени последовательными. Однако они не настолько хороши, как монотонно увеличивающиеся целые числа.

Специальные типы данных

Для некоторых видов данных может не существовать удобного встроенного типа. Одним из таких примеров является временная метка с субсекундной точностью. Некоторые приемы сохранения подобных данных мы показали выше в этой главе.

Другим примером является IP-адрес. Многие используют для их хранения столбцы типа VARCHAR(15). Однако на деле IP-адрес является 32-битным беззнаковым целым числом, а не строкой. Запись с точками, разделяющими байты, предназначена только для удобства их восприятия человеком. Лучше хранить IP-адреса как беззнаковые целые числа. В MySQL имеются функции INET_ATON() и INET_NTOA() для преобра-

[1] С другой стороны, иногда для очень больших таблиц, в которые одновременно записывает данные множество клиентов, такие псевдослучайные значения могут помочь избежать «горячих мест».

зования между двумя представлениями. В будущих версиях MySQL может появиться специальный встроенный тип данных для хранения IP-адресов.

Остерегайтесь автоматически сгенерированных схем

Мы описали наиболее важные особенности типов данных (некоторые оказывают большее влияние на производительность, другие – меньшее), но еще не говорили о пороках автоматически сгенерированных схем.

Плохо написанные программы миграции схем и программы, автоматически генерирующие схемы, могут вызывать серьезные проблемы с производительностью. Некоторые приложения используют поля VARCHAR большой размерности для хранения любого типа данных, другие генерируют различные типы данных для столбцов, которые будут сравниваться при соединении. Изучите схему очень тщательно, если она была сгенерирована автоматически.

Системы объектно-реляционного отображения (Object-relational mapping – ORM) – еще один кошмар для желающих достичь высокой производительности. Некоторые из них позволяют сохранять любой тип данных в произвольном хранилище, а это обычно означает, что они не в состоянии использовать сильные стороны конкретного хранилища. Иногда они записывают каждое свойство объекта в отдельную строку и даже используют хронологический контроль версий, в результате чего у каждого свойства возникает сразу несколько версий!

Такой подход привлекателен для специалистов, предпочитающих работать в объектно-ориентированном стиле, не задумываясь о том, как данные хранятся. Однако приложения, «прячущие сложность от разработчиков», обычно плохо масштабируются. Мы советуем вам хорошо подумать, прежде чем променять производительность на удобство разработки. Всегда проводите тестирование на реалистичных наборах данных, чтобы не обнаружить проблемы с производительностью слишком поздно.

Основы индексирования

Индексы представляют собой структуры, которые помогают MySQL эффективно извлекать данные. Они критичны для достижения хорошей производительности, но многие часто забывают о них или плохо понимают их смысл, поэтому индексирование является главной причиной проблем с производительностью в реальных условиях. Вот почему мы

поместили этот материал в начальной части книги – даже раньше, чем обсуждение оптимизации запросов.

Важность индексов (именуемых в MySQL также ключами) увеличивается по мере роста объема данных. Небольшие, слабо загруженные базы зачастую могут удовлетворительно работать даже без правильно построенных индексов, но по мере роста объемов хранимой в базе информации производительность может упасть очень быстро.

Самый простой способ понять, как работает индекс в MySQL, – представить себе алфавитный указатель в книге. Чтобы выяснить, в какой части издания обсуждается конкретный вопрос, вы смотрите в алфавитный указатель и находите номер страницы, где упоминается термин.

MySQL использует индексы сходным образом. Она ищет значение в структурах данных индекса. Обнаружив соответствие, она может перейти к самой строке. Допустим, выполняется следующий запрос:

```
mysql> SELECT first_name FROM sakila.actor WHERE actor_id = 5;
```

По столбцу `actor_id` построен индекс, поэтому СУБД MySQL будет использовать его для поиска строк, в которых значением поля `actor_id` является 5. Другими словами, она производит поиск значений в индексе и возвращает все строки, содержащие указанное значение.

Индекс содержит данные из указанного столбца или нескольких столбцов. Если индекс построен по нескольким столбцам, то их порядок следования очень важен, поскольку MySQL может осуществлять поиск эффективно только по самой левой части ключа. Как вы увидите ниже, создание индекса по двум столбцам – это совсем не то же самое, что создание двух отдельных индексов по одному столбцу.

Типы индексов

Существует много типов индексов, каждый из которых лучше всего подходит для достижения той или иной цели. Индексы реализуются на уровне подсистем хранения, а не на уровне сервера. Таким образом, они не стандартизованы: в каждой подсистеме индексы работают немного по-разному, и далеко не все подсистемы допускают использование существующего разнообразия индексов. Даже если некоторый тип поддерживается в нескольких подсистемах хранения, внутренняя реализация может различаться.

Не забывая об этом, рассмотрим типы индексов, с которыми умеет взаимодействовать MySQL в настоящее время, отмечая их достоинства и недостатки.

B-Tree-индексы

Когда говорят об индексе без упоминания типа, обычно имеют в виду B-Tree индексы, в которых для хранения данных используется структу-

ра, называемая *B-tree*[1]. Большинство подсистем хранения в MySQL поддерживает этот тип. Исключение – подсистема Archive: в ней вообще отсутствовало индексирование до версии MySQL 5.1, когда появилась возможность строить индекс по одному столбцу типа AUTO_INCREMENT.

Мы используем термин «B-tree» для этих индексов потому, что именно так MySQL называет их в CREATE TABLE и других командах. Однако на внутреннем уровне подсистемы хранения могут использовать совершенно другие структуры данных. Например, в подсистеме NDB Cluster для таких индексов применяется T-tree, хоть называются они по-прежнему BTREE.

Подсистемы хранения представляют B-Tree-индексы на диске по-разному, и это может влиять на производительность. Например, в MyISAM используется техника сжатия префикса, позволяющая уменьшить размер индекса, а InnoDB не сжимает индексы, поскольку это лишило бы ее возможности выполнять некоторые оптимизации. Кроме того, индексы MyISAM ссылаются на индексированные строки по их физическому адресу на диске, а InnoDB – по значениям первичного ключа. Каждый вариант имеет свои достоинства и недостатки.

Общая идея B-дерева заключается в том, что значения хранятся по порядку, и все листовые страницы находятся на одинаковом расстоянии от корня. На рис. 3.1 показано абстрактное представление B-Tree-индекса, которое приблизительно соответствует тому, как работают индексы InnoDB (InnoDB использует структуру данных B+Tree). В MyISAM структура другая, но принципы те же.

B-Tree-индекс ускоряет доступ к данным, поскольку подсистеме хранения не нужно сканировать всю таблицу для поиска нужной информации. Вместо этого она начинает с корневого узла (не показанного на этом рисунке). В корневом узле имеется массив указателей на дочерние узлы, и подсистема хранения переходит по этим указателям. Чтобы найти подходящий указатель, она просматривает значения в узловых страницах, которые определяют верхнюю и нижнюю границы значений в дочерних узлах. В конечном итоге подсистема хранения либо определяет, что искомое значение не существует, либо благополучно достигает листовой страницы.

Листовые страницы представляют собой особый случай, так как в них находятся указатели на индексированные данные, а не на другие страницы. (В разных подсистемах хранения применяются различные указатели на данные). На рисунке выше показана только одна узловая стра-

[1] Во многих подсистемах хранения на самом деле используются индексы типа B+Tree, в которых каждый листовой узел содержит указатель на следующий для ускорения обхода дерева по диапазону значений. Более подробное описание индексов со структурой B-дерева можно найти в литературе по информатике.

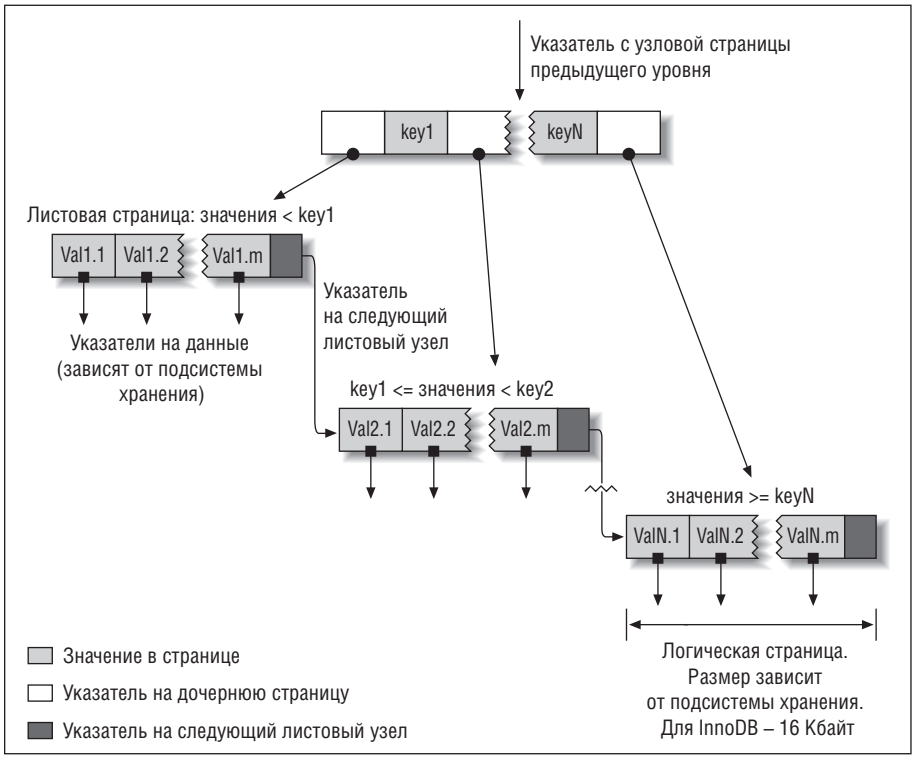


Рис. 3.1. *Индекс, построенный на основе структуры B-tree (технически B+Tree)*

ница и соответствующие ей листовые страницы, но между корнем и листьями может быть много уровней узловых страниц. Глубина дерева зависит от того, насколько велика таблица.

Поскольку в B-Tree индексах индексированные столбцы хранятся в упорядоченном виде, то они полезны для поиска по диапазону данных. Например, при спуске вниз по дереву индекса, построенного по текстовому полю, значения перебираются в алфавитном порядке, поэтому поиск «всех лиц, чьи фамилии начинаются с буквы от В до Д» оказывается эффективным.

Предположим, у нас есть следующая таблица:

```
CREATE TABLE People (
    last_name varchar(50) not null,
    first_name varchar(50) not null,
    dob date not null,
    gender enum('m', 'f') not null,
    key(last_name, first_name, dob)
);
```

Индекс будет содержать значения из столбцов `last_name`, `first_name` и `dob` для каждой строки в таблице. Рис. 3.2 иллюстрирует порядок расположения данных в индексе.

Обратите внимание, что значения в индексе упорядочены в порядке следования столбцов, который указан в команде `CREATE TABLE`. Посмотрите на последние два элемента: есть два человека с одинаковыми фамилией и именем, но разными датами рождения – они отсортированы именно по этому параметру.

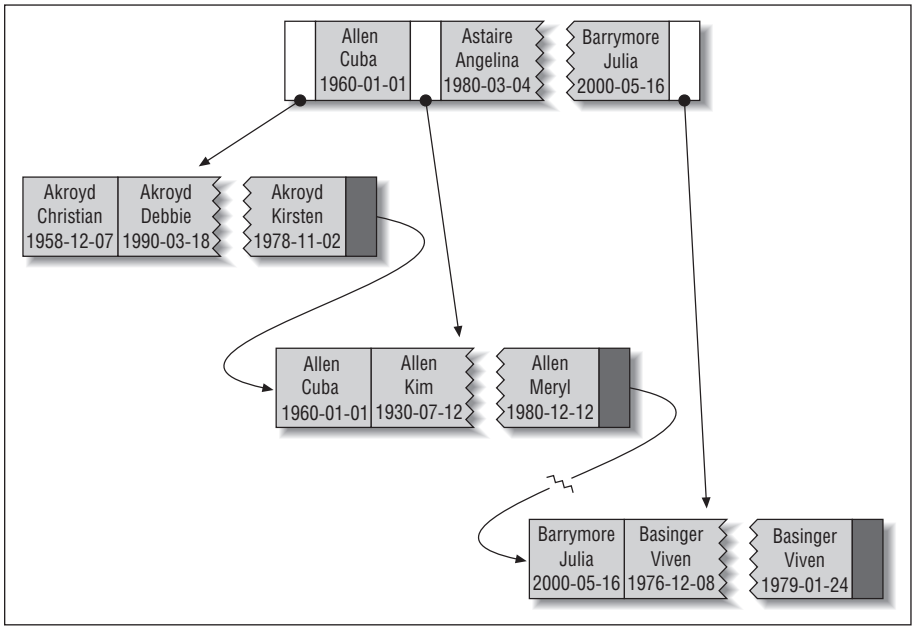


Рис. 3.2. *Пример расположения записей в индексе B-tree (технически B+-tree)*

Типы запросов, в которых может использоваться B-Tree-индекс

B-Tree-индексы хорошо работают при поиске по полному значению ключа, по диапазону ключей или по префиксу ключа. Они полезны только в том случае, когда в процессе поиска используется самый левый префикс ключа[1]. Индекс, показанный нами в предыдущем разделе, будет полезен для запросов следующих типов:

[1] Это особенность СУБД **MySQL, точнее, ее версий. В других СУБД можно выполнять** поиск не только по начальной части ключа, хотя обычно использование префикса эффективнее. Такая возможность, вероятно, должна появиться в **MySQL в будущем. Способы обойти данное ограничение мы покажем** ниже в этой главе.

Поиск по полному значению

При поиске с полным значением ключа задаются критерии для всех столбцов, по которым построен индекс. Например, индекс позволит найти человека по имени Cuba Allen, родившегося 1 января 1960.

Поиск по самому левому префиксу

Индекс позволит найти всех людей с фамилией Allen. В этом случае используется только первый столбец индекса.

Поиск по префиксу столбца

Вы можете искать соответствие по началу значения столбца. Рассматриваемый индекс позволит найти всех людей, чьи фамилии начинаются с буквы J. В этом случае используется только первый столбец индекса.

Поиск по диапазону значений

Индекс позволит найти всех людей с фамилиями, начиная с Allen и кончая Barrymore. В этом случае также используется только первый столбец индекса.

Поиск по полному совпадению одной части и диапазону в другой части

Индекс позволит найти всех людей с фамилией Allen, чьи имена начинаются с буквы K (Kim, Karl и т. п.). Полное совпадение со столбцом `last_name` и поиск по диапазону значений столбца `first_name`.

Запросы только по индексу

B-Tree-индексы могут, как правило, поддерживать запросы только по индексу, когда обращение производится именно к индексу, а не к самой строке. Мы обсудим такую оптимизацию в разделе «Покрывающие индексы» ниже в этой главе на стр. 163.

Поскольку узлы дерева отсортированы, их можно использовать как для поиска значений, так и в запросах с фразой `ORDER BY` (возврат отсортированных строк). В общем случае, если B-Tree индекс позволяет найти строку по определенному критерию, то его можно использовать и для сортировки строк по тому же критерию. Вот почему наш индекс будет полезен для запросов с фразой `ORDER BY`, в которой сортировка соответствует вышеперечисленным видам поиска.

У B-Tree-индексов есть некоторые ограничения:

- Они бесполезны, если в критерии поиска указана не самая левая часть ключа индекса. Например, рассматриваемый индекс не поможет найти людей с именем Bill или всех людей с определенной датой рождения, поскольку эти столбцы не являются самыми левыми в индексе. Аналогично, такой индекс непригоден для поиска людей, чьи фамилии заканчиваются на конкретную букву.
- Нельзя пропускать столбцы индекса. То есть, невозможно найти всех людей, имеющих фамилию Smith и родившихся в конкретный день.

Если не указать значение столбца first_name, то MySQL сможет использовать только первый столбец индекса.

- Подсистема хранения не может оптимизировать поиск по столбцам, находящимся правее первого столбца, по которому осуществляется поиск в заданном диапазоне. Например, для запроса с условием WHERE last_name="Smith" AND first_name LIKE 'J%' AND dob='1976-12-23' будут использованы только первые два столбца индекса, поскольку LIKE задает диапазон (однако сервер может использовать оставшиеся столбцы для других целей). Для столбца, имеющего ограниченное количество значений, вы можете применить обходной маневр, указав условия равенства вместо условия на диапазон. Ниже в этой главе мы приведем подробные примеры на эту тему.

Теперь вы знаете, почему так важен порядок столбцов: все эти ограничения так или иначе связаны именно с ним. Для высокопроизводительных приложений может потребоваться создать индексы с одними и теми же столбцами в различном порядке.

Некоторые из указанных ограничений связаны не с самими B-Tree-индексами, а являются следствием того, как оптимизатор запросов MySQL и подсистемы хранения используют индексы. Часть подобных ограничений может быть устранена в будущем.

Хеш-индексы

Хеш-индекс строится на основе хеш-таблицы и полезен только для точного поиска с указанием всех столбцов индекса[1]. Для каждой строки подсистема хранения вычисляет *хеш-код* индексированных столбцов – сравнительно короткое значение, которое, скорее всего, будет различно для строк с разными значениями ключей. В индексе хранятся хеш-коды и указатели на соответствующие строки.

В MySQL только подсистема хранения Memory поддерживает явные хеш-индексы. Этот тип индекса принимается по умолчанию для таблиц типа Memory, хотя над ними можно строить также и B-Tree-индексы. Подсистема Memory поддерживает неуникальные хеш-индексы, что в мире баз данных является необычным. Если для нескольких строк хеш-код одинаков, то в индексе будет храниться связанный список указателей на эти строки.

Рассмотрим пример. Пусть имеется таблица со следующей структурой:

```
CREATE TABLE testhash (
    fname VARCHAR(50) NOT NULL,
    lname VARCHAR(50) NOT NULL,
    KEY USING HASH(fname)
) ENGINE=MEMORY;
```

[1] Более подробную информацию о хеш-таблицах вы можете найти в специальной научной литературе.

Таблица содержит такие данные:

```
mysql> SELECT * FROM testhash;
+--------+-----------+
| fname  | lname     |
+--------+-----------+
| Arjen  | Lentz     |
| Baron  | Schwartz  |
| Peter  | Zaitsev   |
| Vadim  | Tkachenko |
+--------+-----------+
```

Предположим далее, что для индексирования используется воображаемая хеш-функция `f()`, которая возвращает следующие значения (это просто примеры, а не реальные значения):

```
f('Arjen') = 2323
f('Baron') = 7437
f('Peter') = 8784
f('Vadim') = 2458
```

Структура данных индекса будет выглядеть подобным образом:

Ячейка	Значение
2323	Указатель на строку 1
2458	Указатель на строку 4
7437	Указатель на строку 2
8784	Указатель на строку 3

Обратите внимание, что ячейки упорядочены, а строки – нет. При выполнении следующего запроса

```
mysql> SELECT lname FROM testhash WHERE fname='Peter';
```

MySQL вычислит хеш-код значения `'Peter'` и использует его для поиска указателя в индексе. Поскольку `f('Peter')`= 8784, MySQL будет искать в индексе значение 8784 и найдет указатель на строку 3. Конечным шагом будет сравнение значения в строке 3 с `'Peter'`, с целью убедиться, что это действительно искомая строка.

Поскольку в хеш-индексе хранятся только короткие хеш-коды, то такие индексы очень компактны. Длина хеш-кода не зависит от типа индексируемого столбца – хеш-индекс по столбцу `TINYINT` будет иметь такой же размер, как и хеш-индекс по большому текстовому столбцу.

В результате поиск обычно оказывается молниеносным. Однако хеш-индексы имеют некоторые ограничения:

- Поскольку индекс содержит только хеш-коды и указатели на строки, а не сами значения, MySQL не может использовать данные в индексе, чтобы избежать чтения строк. К счастью, доступ к строкам в памяти очень быстр, так что обычно это не снижает производительность.

- MySQL не может использовать хеш-индексы для сортировки, поскольку строки в нем не хранятся в отсортированном порядке.
- Хеш-индексы не поддерживают поиск по частичному ключу, так как хеш-коды вычисляются для всего индексируемого значения. То есть, если имеется индекс по столбцам(A,B), а во фразе WHERE упоминается только столбец A, индекс не поможет.
- Хеш-индексы поддерживают только сравнения на равенство, использующие операторы =, IN() и <=> (обратите внимание, что <> и <=> – разные операторы). Они не ускоряют поиск по диапазону, например WHERE price > 100.
- Доступ к данным в хеш-индексе очень быстр, если нет большого количества коллизий (нескольких значений с одним и тем же хеш-кодом). При наличии коллизий подсистема хранения вынуждена пройти по каждому указателю на строку, хранящемуся в связанном списке, и сравнить значение в этой строке с искомым.
- Некоторые операции обслуживания индекса могут оказаться медленными, если количество коллизий велико. Например, если вы создаете хеш-индекс по столбцу с очень маленькой селективностью (selectivity), что означает много коллизий, а затем удаляете строку из таблицы, то поиск указателя на эту строку в индексе может оказаться дорогим. Подсистеме хранения придется проверять каждую строку в связанном списке для этого ключа, чтобы найти и удалить ссылку на ту единственную строку, которая была удалена.

Эти ограничения делают хеш-индексы полезными только в отдельных случаях. Однако если они соответствуют потребностям приложения, то могут значительно повысить производительность. В качестве примера можно привести хранилища данных, где классическая схема «звезда» подразумевает наличие соединений с большим числом справочных таблиц. Хеш-индексы – это именно то, что требуется для подобного случая.

Кроме явных хеш-индексов в подсистеме хранения Memory, поддержка уникальных хеш-индексов есть в кластерной подсистеме NDB Cluster. Но они специфичны только для этой подсистемы, которую мы в данной книге не описываем.

Подсистема хранения InnoDB поддерживает так называемые *адаптивные хеш-индексы*. Когда InnoDB замечает, что доступ к некоторым значениям индекса происходит очень часто, она строит для них хеш-индекс в памяти, помимо уже имеющихся B-Tree-индексов. Тем самым к B-Tree-индексам добавляются некоторые свойства хеш-индексов, например очень быстрый поиск. Этот процесс полностью автоматический, и вы не можете ни контролировать, ни настраивать его.

Построение собственных хеш-индексов

Если подсистема хранения не поддерживает хеш-индексы, то вы можете эмулировать их самостоятельно подобно тому, как это делает InnoDB.

Данный подход позволяет использовать некоторые достоинства хеш-индексов, например небольшие размеры при очень длинных ключах.

Идея проста: создайте псевдохеш-индекс поверх стандартного B-Tree-индекса. Он будет не совсем идентичен настоящему хеш-индексу, поскольку для поиска по-прежнему будет использоваться B-Tree-индекс. Однако искаться будут хеш-коды ключей вместо самих ключей. От вас требуется лишь вручную указать хеш-функцию во фразе `WHERE` запроса.

Примером хорошей работы подобного подхода является поиск адресов URL. B-Tree-индексы по адресам URL обычно оказываются очень большими, поскольку сами URL длинные. Обычно запрос к таблице адресов URL выглядит примерно так:

```
mysql> SELECT id FROM url WHERE url="http://www.mysql.com";
```

Но если удалить индекс по столбцу `url` и добавить в таблицу индексированный столбец `url_crc`, то можно переписать этот запрос в таком виде:

```
mysql> SELECT id FROM url WHERE url="http://www.mysql.com"
    -> AND url_crc=CRC32("http://www.mysql.com");
```

Этот подход хорошо работает, поскольку оптимизатор запросов MySQL замечает, что существует небольшой высокоизбирательный индекс по столбцу `url_crc`, и осуществляет поиск в индексе элементов с этим значением (в данном случае 1560514994). Даже если несколько строк имеют одно и то же значение `url_crc`, эти строки очень легко найти с помощью быстрого целочисленного сравнения, а затем отыскать среди них то, которое в точности соответствует полному адресу URL. Альтернативой является индексирование URL как строки, что происходит значительно медленнее.

Одним из недостатков данного решения является необходимость обновлять хеш-значения. Вы можете делать это вручную или – в MySQL 5.0 и более поздних версиях – использовать триггеры. В следующем примере показано, как триггеры могут помочь в вычислении столбца `url_crc` при вставке и обновлении значений. Для начала создадим таблицу:

```
CREATE TABLE pseudohash (
    id int unsigned NOT NULL auto_increment,
    url varchar(255) NOT NULL,
    url_crc int unsigned NOT NULL DEFAULT 0,
    PRIMARY KEY(id)
);
```

Затем напишем триггеры. Мы временно изменяем разделитель команд, чтобы можно было использовать точку с запятой внутри триггера:

```
DELIMITER |

CREATE TRIGGER pseudohash_crc_ins BEFORE INSERT ON pseudohash FOR EACH ROW BEGIN
SET NEW.url_crc=crc32(NEW.url);
```

```
END;
|

CREATE TRIGGER pseudohash_crc_upd BEFORE UPDATE ON pseudohash FOR EACH ROW BEGIN
SET NEW.url_crc=crc32(NEW.url);
END;
|

DELIMITER ;
```

Остается лишь проверить, что триггер действительно изменяет хеш-код:

```
mysql> INSERT INTO pseudohash (url) VALUES ('http://www.mysql.com');
mysql> SELECT * FROM pseudohash;
+----+----------------------+------------+
| id | url                  | url_crc    |
+----+----------------------+------------+
| 1  | http://www.mysql.com | 1560514994 |
+----+----------------------+------------+
mysql> UPDATE pseudohash SET url='http://www.mysql.com/' WHERE id=1;
mysql> SELECT * FROM pseudohash;
+----+-----------------------+------------+
| id | url                   | url_crc    |
+----+-----------------------+------------+
| 1  | http://www.mysql.com/ | 1558250469 |
+----+-----------------------+------------+
```

При таком подходе не следует использовать хеш-функции `SHA1()` или `MD5()`. Они возвращают очень длинные строки, которые требуют много пространства и приводят к замедлению сравнения. Это мощные криптографические функции, предназначенные для почти гарантированного устранения коллизий, а не для наших текущих целей. Простые хеш-функции могут дать приемлемый уровень коллизий с лучшей производительностью.

Если в таблице большое количество строк и функция `CRC32()` дает слишком много коллизий, реализуйте собственную 64-разрядную хеш-функцию. Такая функция должна возвращать целое число, а не строку. Одним из способов реализации 64-разрядной хеш-функции является использование только части значения, возвращаемого функцией `MD5()`. Это, вероятно, менее эффективно, чем написание своей собственной функции (см. раздел «Определяемые пользователем функции» в главе 5 на стр. 290), зато требует минимума усилий:

```
mysql> SELECT CONV(RIGHT(MD5('http://www.mysql.com/'), 16), 16, 10) AS HASH64;
+---------------------+
| HASH64              |
+---------------------+
| 9761173720318281581 |
+---------------------+
```

Комплект инструментов Maatkit (*http://maatkit.sourceforge.net*) включает UDF-функцию, которая реализует очень быстрый алгоритм Фаулера/Нолла/Воу для вычисления 64-разрядного хеш-кода.

Обработка коллизий хеширования

При поиске значения по его хеш-коду следует включать во фразу `WHERE` и само искомое значение:

```
mysql> SELECT id FROM url WHERE url_crc=CRC32("http://www.mysql.com")
    -> AND url="http://www.mysql.com";
```

Следующий запрос *неправилен*, поскольку, если имеется другой URL, для которого функция `CRC32()` возвращает то же значение 1560514994, то будут возвращены обе строки:

```
mysql> SELECT id FROM url WHERE url_crc=CRC32("http://www.mysql.com");
```

Вероятность коллизий растет значительно быстрее, чем можно предположить. Функция `CRC32()` возвращает 32-разрядное целое число, поэтому вероятность коллизии достигает 1% уже при 93000 значений. Чтобы проиллюстрировать это, мы загрузили в таблицу все слова из файла */usr/share/dict/words* вместе со значениями `CRC32()`, в итоге получилось 98569 строк. В этом наборе данных уже есть одна коллизия! Из-за нее следующий запрос возвращает более одной строки:

```
mysql> SELECT word, crc FROM words WHERE crc = CRC32('gnu');
+---------+------------+
| word    | crc        |
+---------+------------+
| codding | 1774765869 |
| gnu     | 1774765869 |
+---------+------------+
```

Правильный запрос выглядит следующим образом:

```
mysql> SELECT word, crc FROM words WHERE crc = CRC32('gnu') AND word = 'gnu';
+------+------------+
| word | crc        |
+------+------------+
| gnu  | 1774765869 |
+------+------------+
```

Чтобы избежать проблем с коллизиями, вы должны указать во фразе `WHERE` оба условия. Если коллизии не являются проблемой – например, потому что вы делаете статистические запросы и вам не нужны точные результаты, – то можно повысить эффективность, оставив во фразе `WHERE` только сравнение со значением функции `CRC32()`.

Пространственные индексы (Spatial, R-Tree)

MyISAM поддерживает пространственные индексы, которые можно строить по стобцам пространственного типа, например `GEOMETRY`. Одна-

ко для того чтобы R-Tree индексы работали, необходимо использовать геоинформационные функции MySQL, например MBRCONTAINS().

Полнотекстовые индексы

Полнотекстовый (FULLTEXT) индекс представляет собой специальный тип индекса для таблиц типа MyISAM. Он позволяет искать в тексте ключевые слова, а не сравнивать искомое значение со значениями в столбце. Полнотекстовый поиск не имеет ничего общего с другими типами поиска. С ним связано много тонкостей, например стоп-слова, стемминг, учет множественного числа, а также булевский поиск. Он гораздо больше напоминает поисковые системы, нежели обычное сравнение с критерием во фразе WHERE.

Наличие полнотекстового индекса по столбцу не делает B-Tree-индекс по этому же столбцу менее ценным. Полнотекстовые индексы предназначены для операций MATCH AGAINST, а не обычных операций с фразой WHERE.

Полнотекстовое индексирование мы обсудим более подробно в разделе «Полнотекстовый поиск» в главе 5 на стр. 307.

Стратегии индексирования для достижения высокой производительности

Создание правильных индексов и их корректное использование существенны для достижения высокой производительности запросов. Мы рассказали о различных типах индексов, об их достоинствах и недостатках. Теперь давайте посмотрим, как на практике использовать мощь индексов.

Существует много способов эффективного выбора и использования индексов. Умение определить, что и когда применять, а также оценивать влияние вашего выбора на производительность системы является тем навыком, который приходит с опытом. Следующие разделы помогут вам понять, как эффективно использовать индексы. Однако не забывайте про эталонное тестирование!

Изоляция столбца

MySQL обычно не может использовать индекс по столбцу, если этот столбец не изолирован в запросе. «Изоляция» столбца означает, что он не должен быть частью выражения или употребляться в качестве аргумента внутри функции.

Например, вот запрос, который не может использовать индекс по столбцу actor_id:

```
mysql> SELECT actor_id FROM sakila.actor WHERE actor_id + 1 = 5;
```

Человеку легко понять, что фраза WHERE эквивалентна выражению actor_id = 4, но MySQL не умеет решать уравнения. Так что это придется де-

лать вам. Следует взять за правило упрощать критерии во фразе WHERE, так, чтобы индексированный столбец оказывался в одиночестве по одну сторону от оператора сравнения.

Вот пример другой распространенной ошибки:

```
mysql> SELECT ... WHERE TO_DAYS(CURRENT_DATE) - TO_DAYS(date_col) <= 10;
```

Этот запрос найдет все строки, где значение столбца date_col отстоит от текущей даты не более чем на 10 дней, но не будет использовать индексы из-за функции TO_DAYS(). Лучше записать этот запрос так:

```
mysql> SELECT ... WHERE date_col >= DATE_SUB(CURRENT_DATE, INTERVAL 10 DAY);
```

В таком виде не будет проблем с использованием индекса, но запрос можно еще улучшить. Параметр CURRENT_DATE не позволяет кэшировать результаты запроса. Для решения этой проблемы замените CURRENT_DATE на литеральное значение:

```
mysql> SELECT ... WHERE date_col >= DATE_SUB('2008-01-17', INTERVAL 10 DAY);
```

Кэширование запросов подробно описано в главе 5.

Префиксные индексы и селективность индекса

Иногда нужно проиндексировать очень длинные символьные столбцы, из-за чего индексы становятся большими и медленными. Одной из стратегий является эмулирование хеш-индекса, как мы показали выше. Но порой этого оказывается недостаточно. Что еще можно сделать?

Вы часто можете сэкономить пространство и получить хорошую производительность, проиндексировав первые несколько символов, а не все значение. Тогда индекс будет занимать меньше места, но станет менее *селективным*. *Селективность индекса* – это отношение количества различных проиндексированных значений (*кардинальности*) к общему количеству строк в таблице ($\#T$). Диапазон возможных значений селективности от $1/\#T$ до 1. Индекс с высокой селективностью хорош тем, что позволяет MySQL при поиске соответствий отфильтровывать больше строк. Уникальный индекс имеет селективность, равную единице.

Префикс столбца часто оказывается весьма избирательным, чтобы обеспечить хорошую производительность. Если вы индексируете столбцы типа BLOB или TEXT, либо очень длинные столбцы типа VARCHAR, то *обязаны* определять префиксные индексы, поскольку MySQL не позволяет индексировать такие столбцы по их полной длине.

Сложность заключается в выборе длины префикса, которая должна быть достаточно велика, чтобы обеспечить хорошую селективность, но не слишком велика, чтобы сэкономить пространство. Префикс избирается настолько длинным, чтобы польза от его использования была почти такая же, как от использования индекса по полному столбцу. Другими словами, кардинальность префикса должна быть почти такой же, как кардинальность всего столбца.

Чтобы определить подходящую длину префикса, найдите наиболее часто встречающиеся значения и сравните их перечень со списком чаще всего используемых префиксов. В тестовой базе данных Sakila[1] нет подходящего примера для демонстрации, поэтому мы создадим таблицу на основе таблицы `city`, так, чтобы у нас было достаточно данных:

```
CREATE TABLE sakila.city_demo(city VARCHAR(50) NOT NULL);
INSERT INTO sakila.city_demo(city) SELECT city FROM sakila.city;
-- Повторить следующую команду пять раз:
INSERT INTO sakila.city_demo(city) SELECT city FROM sakila.city_demo;
-- Рандомизируем распределение (неэффективно, зато удобно):
UPDATE sakila.city_demo
SET city = (SELECT city FROM sakila.city ORDER BY RAND( ) LIMIT 1);
```

Теперь у нас есть тестовый набор значений. Распределение результатов далеко от реалистичного, так что мы использовали функцию `RAND()`. Из-за этого вы будете наблюдать результаты, отличные от наших, но для данного примера это не играет существенной роли. Сначала мы находим наиболее часто встречающиеся города:

```
mysql> SELECT COUNT(*) AS cnt, city
    -> FROM sakila.city_demo GROUP BY city ORDER BY cnt DESC LIMIT 10;
+-----+----------------+
| cnt | city           |
+-----+----------------+
| 65 | London          |
| 49 | Hiroshima       |
| 48 | Teboksary       |
| 48 | Pak Kret        |
| 48 | Yaound          |
| 47 | Tel Aviv-Jaffa  |
| 47 | Shimoga         |
| 45 | Cabuyao         |
| 45 | Callao          |
| 45 | Bislig          |
+-----+----------------+
```

Обратите внимание, что каждое значение встречается от 45 до 65 раз. Теперь найдем наиболее часто встречающиеся префиксы названий городов, начиная с трехбуквенных:

```
mysql> SELECT COUNT(*) AS cnt, LEFT(city, 3) AS pref
    -> FROM sakila.city_demo GROUP BY pref ORDER BY cnt DESC LIMIT 10;
+-----+------+
| cnt | pref |
+-----+------+
| 483 | San  |
| 195 | Cha  |
| 177 | Tan  |
| 167 | Sou  |
```

[1] Сакила – имя дельфина, изображенного на логотипе MySQL. – *Прим. ред.*

```
| 163 | al-  |
| 163 | Sal  |
| 146 | Shi  |
| 136 | Hal  |
| 130 | Val  |
| 129 | Bat  |
+-----+------+
```

Количество вхождений каждого префикса значительно больше, поэтому уникальных префиксов намного меньше, чем уникальных полных названий городов. Идея состоит в увеличении длины префикса до тех пор, пока он не станет почти таким же селективным, как полная длина столбца. Несколько экспериментов позволили выяснить, что семи символов вполне достаточно:

```
mysql> SELECT COUNT(*) AS cnt, LEFT(city, 7) AS pref
    -> FROM sakila.city_demo GROUP BY pref ORDER BY cnt DESC LIMIT 10;
+-----+---------+
| cnt | pref    |
+-----+---------+
|  70 | Santiag |
|  68 | San Fel |
|  65 | London  |
|  61 | Valle d |
|  49 | Hiroshi |
|  48 | Teboksa |
|  48 | Pak Kre |
|  48 | Yaound  |
|  47 | Tel Avi |
|  47 | Shimoga |
+-----+---------+
```

Другой способ определить подходящую длину префикса состоит в том, чтобы вычислить селективность полного столбца и попытаться подобрать длину префикса, обеспечивающую близкую селективность. Вот как найти селективность полного столбца:

```
mysql> SELECT COUNT(DISTINCT city)/COUNT(*) FROM sakila.city_demo;
+-------------------------------+
| COUNT(DISTINCT city)/COUNT(*) |
+-------------------------------+
| 0.0312                        |
+-------------------------------+
```

В среднем префикс будет примерно так же хорош, если его селективность равна приблизительно 0.031. Можно оценить несколько разных длин префиксов в одном запросе, что полезно для очень больших таблиц. Вот как найти селективность для нескольких значений длины префикса в одном запросе:

```
mysql> SELECT COUNT(DISTINCT LEFT(city, 3))/COUNT(*) AS sel3,
    -> COUNT(DISTINCT LEFT(city, 4))/COUNT(*) AS sel4,
```

```
    -> COUNT(DISTINCT LEFT(city, 5))/COUNT(*) AS sel5,
    -> COUNT(DISTINCT LEFT(city, 6))/COUNT(*) AS sel6,
    -> COUNT(DISTINCT LEFT(city, 7))/COUNT(*) AS sel7
    -> FROM sakila.city_demo;
+--------+--------+--------+--------+--------+
| sel3   | sel4   | sel5   | sel6   | sel7   |
+--------+--------+--------+--------+--------+
| 0.0239 | 0.0293 | 0.0305 | 0.0309 | 0.0310 |
+--------+--------+--------+--------+--------+
```

Этот запрос показывает, что последовательное увеличение длины префикса дает небольшое улучшение селективности вплоть до семи символов.

Недостаточно обращать внимание только на среднюю селективность. Следует подумать также о селективности *в худшем случае*. На основе средней селективности вы можете прийти к выводу, что префикса длиной в четыре или пять символов достаточно, но если ваши данные распределены очень неравномерно, это может завести вас в ловушку. Посмотрев на количество вхождений наиболее распространенных четырехбуквенных префиксов названий городов, вы ясно увидите неравномерность:

```
mysql> SELECT COUNT(*) AS cnt, LEFT(city, 4) AS pref
-> FROM sakila.city_demo GROUP BY pref ORDER BY cnt DESC LIMIT 5;
+-----+------+
| cnt | pref |
+-----+------+
| 205 | San  |
| 200 | Sant |
| 135 | Sout |
| 104 | Chan |
| 91 | Toul  |
+-----+------+
```

При длине в четыре символа наиболее распространенные префиксы встречаются значительно чаще, чем самые распространенные полные значения. То есть селективность по этим значениям ниже, чем средняя селективность. Если у вас есть более реалистичный набор данных, чем эта сгенерированная случайным образом выборка, то, вероятно, этот эффект может оказаться значительно более выраженным. Например, построение четырехзначного префиксного индекса по реальным названиям городов мира даст очень низкую селективность по городам, начинающимся на «San» и «New», которых очень много.

Теперь, отыскав подходящую длину префикса для наших тестовых данных, создадим индекс по префиксу столбца:

```
mysql> ALTER TABLE sakila.city_demo ADD KEY (city(7));
```

Префиксные индексы могут стать хорошим способом уменьшения размера и повышения быстродействия индекса, но у них есть и недостат-

ки: MySQL не может использовать префиксные индексы ни для запросов с фразами ORDER BY и GROUP BY, ни как покрывающие индексы.

Иногда имеет смысл создавать суффиксные индексы (например, для поиска всех адресов электронной почты из определенного домена). Сам MySQL не поддерживает индексы по реверсированному ключу (reversed index). Но вы можете самостоятельно хранить реверсированные строки и создавать по ним префиксный индекс. Поддерживать этот индекс можно с помощью триггеров (см. выше в разделе «Построение собственных хеш-индексов» этой главы на стр. 143).

Кластерные индексы

Кластерные индексы[1] не являются отдельным типом индекса. Скорее, это подход к хранению данных. Детали в разных реализациях отличаются, но в InnoDB кластерный индекс фактически содержит и B-Tree-индекс, и сами строки в одной и той же структуре.

Когда над таблицей построен кластерный индекс, в листовых страницах индекса хранятся сами строки. Термин «кластерный» означает, что строки с близкими значениями ключа хранятся по соседству[2]. Над таблицей можно построить только один кластерный индекс, поскольку невозможно хранить одну и ту же строку одновременно в двух местах (однако *покрывающие индексы* позволяют эмулировать несколько кластерных индексов, о чем будет рассказано ниже в этой главе).

Поскольку за реализацию индексов отвечают подсистемы хранения, не все из них поддерживают кластерные индексы. В настоящее время этим могут похвастаться только solidDB и InnoDB. В настоящем разделе мы будем говорить исключительно об InnoDB, но обсуждаемые принципы, по крайней мере, частично будут применимы к любой подсистеме хранения, поддерживающей кластерные индексы в настоящее время или в будущем.

На рис. 3.3 показано, как располагаются записи в кластерном индексе. Обратите внимание, что листовые страницы содержат сами строки, а узловые – только индексированные столбцы. В рассматриваемом примере индексированный столбец содержит целочисленные значения.

Некоторые СУБД позволяют выбрать, какой индекс сделать кластерным, но на данный момент ни одна из подсистем хранения MySQL не обладает такой возможностью. InnoDB кластеризует данные по первичному ключу. Это означает, что «индексированный столбец» на рис. 3.3 является столбцом, содержащим первичный ключ.

---

[1] Пользователи Oracle, вероятно, найдут сходство между «индексно-организованной таблицей» и таблицей **InnoDB с построенным кластерным индексом**.

[2] Это не всегда так, и вы скоро узнаете, почему.

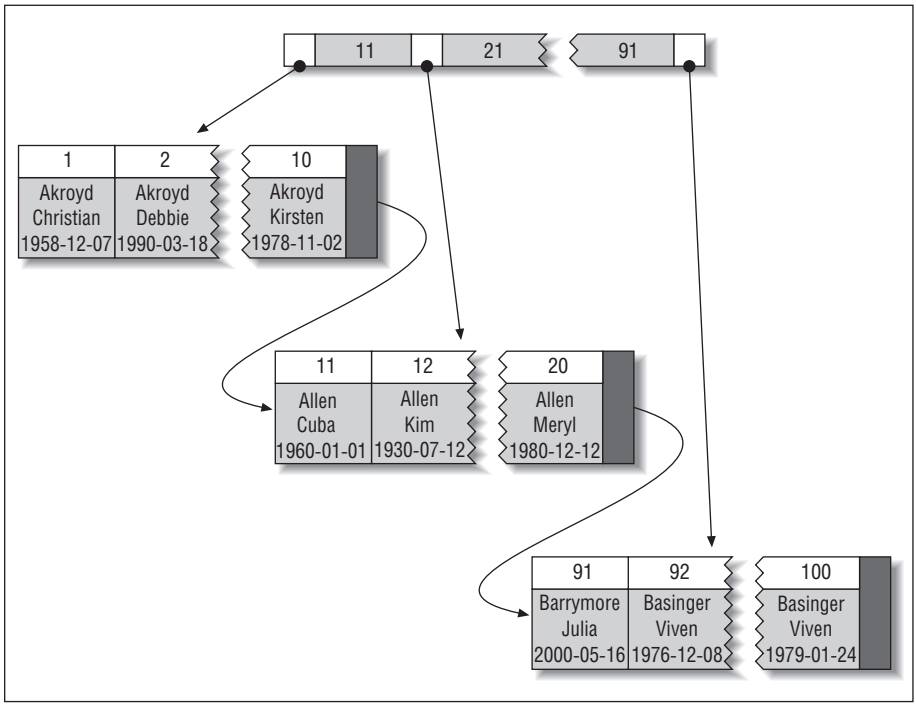


Рис. 3.3. *Расположение записей в кластерном индексе*

Если вы не определили первичный ключ, то InnoDB попытается использовать вместо него уникальный индекс, не допускающий пустых значений. Если такого индекса не существует, InnoDB определит скрытый первичный ключ за вас и затем кластеризует таблицу по нему[1]. InnoDB кластеризует записи вместе только внутри страницы. Разные страницы с близкими значениями ключей могут оказаться далеко друг от друга.

Первичный кластерный ключ иногда может увеличить производительность, а иногда заметно снизить ее. Таким образом, решение о кластеризации нужно принимать обдуманно, особенно при замене подсистемы хранения таблицы с InnoDB на какую-то другую и наоборот.

Кластеризованные данные имеют несколько очень важных достоинств:

- Вы можете хранить связанные данные рядом. Например, при реализации почтового ящика можно кластеризовать таблицу по столбцу `user_id`, тогда для выборки всех сообщений одного пользователя нужно будет прочитать с диска лишь небольшое количество страниц. Если не использовать кластеризацию, то для каждого сообщения может потребоваться отдельная операция дискового ввода/вывода.

[1] Так же действует подсистема хранения solidDB.

- Быстрый доступ к данным. Кластерный индекс хранит и индекс, и данные вместе в одной B-Tree структуре, поэтому извлечение строк из кластерного индекса обычно происходит быстрее, чем сопоставимый поиск в некластерном индексе.
- Использующие покрывающие индексы запросы могут получить значение первичного ключа из листового узла.

Эти преимущества значительно увеличат производительность, если вы спроектируете свои таблицы и запросы с их учетом. Однако у кластерных индексов есть и недостатки:

- Кластеризация дает значительные улучшения, когда рабочая нагрузка характеризуется большим количеством операций ввода/вывода. Если данные помещаются в памяти, то порядок доступа к ним не имеет значения, и тогда кластерные индексы не принесут большой пользы.
- Скорость операций вставки сильно зависит от порядка обработки данных. Вставка строк в порядке, соответствующем первичному ключу, является самым быстрым способом загрузить данные в таблицу InnoDB. Если вы загружаете большое количество данных в другом порядке, то по окончании загрузки имеет смысл реорганизовать таблицу с помощью команды `OPTIMIZE TABLE`.
- Обновление столбцов кластерного индекса обходится дорого, поскольку InnoDB вынуждена перемещать каждую обновленную строку в новое место.
- Для таблиц с кластерным индексом вставка новых строк или обновление первичного ключа, требующее перемещения строки, может приводить к *расщеплению страницы*. Это происходит тогда, когда значение ключа строки таково, что строка должна была помещена в страницу, заполненную данными. Чтобы строка поместилась, подсистема хранения вынуждена в этом случае разбить страницу на две. Из-за расщепления страниц таблица занимает больше места на диске.
- Полное сканирование кластерных таблиц может оказаться более медленным, особенно если строки упакованы менее плотно или хранятся непоследовательно из-за расщепления страниц.
- Вторичные (некластерные) индексы могут оказаться больше, чем вы ожидаете, поскольку в листовых узлах хранятся значения столбцов, составляющих первичный ключ.
- Для доступа к данным по вторичному индексу требуется просмотр двух индексов вместо одного.

Последний момент может быть не совсем ясен. Почему доступ с использованием вторичного индекса требует двух операций просмотра? Ответ заключается в природе «указателей на строки», которые хранятся во вторичном индексе. Не забывайте, что в листовом узле содержится не указатель на физический адрес строки, а значение ее первичного ключа.

Это означает, что в процессе поиска строки по вторичному индексу подсистема хранения должна сначала найти в нем листовый узел, а затем использовать хранящееся там значение первичного ключа для отыскания по нему самой строки. Это двойная работа: два прохода по B-дереву вместо одного (в InnoDB адаптивный хеш-индекс помогает уменьшить эти потери).

Сравнение размещения данных в InnoDB и MyISAM

Различия в организации кластеризованного и некластеризованного размещения данных, а также соответствующая разница между первичными и вторичными индексами могут приводить к путанице и неожиданностям. Рассмотрим, как InnoDB и MyISAM разместят данные следующей таблицы:

```
CREATE TABLE layout_test (
    col1 int NOT NULL,
    col2 int NOT NULL,
    PRIMARY KEY(col1),
    KEY(col2)
);
```

Предположим, что в таблицу было добавлено 10 000 строк. Значение первичного ключа для каждой вставляемой строки случайным образом выбиралось из диапазона от 1 до 10 000. Затем была произведена оптимизация с помощью команды `OPTIMIZE TABLE`. Другими словами, данные размещаются на диске оптимальным образом (дефрагментированы), но строки могут располагаться в случайном порядке. Элементам столбца `col2` присвоены случайные значения между 1 и 100, поэтому есть много дубликатов.

Размещение данных в MyISAM

Размещение данных в подсистеме MyISAM проще, поэтому мы начнем с него. MyISAM сохраняет данные на диске в том порядке, в котором они были вставлены, как показано на рис. 3.4.

Рядом со строками мы привели их номера, начиная с нуля. Поскольку строки имеют фиксированный размер, MyISAM может найти любую из них путем смещения на требуемое количество байтов от начала таблицы (MyISAM не всегда использует «номера строк», которые мы показали; в зависимости от того, имеют ли строки фиксированный или переменный размер, эта подсистема хранения использует различные стратегии).

При таком размещении построение индекса не вызывает сложности. Мы проиллюстрируем это с помощью последовательности диаграмм, отбросив такие физические детали, как страницы, и показывая в индексе только «узлы». Каждый листовой узел в индексе может просто содержать номер строки. На рис. 3.5 проиллюстрирован первичный ключ таблицы.

Мы опустили некоторые детали, например, то, что у одного внутреннего узла B-дерева может быть несколько внутренних узлов-потомков, но

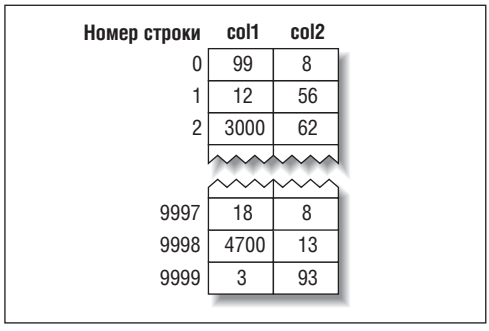


Рис. 3.4. *Размещение данных для таблицы layout_test в MyISAM*

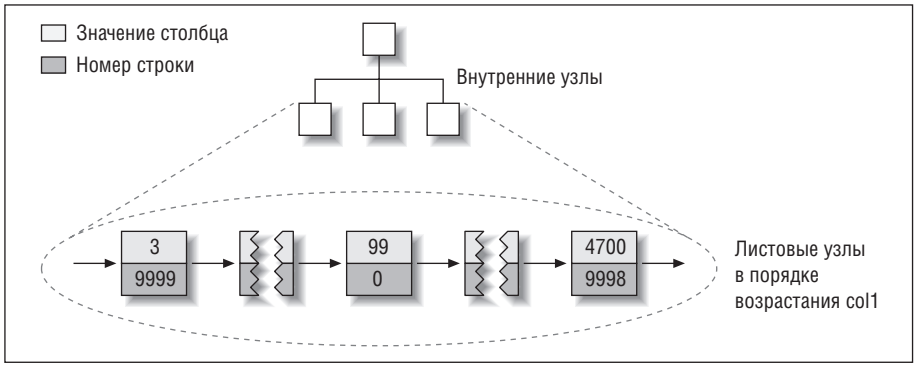


Рис. 3.5. *Размещение первичного ключа для таблицы layout_test в MyISAM*

для общего понимания размещения данных в некластерной подсистеме хранения это не существенно.

Что можно сказать об индексе по столбцу `col2`? Есть ли здесь что-то особенное? Оказывается, ничего – это такой же индекс, как любой другой. На рис. 3.6 показан индекс по столбцу `col2`.

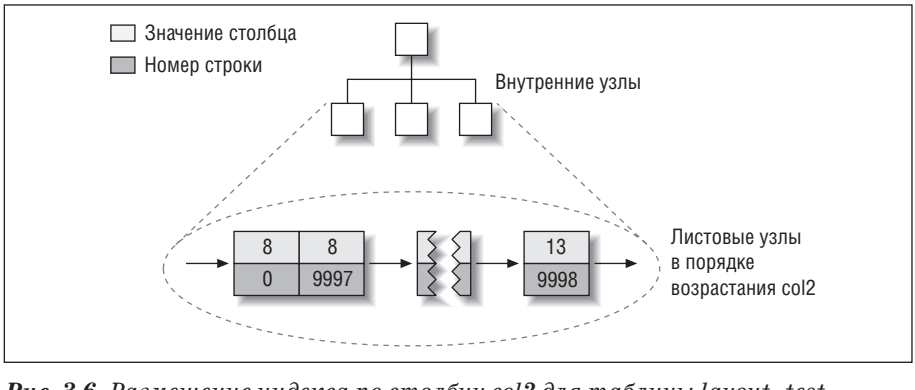


Рис. 3.6. *Размещение индекса по столбцу col2 для таблицы layout_test в MyISAM*

Фактически в MyISAM отсутствуют структурные различия между первичным ключом и любым другим индексом. Первичный ключ является просто уникальным индексом, не допускающим пустых значений под названием `PRIMARY`.

Размещение данных в InnoDB

Подсистема InnoDB хранит те же самые данные совсем по-другому в силу своей кластерной организации. InnoDB формирует таблицу так, как показано на рис. 3.7.

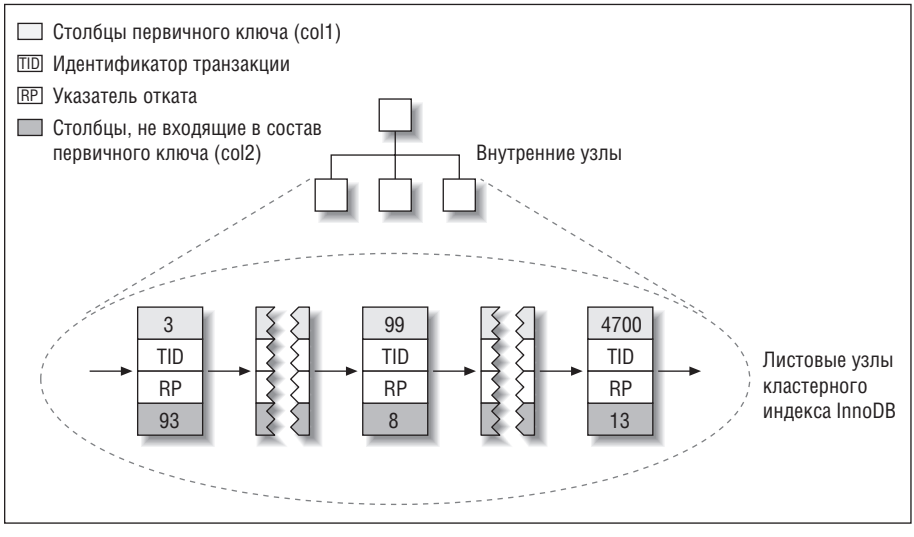


Рис. 3.7. *Размещение первичного ключа для таблицы layout_test в InnoDB*

На первый взгляд, особых отличий от рис. 3.5 нет. Но посмотрите внимательнее, и вы заметите, что на рисунке показана *вся таблица*, а не только индекс. Поскольку кластерный индекс в InnoDB «является» таблицей, то отдельного хранилища для строк, как в MyISAM, нет.

Каждый листовый узел в кластерном индексе содержит значение первичного ключа, идентификатор транзакции и указатель отката, который InnoDB использует для поддержки транзакций и механизма MVCC, а также прочие столбцы (в данном случае `col2`). Если первичный ключ создан по префиксу столбца, то в InnoDB вместе с остальными хранится и полное значение этого столбца.

Вторичные индексы в InnoDB сильно отличаются от кластерных. Листовые узлы вторичных индексов в данной системе содержат вместо «указателей на строки» значения первичного ключа, которые выступают в роли таких «указателей». Такая стратегия уменьшает объем работы, необходимой для обслуживания вторичных индексов при перемещении строки или в момент расщепления страницы данных. Использо-

вание значений первичного ключа строки в качестве указателя увеличивает размер индекса, но это также означает, что InnoDB может перемещать строку без обновления указателей на нее.

Рис. 3.8 иллюстрирует индекс по столбцу `col2` для демонстрационной таблицы. Каждый листовый узел содержит индексированные столбцы (в данном случае только `col2`), за которыми следуют значения первичного ключа (`col1`).

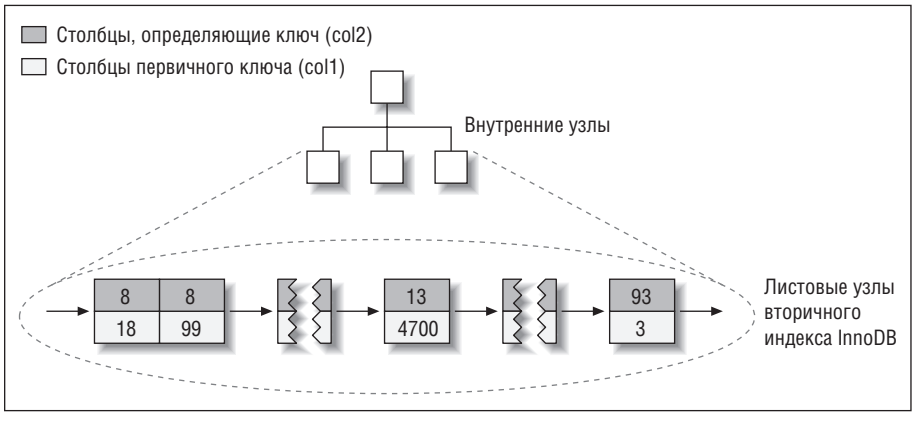


***Рис. 3.8.** Размещение вторичного индекса для таблицы layout_test в InnoDB*

Эти диаграммы иллюстрируют листовые узлы B-Tree индекса, но мы умышленно опустили детали, касающиеся нелистовых узлов. Каждый нелистовой узел B-Tree индекса в InnoDB содержит индексированные столбцы плюс указатель на узел следующего уровня (которым может быть либо другой нелистовой, либо листовой узел). Это относится ко всем индексам, как кластерным, так и вторичным.

На рис. 3.9 показано абстрактное представление организации таблицы в InnoDB и MyISAM. Легко увидеть различия между тем, как хранятся данные и индексы в этих двух системах.

Если вы не понимаете, чем отличается кластерное и некластерное хранение и почему это так важно, не расстраивайтесь. Это станет яснее, когда вы узнаете больше, особенно в конце этого раздела и в следующей главе. Данные концепции очень непросты, и для их полного осмысления требуется время.

Вставка строк в порядке первичного ключа в InnoDB

Если вы используете InnoDB и вам не требуется никакая конкретная кластеризация, то имеет смысл определить *суррогатный ключ*, то есть первичный ключ, значение которого не имеет прямой связи с данными вашего приложения. Обычно самым простым способом является использование столбца с атрибутом `AUTO_INCREMENT`. Это гарантирует, что

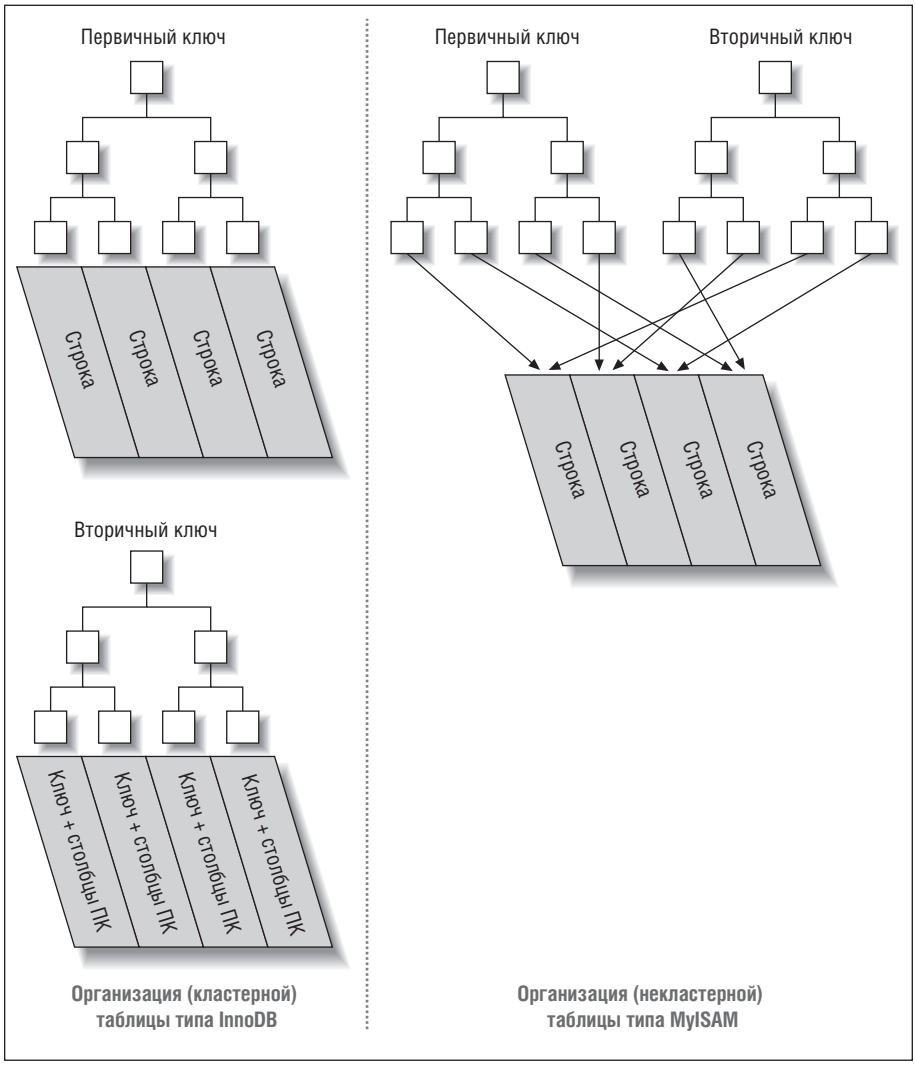


Рис. 3.9. *Кластерные и некластерные таблицы*

значение поля, по которому построен первичный ключ, монотонно возрастает, что в свою очередь обеспечивает лучшую производительность соединений, использующих первичный ключ.

Лучше избегать случайных (непоследовательных) кластерных ключей. Например, использование значений UUID является плохим выбором с точки зрения производительности: это делает вставку в кластерный индекс случайной, что является худшим сценарием, и не приводит к полезной кластеризации данных.

В целях демонстрации мы провели тесты производительности для двух ситуаций. В первом случае выполнялась вставка в таблицу `userinfo` с целочисленным идентификатором, определенную следующим образом:

```
CREATE TABLE userinfo (
    id              int unsigned NOT NULL AUTO_INCREMENT,
    name            varchar(64) NOT NULL DEFAULT '',
    email           varchar(64) NOT NULL DEFAULT '',
    password        varchar(64) NOT NULL DEFAULT '',
    dob             date DEFAULT NULL,
    address         varchar(255) NOT NULL DEFAULT '',
    city            varchar(64) NOT NULL DEFAULT '',
    state_id        tinyint unsigned NOT NULL DEFAULT '0',
    zip             varchar(8) NOT NULL DEFAULT '',
    country_id      smallint unsigned NOT NULL DEFAULT '0',
    gender          ('M','F') NOT NULL DEFAULT 'M',
    account_type    varchar(32) NOT NULL DEFAULT '',
    verified        tinyint NOT NULL DEFAULT '0',
    allow_mail      tinyint unsigned NOT NULL DEFAULT '0',
    parrent_account int unsigned NOT NULL DEFAULT '0',
    closest_airport varchar(3) NOT NULL DEFAULT '',
    PRIMARY KEY (id),
    UNIQUE  KEY email (email),
    KEY     country_id (country_id),
    KEY     state_id (state_id),
    KEY     state_id_2 (state_id,city,address)
) ENGINE=InnoDB
```

Обратите внимание на целочисленный автоинкрементный первичный ключ.

Вторая таблица, `userinfo_uuid`, идентична таблице `userinfo`, за исключением того, что первичным ключом является UUID, а не целое число:

```
CREATE TABLE userinfo_uuid (
    uuid varchar(36) NOT NULL,
    ...
```

Мы протестировали обе таблицы. Сначала мы вставили в каждую по миллиону строк на сервере, имеющем достаточно памяти для размещения в ней индексов. Затем мы вставили по три миллиона строк в те же таблицы, и это увеличило индексы настолько, что они перестали помещаться в памяти. В табл. 3.2 приведено сравнение результатов тестирования.

Обратите внимание: в случае первичного ключа типа UUID не только вставка строк заняла больше времени, но и размер индекса значительно увеличился. Одной из причин является больший размер первичного ключа, но, несомненно, влияние оказали также расщепление страниц и проистекающая из этого фрагментация.

Таблица 3.2. Результаты тестирования вставки строк в таблицы InnoDB

Таблица	Количество строк	Время (сек)	Размер индекса (Мбайт)
userinfo	1 000 000	137	342
userinfo_uuid	1 000 000	180	544
userinfo	3 000 000	1233	1036
userinfo_uuid	3 000 000	4525	1707

Чтобы понять, почему это так, давайте посмотрим, что происходило в индексе, когда мы вставляли данные в первую таблицу. На рис. 3.10 показано, как вставляемые строки сначала заполняют одну страницу, а затем переходят на следующую.

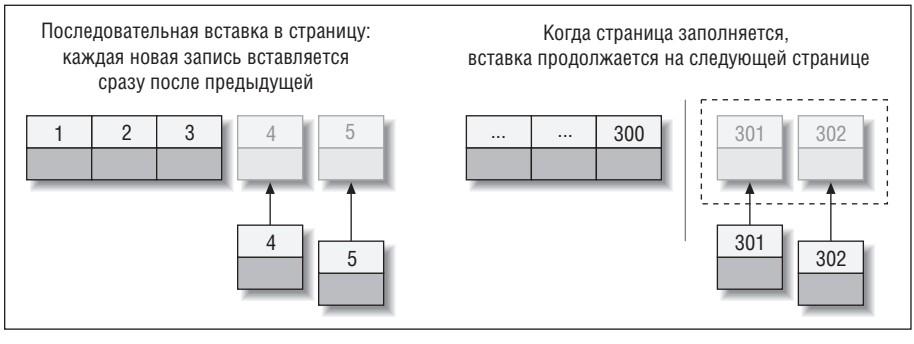


Рис. 3.10. *Вставка последовательных значений индекса в кластерный индекс*

Как видно из рис. 3.10, InnoDB сохраняет новую запись непосредственно после предыдущей, поскольку значения первичного ключа являются последовательными. Когда коэффициент заполнения страницы достигает максимально допустимого значения (в InnoDB коэффициент первоначального заполнения составляет 15/16, чтобы оставить место для будущих модификаций), следующая запись размещается на новой странице. После окончания такой последовательной загрузки данных страницы оказались почти заполненными упорядоченными записями, что крайне желательно.

Совсем иное происходило, когда мы вставляли данные во вторую таблицу с кластерным индексом по столбцу, содержащему UUID (рис. 3.11).

Поскольку значение первичного ключа в каждой последующей строке не обязательно больше, чем в предыдущей, InnoDB не всегда может разместить новую строку в конце индекса. Ей приходится искать для строки подходящее положение – в среднем где-то посередине уже существующих данных – и освобождать для нее место. Это вызывает большое количество дополнительной работы и приводит к неоптимальному размещению данных. Вот сводка недостатков:

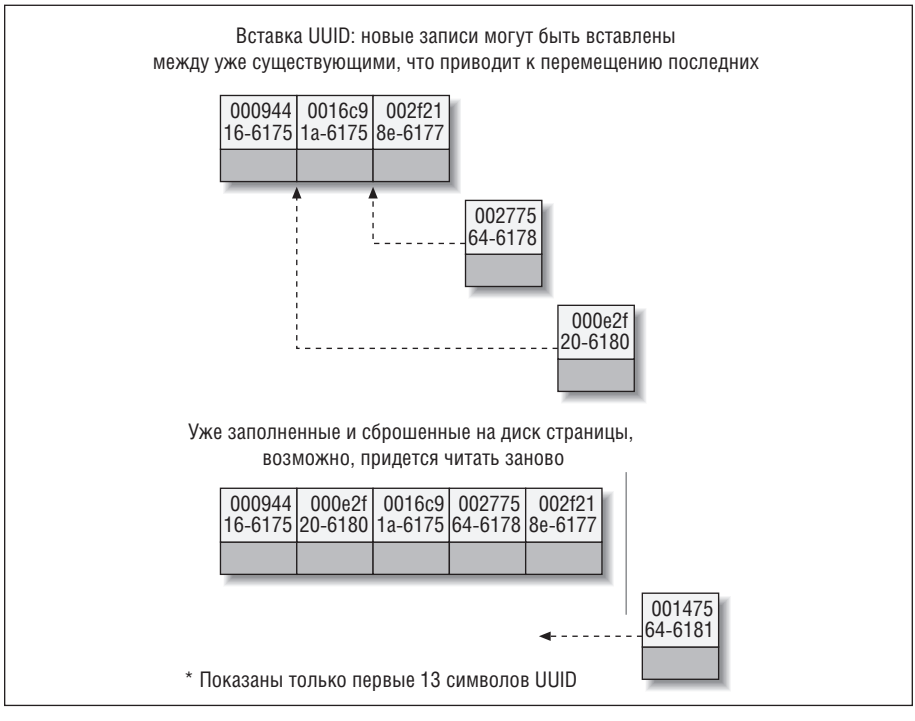


Рис. 3.11. *Вставка непоследовательных значений индекса в кластерный индекс*

- Страница, куда должна попасть строка, может оказаться сброшенной на диск и удаленной из кэша, тогда InnoDB придется искать ее и считывать с диска, прежде чем вставить новую строку. Это приводит к большому количеству случайных операций ввода/вывода.
- InnoDB иногда приходится расщеплять страницы, чтобы освободить место для новых строк. Это требует перемещения большого объема данных.
- Из-за расщепления страницы оказываются заполнены беспорядочно и неплотно, что нередко приводит к фрагментации.

После загрузки таких случайных значений в кластерный индекс имеет смысл запустить команду `OPTIMIZE TABLE`, которая перестроит таблицу и заполнит страницы оптимальным образом.

Моралью всей этой истории является то, что при использовании InnoDB вам нужно стремиться к вставке данных в порядке, соответствующем первичному ключу, и стараться использовать такой кластерный ключ, который монотонно возрастает для новых строк.

Когда вставка в порядке первичного ключа оказывается хуже

При нынешней реализации InnoDB вставка в порядке первичного ключа в условиях высокой конкуренции может создать единственную точку конкуренции. Этой «горячей точкой» является последняя страница первичного индекса. Поскольку все вставки происходят именно здесь, возникает состязание за блокировку следующего ключа и/или блокировку автоинкремента (горячей точкой может оказаться и та, и другая блокировка, а то и сразу обе). Если вы сталкиваетесь с такой проблемой, то можете изменить либо перепроектировать таблицу или приложение, либо настроить InnoDB под такую конкретную нагрузку. О настройке InnoDB читайте в главе 6.

Покрывающие индексы

Индексы являются средством эффективного поиска строк, но MySQL может также использовать индекс для извлечения данных, не считывая строку таблицы. В конце концов, листовые узлы индекса содержат те значения, которые они индексируют. Зачем просматривать саму строку, если чтение индекса уже может дать нужные данные? Индекс, который содержит (или «покрывает») все данные, необходимые для формирования результатов запроса, называется *покрывающим индексом*.

Покрывающие индексы могут служить очень мощным инструментом и значительно увеличивать производительность. Рассмотрим преимущества считывания индекса вместо самих данных.

- Записи индекса обычно компактнее полной строки, поэтому, если MySQL читает только индекс, то обращается к значительно меньшему объему данных. Это очень важно в случае кэшированной рабочей нагрузки, когда время отклика определяется в основном копированием данных. Это также полезно в случае большого количества операций ввода/вывода, поскольку индексы меньше, чем данные, и лучше помещаются в памяти (что особенно справедливо в отношении подсистемы хранения MyISAM, которая может упаковывать индексы, дополнительно уменьшая их размер).
- Индексы отсортированы по индексируемым значениям (по крайней мере, внутри страницы), поэтому для поиска по диапазону, характеризуемому большим объемом ввода/вывода, потребуется меньше операций обращения к диску по сравнению с извлечением каждой строки из произвольного места хранения. Для некоторых подсистем, например MyISAM, вы можете даже оптимизировать (`OPTIMIZE`) таблицу и получить полностью отсортированные индексы, вследствие

чего для простых запросов по диапазону доступ к индексу будет вообще последовательным.

- Большинство подсистем хранения кэширует индексы лучше, чем сами данные (заметным исключением является Falcon). Некоторые подсистемы хранения, например MyISAM, кэшируют в памяти MySQL только индексы. Поскольку кэширование данных для MyISAM выполняет операционная система, доступ к ним обычно требует системного вызова. Это может оказать огромное влияние на производительность, особенно в случае кэшированной рабочей нагрузки, когда системный вызов является самой дорогостоящей частью доступа к данным.
- Покрывающие индексы особенно полезны в случае таблиц InnoDB из-за кластерных индексов. Вторичные индексы InnoDB хранят значения первичного ключа строки в листовых узлах. Таким образом, вторичный индекс, который «покрывает» запрос, позволяет избежать еще одного поиска по первичному индексу.

Во всех этих сценариях обычно значительно дешевле удовлетворить запрос с использованием только индекса вместо того, чтобы извлекать всю запись из таблицы.

Не каждый тип индекса может выступать в роли покрывающего. Индекс должен хранить значения индексируемых столбцов. Хеш-индексы, пространственные индексы и полнотекстовые индексы такие значения не хранят, поэтому MySQL может использовать в качестве покрывающих только B-Tree-индексы. Кроме того, различные подсистемы хранения реализуют покрывающие индексы по-разному, а некоторые не поддерживают их вовсе (на момент написания книги подсистемы Memory и Falcon не поддерживали покрывающие индексы).

Запустив команду `EXPLAIN` для запроса, «покрываемого» индексом, вы увидите в столбце `Extra` сообщение «Using index»[1]. Например, таблица `sakila.inventory` имеет многостолбцовый индекс по (`store_id`, `film_id`). MySQL может использовать этот индекс для ответа на запросы, в которых упоминаются только эти два столбца, например:

```
mysql> EXPLAIN SELECT store_id, film_id FROM sakila.inventory\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: inventory
         type: index
possible_keys: NULL
          key: idx_store_id_film_id
```

[1] Легко спутать сообщение «Using index» в столбце `Extra` с сообщением «index» в столбце `type`. Однако это совершенно разные сообщения. Столбец `type` не имеет никакого отношения к покрывающим индексам. Он показывает тип доступа запроса или поясняет, как запрос будет находить строки.

```
        key_len: 3
            ref: NULL
           rows: 4673
          Extra: Using index
```

«Покрываемые» индексом запросы имеют тонкости, которые могут отключить эту оптимизацию. Оптимизатор MySQL перед выполнением запроса принимает решение, покрывает ли его какой-либо индекс. Предположим, индекс покрывает условие WHERE, но не весь запрос. Даже если условие во фразе WHERE не выполняется, MySQL 5.1 и более ранние версии в любом случае извлекут строку, даже несмотря на то, что она не нужна и будет впоследствии отфильтрована.

Давайте посмотрим, почему это может произойти и как переписать запрос, чтобы обойти данную проблему. Начнем со следующего запроса:

```
mysql> EXPLAIN SELECT * FROM products WHERE actor='SEAN CARREY'
    -> AND title like '%APOLLO%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: products
         type: ref
possible_keys: ACTOR,IX_PROD_ACTOR
          key: ACTOR
      key_len: 52
          ref: const
         rows: 10
        Extra: Using where
```

Индекс не может покрыть этот запрос по двум причинам:

- Ни один индекс не покрывает запрос, поскольку мы выбрали все столбцы из таблицы, а ни один индекс не покрывает все столбцы. Однако есть обходной маневр, который MySQL теоретически может использовать: во фразе WHERE упоминаются только столбцы, которые покрываются индексом, поэтому MySQL может использовать индекс для поиска актера, проверить, соответствует ли заданному критерию название, и только потом считывать всю строку.
- MySQL не может выполнять операцию LIKE в индексе. Это ограничение API подсистемы хранения, которое допускает в операциях с индексами только простые сравнения. MySQL может выполнять сравнения LIKE по префиксу, поскольку допускает преобразовывание их в простые сравнения, однако наличие метасимвола в начале шаблона не позволяет подсистеме хранения провести сопоставление. Таким образом, самому серверу MySQL придется выполнять извлечение и сравнение значений из строки, а не из индекса.

Существует способ обойти обе проблемы путем комбинирования разумного индексирования и переписывания запроса. Мы можем расши-

рить индекс, чтобы он покрывал столбцы (artist, title, prod_id), и переписать запрос следующим образом:

```
mysql> EXPLAIN SELECT *
    -> FROM products
    ->   JOIN (
    ->     SELECT prod_id
    ->     FROM products
    ->     WHERE actor='SEAN CARREY' AND title LIKE '%APOLLO%'
    ->   ) AS t1 ON (t1.prod_id=products.prod_id)\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: <derived2>
          ...пропущено...
*************************** 2. row ***************************
           id: 1
  select_type: PRIMARY
        table: products
          ...пропущено...
*************************** 3. row ***************************
           id: 2
  select_type: DERIVED
        table: products
         type: ref
possible_keys: ACTOR,ACTOR_2,IX_PROD_ACTOR
          key: ACTOR_2
      key_len: 52
          ref:
         rows: 11
        Extra: Using where; Using index
```

Теперь MySQL использует покрывающий индекс на первом этапе запроса, когда ищет строки в подзапросе во фразе FROM. Он не использует индекс для покрытия всего запроса, но это лучше, чем ничего.

Эффективность такой оптимизации зависит от того, сколько строк отвечает условию WHERE. Предположим, таблица products содержит миллион строк. Давайте посмотрим, как эти два запроса выполняются с тремя различными наборами данных, каждый из которых содержит миллион строк:

1. В первом примере в 30 000 записей в столбце actor указан Sean Carrey, а 20 000 из них содержат Apollo в столбце title.
2. Во втором примере в 30 000 записей в столбце actor указан Sean Carrey, а 40 из них содержат Apollo в столбце title.
3. В третьем примере в 50 записях в столбце actor указан Sean Carrey, а 10 из них содержат Apollo в столбце title.

Мы использовали эти три набора данных для эталонного тестирования обоих вариантов запроса и получили результаты, показанные в табл. 3.3.

Таблица 3.3. Результаты тестирования для запросов, покрываемых и не покрываемых индексами

Набор данных	Исходный запрос, запросов в секунду	Оптимизированный запрос, запросов в секунду
Пример 1	5	5
Пример 2	7	35
Пример 3	2400	2000

Интерпретировать эти результаты нужно следующим образом:

- В примере 1 запрос возвращает большой результирующий набор, поэтому мы не видим эффекта от оптимизации. Большая часть времени тратится на считывание и отправку данных.
- В примере 2, где второе условие фильтрации оставляет только небольшой результирующий набор, видно, насколько эффективна предложенная оптимизация: производительность возрастает в пять раз. Эффективность достигается за счет того, что считывается всего 40 полных строк вместо 30 000 в первом примере.
- Пример 3 демонстрирует случай, когда подзапрос оказывается неэффективным. Оставшийся после фильтрации по индексу набор результатов так мал, что подзапрос оказывается дороже, чем считывание всех данных из таблицы.

Эта оптимизация иногда становится эффективным способом избежать считывания ненужных строк в MySQL 5.1 и более ранних версиях. СУБД MySQL 6.0 сама умеет избегать этой дополнительной работы, поэтому при обновлении до указанной версии вы сможете упростить свои запросы.

В большинстве подсистем хранения индекс может «покрывать» только запросы, которые обращаются к столбцам, являющимся частью индекса. Однако InnoDB позволяет немного развить эту оптимизацию. Вспомните, что вторичные индексы InnoDB хранят в листовых узлах значения первичного ключа. Это означает, что вторичные индексы имеют «дополнительные столбцы», которые можно использовать для «покрытия» запросов.

Например, по столбцу `last_name` таблицы `sakila.actor` типа InnoDB построен индекс, который может покрывать запросы, извлекающие столбец первичного ключа `actor_id`, хотя этот столбец технически не является частью индекса:

```
mysql> EXPLAIN SELECT actor_id, last_name
    -> FROM sakila.actor WHERE last_name = 'HOPPER'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: ref
```

```
possible_keys: idx_actor_last_name
          key: idx_actor_last_name
      key_len: 137
          ref: const
         rows: 2
        Extra: Using where; Using index
```

Использование просмотра индекса для сортировки

В СУБД MySQL есть два способа получения отсортированных результатов: использовать файловую сортировку или просматривать индекс по порядку[1]. О том, что MySQL собирается просматривать индекс, можно узнать, обнаружив слово «index» в столбце `type` таблицы, которую выдает команда `EXPLAIN` (не путайте с сообщением «Using index» в столбце `Extra`).

Просмотр самого индекса производится быстро, поскольку сводится просто к перемещению от одной записи к другой. Однако если СУБД MySQL не использует индекс для «покрытия» запроса, ей приходится считывать каждую строку, которую она находит в индексе. В основном это операции ввода/вывода с произвольным доступом, поэтому чтение данных в порядке, соответствующем индексу, обычно происходит значительно медленнее, чем последовательный просмотр таблицы, особенно если рабочая нагрузка характеризуется большим объемом ввода/вывода.

MySQL может использовать один и тот же индекс как для сортировки, так и для поиска строк. По возможности старайтесь проектировать индексы так, чтобы они были полезны для решения обеих задач.

Сортировка результатов по индексу работает только в тех случаях, когда порядок элементов в точности соответствует порядку, указанному во фразе `ORDER BY`, а все столбцы отсортированы в одном направлении (по возрастанию или по убыванию). Если в запросе соединяется несколько таблиц, то необходимо, чтобы во фразе `ORDER BY` упоминались только столбцы из первой таблицы. Фраза `ORDER BY` имеет те же ограничения, что и поисковые запросы: должен быть указан самый левый префикс ключа. Во всех остальных случаях MySQL использует файловую сортировку.

Есть один случай, когда во фразе `ORDER BY` не обязательно должен быть указан левый префикс ключа: если для начальных столбцов индекса в параметрах `WHERE` или `JOIN` заданы константы – они могут заполнить пропуски при сканировании индекса.

Например, таблица `rental` в демонстрационной тестовой базе данных Sakila имеет индекс по столбцам (`rental_date`, `inventory_id`, `customer_id`):

[1] **В MySQL есть два алгоритма файловой сортировки. Подробности см. в разделе** «Оптимизация сортировки» главы 4 на стр. 229.

```
CREATE TABLE rental (
    ...
    PRIMARY KEY (rental_id),
    UNIQUE KEY rental_date (rental_date,inventory_id,customer_id),
    KEY idx_fk_inventory_id (inventory_id),
    KEY idx_fk_customer_id (customer_id),
    KEY idx_fk_staff_id (staff_id),
    ...
);
```

Для сортировки результатов в следующем запросе MySQL использует индекс `rental_date`, что доказывает отсутствие упоминания о файловой сортировке в команде `EXPLAIN`:

```
mysql> EXPLAIN SELECT rental_id, staff_id FROM sakila.rental
    -> WHERE rental_date = '2005-05-25'
    -> ORDER BY inventory_id, customer_id\G
*************************** 1. row ***************************
          type: ref
possible_keys: rental_date
           key: rental_date
          rows: 1
         Extra: Using where
```

Это работает даже несмотря на то, что во фразе `ORDER BY` указан не левый префикс индекса, поскольку для первого столбца в индексе задано ограничение по равенству.

Приведем еще несколько запросов, в которых для сортировки можно использовать индекс. В следующем примере это возможно, потому что для первого столбца индекса в запросе задана константа, в `ORDER BY` производится сортировка по второму столбцу. В совокупности мы получаем левый префикс ключа:

```
... WHERE rental_date = '2005-05-25' ORDER BY inventory_id DESC;
```

Следующий запрос также работает, поскольку два столбца, указанных во фразе `ORDER BY`, образуют левый префикс ключа:

```
... WHERE rental_date > '2005-05-25' ORDER BY rental_date, inventory_id;
```

А вот запросы, которые *не могут* использовать индекс для сортировки:

- Здесь указано два разных направления сортировки, но все столбцы индекса отсортированы по возрастанию:

```
... WHERE rental_date = '2005-05-25' ORDER BY inventory_id DESC, customer_id ASC;
```

- Здесь во фразе `ORDER BY` указан столбец, отсутствующий в индексе:

```
... WHERE rental_date = '2005-05-25' ORDER BY inventory_id, staff_id;
```

- Здесь столбцы, заданные во фразах WHERE и ORDER BY, не образуют левый префикс ключа:

```
... WHERE rental_date = '2005-05-25' ORDER BY customer_id;
```

- Этот запрос содержит в первом столбце условие поиска по диапазону, поэтому MySQL не использует оставшуюся часть индекса:

```
... WHERE rental_date > '2005-05-25' ORDER BY inventory_id, customer_id;
```

- Здесь для столбца inventory_id есть несколько сравнений на равенство. С точки зрения сортировки, это, в сущности, то же самое, что и условие поиска по диапазону:

```
... WHERE rental_date = '2005-05-25' AND inventory_id IN(1,2) ORDER BY customer_id;
```

- Вот пример, где MySQL теоретически может использовать индекс для сортировки соединения, но не делает этого, поскольку оптимизатор помещает таблицу film_actor в соединение второй по счету (в главе 4 рассказано, как можно изменить порядок соединения):

```
mysql> EXPLAIN SELECT actor_id, title FROM sakila.film_actor
    -> INNER JOIN sakila.film USING(film_id) ORDER BY actor_id\G
+------------+----------------------------------------------+
| table      | Extra                                        |
+------------+----------------------------------------------+
| film       | Using index; Using temporary; Using filesort |
| film_actor | Using index                                  |
+------------+----------------------------------------------+
```

Одним из самых важных применений сортировки по индексу является запрос, в котором есть и фраза `ORDER BY`, и фраза `LIMIT`. Далее мы остановимся на этом вопросе более подробно.

Упакованные (сжатые по префиксу) индексы

MyISAM использует префиксное сжатие для уменьшения размера индекса, обеспечивая таким образом размещение большей части индекса в памяти и значительное увеличение производительности в некоторых случаях. По умолчанию эта подсистема хранения упаковывает только строковые значения, но вы можете затребовать также сжатие целочисленных значений.

Для сжатия блока индекса MyISAM сохраняет первое значение полностью, а при сохранении каждого последующего значения в том же блоке записывает только количество байтов, совпадающих с частью префикса, плюс отличающиеся данные суффикса. Например, если первым значением является слово «perform», а вторым – «performance», то второе значение будет записано как «7,ance». MyISAM может также осуществлять префиксное сжатие смежных указателей на строки.

Сжатые блоки занимают меньше места, но замедляют некоторые операции. Поскольку сжатый префикс каждого значения зависит от значения, предшествующего ему, MyISAM не может выполнить двоичный поиск для нахождения нужного элемента и вынужден просматривать блок с самого начала. Последовательный просмотр в прямом направлении выполняется быстро, но просмотр в обратном направлении – на-

пример, в случае ORDER BY DESC – работает хуже. Любая операция, которая требует нахождения строки в середине блока, приводит к просмотру в среднем половины данных.

Наши тесты показали, что при большой загрузке процессора упакованные ключи замедляют поиск по индексу в таблицах MyISAM в несколько раз из-за дополнитеного сканирования, требуемого для выполнения произвольного поиска. Поиск упакованного ключа в обратном направлении происходит еще медленнее. Приходится искать компромисс между использованием процессора и памяти с одной стороны, и дисковых ресурсов с другой. Упаковка индексов может уменьшить их дисковый размер на порядок, поэтому в случае рабочей нагрузки с большим объемом ввода/вывода это часто дает ощутимый эффект для некоторых запросов.

Вы можете управлять упаковкой индексов в таблице с помощью параметра PACK_KEYS команды CREATE TABLE.

Избыточные и дублирующие индексы

MySQL позволяет создавать несколько индексов по одному и тому же столбцу. Она не «замечает» и не защищает вас от таких ошибок. СУБД MySQL должна обслуживать каждый дублирующий индекс отдельно, а оптимизатор запросов в своей работе будет учитывать их все. Это может вызвать серьезное падение производительности.

Дублирующими являются индексы одного типа, созданные на основе того же набора столбцов в одинаковом порядке. Старайтесь избегать их создания, а если найдете – удаляйте.

Иногда можно создать дублирующие индексы, даже не ведая о том. Например, взгляните на следующий код:

```
CREATE TABLE test (
    ID INT NOT NULL PRIMARY KEY,
    UNIQUE(ID),
    INDEX(ID)
);
```

Неопытный пользователь может подумать, что здесь столбец описан как первичный ключ, добавлено ограничение UNIQUE и, кроме того, создан индекс для использования в запросах. На самом деле, MySQL реализует ограничения UNIQUE и PRIMARY KEY с помощью индексов, так что на деле создаются три индекса по одному и тому же столбцу! Обычно нет причин для таких действий, за исключением случаев, когда вы хотите иметь различные типы индексов по одному столбцу для выполнения различных типов запросов[1].

[1] Индекс не обязательно является дублирующим, если это индекс другого типа. Часто есть веские причины для одновременного наличия индексов KEY(col) и FULLTEXT KEY(col).

Избыточные индексы несколько отличаются от дублирующих. Если существует индекс по паре столбцов (A, B), то отдельный индекс по столбцу A будет избыточным, поскольку он является префиксом первого. То есть, индекс по (A, B) может быть также использован как индекс по столбцу A (такой тип избыточности относится только к B-Tree-индексам). Однако ни индекс по столбцам (B, A), ни индекс по столбцу B не будет избыточным, поскольку B не является левым префиксом для (A, B). Более того, индексы различных типов (например, хеш-индексы или полнотекстовые индексы) не являются избыточными по отношению к B-Tree-индексам вне зависимости от того, какие столбцы они покрывают.

Избыточность обычно появляется при добавлении индексов к таблице. Например, кто-то может добавить индекс по (A, B) вместо расширения существующего индекса по столбцу A для покрытия (A, B).

В большинстве случаев избыточные индексы не нужны, и для того чтобы избежать их появления, лучше расширять существующие индексы, а не добавлять новые. Однако бывают случаи, когда избыточные индексы необходимы по причинам, связанным с производительностью. Расширение существующего индекса может значительно увеличить его размер и привести к падению производительности некоторых запросов.

Например, если у вас есть индекс по целочисленному столбцу, и вы расширяете его длинным столбцом типа VARCHAR, он может стать значительно более медленным. Особенно это относится к случаям, когда ваши запросы используют покрывающий индекс, или если это таблица MyISAM, и вы осуществляете много просмотров по диапазону (из-за сжатия префикса в MyISAM).

Вспомните таблицу userinfo, описанную в разделе «Вставка строк в порядке первичного ключа в InnoDB» выше в этой главе на стр. 158. Эта таблица содержит миллион строк и для каждого значения столбца state_id имеется примерно 20 000 записей. Имеется индекс по столбцу state_id, который полезен в следующем запросе. Мы назовем этот запрос Q1:

```
mysql> SELECT count(*) FROM userinfo WHERE state_id=5;
```

Простой эталонный тест показывает для этого запроса скорость выполнения почти 115 запросов в секунду (QPS). Рассмотрим также запрос Q2, который извлекает несколько столбцов, а не просто подсчитывает строки:

```
mysql> SELECT state_id, city, address FROM userinfo WHERE state_id=5;
```

Для этого запроса результат меньше 10 QPS[1]. Простым решением для увеличения производительности является расширение индекса до (state_id, city, address), чтобы индекс покрывал запрос:

[1] В данном случае все данные помещаются в память. Если размер таблицы велик и рабочая нагрузка характеризуется большим объемом ввода/вывода, то различие станет гораздо заметнее.

```
mysql> ALTER TABLE userinfo DROP KEY state_id,
    -> ADD KEY state_id_2 (state_id, city, address);
```

После расширения индекса запрос Q2 выполняется быстрее, но запрос Q1 становится медленнее. Если мы хотим сделать эти два запроса быстрыми, нам нужно оставить оба индекса, даже несмотря на то, что индекс по одному столбцу является избыточным. В табл. 3.4 показаны подробные результаты для обоих запросов и стратегий индексирования с подсистемами хранения MyISAM и InnoDB. Обратите внимание, что в случае InnoDB производительность запроса Q1 падает не так сильно только с индексом `state_id_2`, поскольку в InnoDB не используется сжатие ключа.

Таблица 3.4. Результаты эталонного тестирования для запросов SELECT при различных стратегиях индексирования

	Только state_id	Только state_id_2	И state_id, и state_id_2
MyISAM, Q1	114,96	25,40	112,19
MyISAM, Q2	9,97	16,34	16,37
InnoDB, Q1	108,55	100,33	107,97
InnoDB, Q2	12,12	28,04	28,06

Недостатком наличия двух индексов является стоимость их поддержки. В табл. 3.5 показано, сколько времени занимает вставка миллиона строк в таблицу.

Таблица 3.5. Скорость вставки миллиона строк при различных стратегиях индексирования, сек

	Только state_id	И state_id, и state_id_2
InnoDB, памяти достаточно для обоих индексов	80	136
MyISAM, памяти хватает только для одного индекса	72	470

Как видите, вставка новых строк в таблицу с большим количеством индексов происходит значительно медленнее. Это общее правило: добавление новых индексов может серьезно повлиять на производительность операций `INSERT`, `UPDATE` и `DELETE`, особенно если новый индекс не помещается в памяти.

Индексы и блокировки

Индексы играют очень важную роль в InnoDB, поскольку позволяют блокировать меньше строк при выполнении запроса. Это существенно, так как в версии MySQL 5.0 подсистема хранения InnoDB не разблокирует строку до фиксации транзакции.

Если запрос не обращается к строкам, которые ему не нужны, то блокируется меньше строк, и это лучше для производительности по двум причинам. Во-первых, даже несмотря на то, что блокировки строк в InnoDB реализованы весьма эффективно и потребляют очень мало памяти, все же с ними сопряжены некоторые накладные расходы. Во-вторых, блокировка большего количества строк, чем необходимо, увеличивает конкуренцию за блокировки и уменьшает степень параллелизма.

InnoDB блокирует строки только в момент доступа к ним, а индекс позволяет уменьшить количество строк, к которым обращается InnoDB, и, следовательно, блокирует их. Однако это работает только в том случае, когда InnoDB может отфильтровывать ненужные строки на *уровне подсистемы хранения*. Если индекс не позволяет InnoDB сделать это, то сервер MySQL вынужден применять фразу `WHERE` после того, как InnoDB извлечет строки и вернет их серверу. К этому моменту уже невозможно избежать блокировки строк: они заблокированы InnoDB и сервер уже не может их разблокировать.

Это проще понять на примере. Мы снова используем демонстрационную базу данных Sakila:

```
mysql> SET AUTOCOMMIT=0;
mysql> BEGIN;
mysql> SELECT actor_id FROM sakila.actor WHERE actor_id < 5
    -> AND actor_id <> 1 FOR UPDATE;
+----------+
| actor_id |
+----------+
| 2        |
| 3        |
| 4        |
+----------+
```

Этот запрос возвращает только строки со второй по четвертую, но на деле он захватывает монопольные блокировки на строки с первой по четвертую. InnoDB блокирует первую строку потому, что план, который оптимизатор выбрал для этого запроса, предусматривает поиск по диапазону индекса:

```
mysql> EXPLAIN SELECT actor_id FROM sakila.actor
    -> WHERE actor_id < 5 AND actor_id <> 1 FOR UPDATE;
+----+-------------+-------+-------+---------+--------------------------+
| id | select_type | table | type  | key     | Extra                    |
+----+-------------+-------+-------+---------+--------------------------+
|  1 | SIMPLE      | actor | range | PRIMARY | Using where; Using index |
+----+-------------+-------+-------+---------+--------------------------+
```

Другими словами, низкоуровневая операция подсистемы хранения такова: «начать с первой строки индекса и извлекать все строки до тех пор, пока выражение `actor_id < 5` остается истинным». Сервер не сообщил подсистеме InnoDB об условии `WHERE`, которое отбрасывает первую строку. Обратите внимание на наличие сообщения «Using where» в столбце

Extra вывода команды EXPLAIN. Это указывает на то, что сервер MySQL применяет фильтр WHERE после того, как подсистема хранения вернула строки.

Резюме стратегий индексирования

Теперь, больше узнав об индексировании, вы, возможно, захотите понять, с чего начинать работу со своими собственными таблицами. Самой важной задачей является проверка запросов, которые планируется запускать чаще всего, но нужно еще подумать и о сравнительно редко выполняемых операциях, например о вставке и обновлении данных. Постарайтесь избежать распространенной ошибки – создавать индексы без знания того, какие запросы будут их использовать. Также имеет смысл задаться вопросом, образуют ли все созданные вами индексы оптимальную конфигурацию.

Иногда достаточно одного взгляда на запрос, чтобы понять, какие индексы потребуются для его обработки. Но иногда у вас будут запросы различных типов, и вы не сможете добавить оптимальные индексы для каждого из них. Тогда вам потребуется идти на компромисс. Чтобы найти лучший баланс, вам стоит провести эталонное тестирование и профилирование.

В первую очередь надо выяснить время отклика. Подумайте о добавлении индекса для запросов, которые выполняются слишком долго. Затем найдите запросы, вызывающие наибольшую нагрузку (о том, как эту нагрузку измерить, см. главу 2), и добавьте индексы для них. Если в вашей системе память, процессор или дисковые операции становятся узким местом, примите это во внимание. Например, если вы запускаете много длинных агрегирующих запросов для формирования сводных отчетов, то диску станет легче от наличия покрывающих индексов, которые поддерживают запросы с фразой GROUP BY.

Где это возможно, старайтесь расширять существующие индексы, а не добавлять новые. Обычно эффективнее поддерживать один многостолбцовый индекс, чем несколько одностолбцовых. Если вы еще не знаете распределения своих запросов, старайтесь сделать индексы максимально селективными, поскольку они дают больший выигрыш.

Вот второй запрос, который доказывает, что первая строка заблокирована, несмотря на то, что она включена в результаты первого запроса. Оставив первое соединение открытым, запустите второе соединение и выполните такой запрос:

```
mysql> SET AUTOCOMMIT=0;
mysql> BEGIN;
mysql> SELECT actor_id FROM sakila.actor WHERE actor_id = 1 FOR UPDATE;
```

Запрос зависнет в ожидании момента, когда предыдущая транзакция освободит блокировку первой строки. Это поведение необходимо для того, чтобы покомандная репликация (см. главу 8) работала правильно.

Как показывает данный пример, InnoDB может блокировать строки, которые ей на деле не нужны, даже при использовании индекса. Все становится еще хуже, когда InnoDB не может использовать индекс для поиска и блокировки строк: если для запроса нет индекса, MySQL придется выполнять полное сканирование таблицы и блокировать каждую строку, вне зависимости от того, нужна она или нет[1].

Есть малоизвестная подробность об InnoDB, индексах и блокировке: InnoDB может захватывать разделяемые блокировки на вторичные индексы (на чтение), но для доступа к первичному ключу необходимы монопольные блокировки (на запись). Это исключает возможность использования покрывающего индекса и может сделать команду `SELECT FOR UPDATE` значительно более медленной, чем `LOCK IN SHARE MODE` или неблокирующий запрос.

Практические примеры индексирования

Проще всего понять концепции индексирования на практических примерах, поэтому мы рассмотрим несколько типичных ситуаций.

Предположим, нам нужно спроектировать интерактивный сайт знакомств с профилями пользователей, в которые включены различные столбцы, например: страна, регион, город, пол, возраст, цвет глаз и т. п. Сайт должен поддерживать поиск в профилях по различным комбинациям этих свойств. Он также должен позволять пользователю сортировать и фильтровать результаты по времени последнего посещения сайта владельцем профиля, по оценкам его другими пользователями и т. д. Каким образом спроектировать индексы для таких сложных требований?

Как ни странно, первое, что мы должны решить, будем ли мы использовать сортировку с помощью индексов или подойдет обычная сортировка (filesort) . Сортировка с помощью индексов налагает ограничения на то, как должны быть построены индексы и запросы. Например, мы не можем использовать индекс для фразы `WHERE age BETWEEN 18 AND 25`, если в том же самом запросе индекс используется для сортировки пользова-

[1] Предполагалось, что это будет исправлено в версии MySQL 5.1 с помощью ведения построчного двоичного журнала и уровня изоляции `READ COMMITTED`, но ситуация осталась такой же во всех протестированных нами версиях MySQL, включая 5.1.22.

телей по оценкам, которые дают им другие пользователи. Если СУБД MySQL задействует в запросе индекс для поиска по диапазону, то она не может использовать другой индекс (или суффикс того же самого индекса) с целью упорядочивания. Учитывая, что это будет одна из самых распространенных фраз `WHERE`, мы считаем само собой разумеющимся, что для многих запросов потребуется обычная сортировка (filesort).

Поддержка нескольких видов фильтрации

Теперь нам нужно посмотреть, какие столбцы содержат много различных значений, и какие столбцы появляются во фразах `WHERE` чаще всего. Индексы по столбцам с большим количеством различных значений будут высокоселективны. Обычно это хорошо, поскольку высокая селективность позволяет MySQL более эффективно отфильтровывать ненужные строки.

Столбец `country` может оказаться селективным, а может и нет, но в любом случае он, скорее всего, будет появляться в большинстве запросов. Селективность столбца `sex`, несомненно, низка, но он, вероятно, будет присутствовать в любом запросе. Имея это в виду, мы создаем набор индексов для нескольких различных сочетаний столбцов с префиксом по `(sex, country)`.

Известно, что бесполезно индексировать столбцы с очень низкой селективностью. Так зачем мы помещаем неселективный столбец в начале каждого индекса? Мы сошли с ума?

Для такого выбора у нас есть две причины. Первая заключается в том, что, как уже указывалось ранее, почти в каждом запросе будет использоваться столбец `sex`. При проектировании сайта мы можем даже сделать так, что пользователи будут выполнять поиск, только включив пол в состав критериев. Но что важнее, добавление этого столбца не принесет особых потерь, поскольку у нас есть в запасе один фокус.

Вот в чем он заключается: даже если запрос не выполняет фильтрации по столбцу `sex`, мы можем обеспечить использование этого индекса, добавив во фразу `WHERE` выражение `AND sex IN('m', 'f')`. На деле это выражение не будет осуществлять никакой фильтрации строк, поэтому функциональность запроса будет точно такой же, что и при отсутствии столбца `sex` во фразе `WHERE`. Однако нам *нужно* включить этот столбец, поскольку это позволит MySQL использовать больший префикс индекса. Этот прием полезен в ситуациях, подобных вышеописанной, но если столбец содержит много различных значений, это будет работать плохо, поскольку список `IN()` станет слишком большим.

Этот случай иллюстрирует общий принцип: учитывайте все факторы. Разрабатывая индексы, думайте не только об их оптимизации под существующие запросы, но также об оптимизации запросов под эти индексы. Если вы видите необходимость в индексе, но считаете, что он мо-

жет оказать негативное влияние на некоторые запросы, спросите себя: нельзя ли изменить запросы? Оптимизируйте запросы и индексы одновременно, чтобы найти наилучший компромисс. Нельзя спроектировать хорошую схему индексирования в вакууме.

Теперь нужно подумать о том, какие еще могут встретиться комбинации условий `WHERE`, и решить, не будет ли часть из них слишком медленной без правильного индексирования. Индекс по столбцам (`sex, country, age`) является очевидным выбором и, скорее всего, потребуются индексы по столбцам (`sex, country, region, age`) и (`sex, country, region, city, age`).

Таким образом, у нас появляется очень много индексов. Если мы хотим использовать одни и те же индексы в запросах разного типа и не создавать большое число комбинаций разных условий, то можем использовать вышеупоянутый прием с `IN()` и избавиться от индексов по столбцам (`sex, country, age`) и (`sex, country, region, age`). Если они не указаны в форме поиска, мы можем сделать так, чтобы на префиксные стобцы индекса налагались ограничения равенства, указав список всех стран или всех регионов страны (объединение списков всех стран, всех регионов и всех полов, вероятно, окажется слишком большим).

Эти индексы будут полезны для наиболее часто задаваемых критериев поиска, но как разработать индексы для реже встречающихся вариантов, например `has_pictures`, `eye_color`, `hair_color` и `education`? Если эти столбцы не являются высокоселективными и используются нечасто, мы можем просто пропустить их и позволить MySQL сканировать некоторое количество дополнительных строк. В качестве альтернативы можно добавить их перед столбцом `age` и использовать вышеописанный прием с `IN()`, чтобы обрабатывать случай, когда они не указаны.

Наверное, вы обратили внимание, что мы помещаем столбец `age` в конец индекса. Что особенного в этом столбце и почему он должен располагаться именно там? Мы хотим, чтобы MySQL использовал как можно больше столбцов индекса, поскольку он задействует только левый префикс вплоть до первого условия, задающего диапазон значений, включительно. Для всех остальных упомянутых нами столбцов во фразе `WHERE` задается сравнение на равенство, но для `age` почти наверняка понадобится поиск по диапазону (например, `age BETWEEN 18 AND 25`).

Мы могли бы преобразовать это условие в список `IN()`, например `age IN(18, 19, 20, 21, 22, 23, 24, 25)`, но для запросов такого типа это не всегда возможно. Общий принцип, который мы пытаемся проиллюстрировать, заключается в том, чтобы помещать столбец, для которого может быть задано условие поиска по диапазону, в конец индекса, тогда оптимизатор будет использовать индекс максимально полно.

Мы уже говорили, что можно увеличивать количество столбцов в индексе и использовать списки `IN()` в случаях, когда эти столбцы не являются частью фразы `WHERE`, но есть вероятность перестараться и нарвать-

ся на проблемы. Использование слишком большого количества списков вызывает лавинообразный рост числа комбинаций, которые оптимизатор должен оценить, и может значительно снизить скорость выполнения запроса. Взгляните на следующую фразу WHERE:

```
WHERE eye_color IN('brown','blue','hazel')
    AND hair_color IN('black','red','blonde','brown')
    AND sex IN('M','F')
```

Оптимизатор преобразует это в $4 \times 3 \times 2 = 24$ комбинации, и фразе WHERE придется проверить каждую из них. 24 комбинации – это не слишком много, но если их количество исчисляется тысячами, возникнут проблемы. В старых версиях MySQL были серьезные сложности с большим числом комбинаций IN(): оптимизация запроса занимала больше времени, чем его выполнение, и требовала большого объема памяти. Последние версии MySQL прекращают вычисление комбинаций, если их количество становится слишком велико, но это обстоятельство ограничивает использование индекса в MySQL.

Устранение дополнительных условий поиска по диапазону

Предположим, имеется столбец last_online, и мы хотим показывать тех пользователей, которые заходили на сайт в течение предыдущей недели:

```
WHERE eye_color IN('brown','blue','hazel')
    AND hair_color IN('black','red','blonde','brown')
    AND sex IN('M','F')
    AND last_online > DATE_SUB('2008-01-17', INTERVAL 7 DAY)
    AND age BETWEEN 18 AND 25
```

С этим запросом у нас будет проблема: в нем есть два условия на вхождение в диапазон. MySQL может использовать либо критерий по столбцу last_online, либо критерий по столбцу age, но не оба сразу.

Если ограничение по столбцу last_online появляется без ограничения по столбцу age, или если столбец last_online является более селективным, чем столбец age, то мы можем добавить другой набор индексов со столбцом last_online в конец. Но если мы не можем преобразовать столбец age в список IN(), а нам необходимо увеличение скорости при одновременной фильтрации по столбцам last_online и age? На данный момент не существует простого способа сделать это, однако мы можем преобразовать один из диапазонов в сравнение на равенство. Для этого мы добавим столбец active, значения которого будут вычисляться периодически запускаемым заданием. Когда пользователь заходит на сайт, мы записываем в столбец значение 1, а если пользователь не появлялся на сайте в течение семи дней, то периодическое задание будет присваивать ему значение 0.

Что такое условие поиска по диапазону?

Из вывода команды EXPLAIN иногда трудно понять, то ли MySQL просматривает диапазон значений, то ли список значений. EXPLAIN использует один и тот же термин «range» для обозначения обоих случаев. Например, MySQL указывает для следующего запроса тип «range», как видно из строки type:

```
mysql> EXPLAIN SELECT actor_id FROM sakila.actor
    -> WHERE actor_id > 45\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: range
```

А как вам такой запрос?

```
mysql> EXPLAIN SELECT actor_id FROM sakila.actor
    -> WHERE actor_id IN(1, 4, 99)\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: range
```

Глядя на вывод команды EXPLAIN, невозможно увидеть разницу, но мы различаем поиск по диапазону и множественные условия на равенство. В нашей терминологии второй запрос представляет собой множественные условия на равенство.

Это не придирка: упомянутые два типа доступа к индексам выполняются по-разному. Условие на вхождение в диапазон заставляет MySQL игнорировать все дальнейшие столбцы в индексе, а условие множественного равенства не налагает таких ограничений.

Подобный подход позволяет MySQL использовать такие индексы, как (active, sex, country, age). Столбец может оказаться не совсем точным, но запросу такого типа высокая точность и не нужна. Если нам требуется точность, мы можем оставить условие last_online во фразе WHERE, *но не индексировать этот столбец*. Данный прием подобен тому, который мы использовали для эмулирования хеш-индексов в процессе поиска адресов URL выше в этой главе. Для фильтрации по этому условию не используется индекс, но, поскольку маловероятно, что будет отброшено много найденных по данному индексу строк, его наличие все равно не дало бы ощутимого выигрыша. Иными словами, от отсутствия этого индекса производительность запроса не пострадает.

Теперь вы, вероятно, поняли суть: если пользователь хочет видеть и активных, и неактивных пользователей, мы можем добавить список IN().

Мы присоединили много таких списков, но в качестве альтернативы можно создавать отдельные индексы, способные обслужить любую комбинацию столбцов, по которым мы хотим осуществлять фильтрацию. Имеет смысл использовать, по крайней мере, следующие индексы: (active, sex, country, age), (active, country, age), (sex, country, age) и (country, age). Хотя такие индексы порой оказываются более эффективными для каждого конкретного запроса, накладные расходы на обслуживание их всех в сочетании с требуемым ими дополнительным пространством делают подобную стратегию сомнительной.

Это тот случай, когда модификация оптимизатора может реально повлиять на выбор эффективной стратегии индексирования. Если в будущей версии MySQL научится выполнять непоследовательный просмотр индекса, станет возможно использовать несколько условий поиска по диапазону в одном индексе. Тогда нам не потребуются списки IN() для рассмотренных здесь типов запросов.

Оптимизация сортировки

Последний вопрос, которого мы хотим коснуться в этом разделе, – сортировка. Сортировка без использования индекса (filesort) небольшого результирующего набора происходит быстро, но что, если запрос находит миллионы строк? Например, что будет, если во фразе WHERE будет присутствовать только столбец sex?

Мы можем добавить специальные индексы для таких случаев с низкой селективностью. Например, индекс по столбцам (sex, rating) можно использовать для следующего запроса:

```
mysql> SELECT <cols> FROM profiles WHERE sex='M' ORDER BY rating LIMIT 10;
```

Этот запрос содержит и фразу ORDER BY, и фразу LIMIT, и без индекса он будет работать очень медленно.

Даже с индексом запрос может оказаться медленным, если интерфейс пользователя предусматривает разбиение на страницы и кто-то запрашивает страницу, находящуюся далеко от начала. Вот пример неудачного сочетания фраз ORDER BY и LIMIT со смещением:

```
mysql> SELECT <cols> FROM profiles WHERE sex='M' ORDER BY
rating LIMIT 100000, 10;
```

Такие запросы могут оказаться серьезной проблемой вне зависимости от того, как они проиндексированы, поскольку из-за большого смещения они должны тратить основное время на просмотр значительного количества строк, которые потом будут отброшены. Денормализация, предварительное вычисление и кэширование являются, вероятно, единственными стратегиями, полезными для обработки подобных запросов. Еще лучшей стратегией можно назвать ограничение количества страниц, которые вы позволяете просматривать. Это вряд ли по-

влияет на возможности пользователя, поскольку никто не станет открывать десятитысячную страницу результатов поиска.

Еще одной хорошей стратегией оптимизации подобных запросов является использование покрывающего индекса с целью извлекать только столбцы первичного ключа тех строк, которые нужны в конечном итоге. Затем вы можете снова соединить их с таблицей для извлечения всех нужных столбцов. Это помогает минимизировать объем работы, выполняемой СУБД MySQL в процессе сбора данных, которые она затем отбросит. Вот пример, требующий для большей эффективности наличия индекса по столбцам (sex, rating):

```
mysql> SELECT <cols> FROM profiles INNER JOIN (
    ->   SELECT <primary key cols> FROM profiles
    ->   WHERE x.sex='M' ORDER BY rating LIMIT 100000, 10
    -> ) AS x USING(<primary key cols>);
```

Обслуживание индексов и таблиц

После того как вы создали таблицы с нужными типами данных и добавили индексы, ваша работа еще не закончена: таблицы и индексы еще требуется обслуживать с целью обеспечения нормальной работы. Тремя главными задачами обслуживания таблицы являются поиск и устранение повреждений, актуализация статистики по индексам и уменьшение фрагментации.

Поиск и исправление повреждений таблицы

Худшее, что может случиться с таблицей, – это ее повреждение. С таблицами MyISAM такое часто происходит при аварийных остановах. Однако повреждения индексов в результате аппаратных сбоев, внутренних программных ошибок MySQL или операционной системы могут возникать во всех подсистемах хранения.

Поврежденные индексы могут привести к тому, что запросы будут возвращать неправильные результаты, вызывать ошибки дублирования ключа, хотя дубликатов нет, и даже приводить к блокировкам и аварийным остановам. Если вы сталкиваетесь со странным поведением – например ошибками, которые, по вашему мнению, происходить просто не могут, – запустите команду CHECK TABLE, чтобы выяснить, не повреждена ли таблица (обратите внимание, что некоторые подсистемы хранения не поддерживают эту команду, а другие поддерживают многочисленные параметры, позволяющие указать, насколько тщательно нужно проверить таблицу). Команда CHECK TABLE обычно выявляет большинство ошибок в таблицах и индексах.

Исправить поврежденную таблицу позволяет команда REPAIR TABLE, но и ее поддерживают не все подсистемы хранения. В таком случае можно выполнить «пустую» команду ALTER, например просто указав ту же

подсистему, которая и так уже используется для таблицы. Вот пример для таблицы типа InnoDB:

```
mysql> ALTER TABLE innodb_tbl ENGINE=INNODB;
```

В качестве альтернативы можно либо использовать утилиту восстановления для данной подсистемы хранения, например *myisamchk*, либо выгрузить все данные в файл, а затем загрузить обратно. Однако если повреждение находится в системной области или в «области строк» таблицы, а не в индексе, все это может оказаться бесполезным. В таком случае, возможно, останется лишь восстановить таблицу из резервной копии или попытаться извлечь какие-то данные из поврежденных файлов (см. главу 11).

Обновление статистики индекса

Для принятия решения о том, как использовать индексы, оптимизатор запросов MySQL использует два вызова API, чтобы узнать у подсистемы хранения, каким образом распределены значения в индексе. Первым является вызов `records_in_range()`, который получает границы диапазона и возвращает (возможно, оценочное) количество записей в этом диапазоне. Второй вызов, `info()`, может возвращать данные различных типов, включая кардинальность (сколько записей имеется для каждого значения ключа).

Если подсистема хранения не сообщает оптимизатору точную информацию о количестве строк, которые должен будет просмотреть запрос, оптимизатор использует для оценки этой величины статистику по индексам. Обновить статистику вы можете с помощью команды `ANALYZE TABLE`. Алгоритмы работы оптимизатора MySQL основаны на вычислении стоимости, а главной метрикой стоимости является объем данных, к которому обращается запрос. Если статистика никогда не собиралась или устарела, оптимизатор может принимать неправильные решения. Выход состоит в запуске команды `ANALYZE TABLE`.

Каждая подсистема хранения по-своему реализует статистику по индексам, поэтому частота сбора статистики, равно как и стоимость запуска команды `ANALYZE TABLE`, различается.

- Подсистема Memory вообще не хранит статистику по индексам.
- MyISAM сохраняет статистику на диске, а команда `ANALYZE TABLE` выполняет полное сканирование индекса для вычисления кардинальности. На время этого процесса вся таблица блокируется.
- InnoDB не сохраняет статистику на диске, а вместо этого оценивает ее с помощью случайной выборки из индекса в момент первого открытия таблицы. Так как команда `ANALYZE TABLE` для InnoDB пользуется случайной выборкой, собираемая статистика InnoDB менее точна, зато ее не требуется обновлять вручную, если только с момента последнего перезапуска сервера прошло не очень много времени. Ко-

манда ANALYZE TABLE в InnoDB не вызывает блокировок и является относительно недорогой, поэтому можно выполнять оперативное обновление статистики, практически не влияя на работу сервера.

Вы можете выяснить кардинальность ваших индексов с помощью команды SHOW INDEX FROM. Например:

```
mysql> SHOW INDEX FROM sakila.actor\G
*************************** 1. row ***************************
       Table: actor
  Non_unique: 0
    Key_name: PRIMARY
Seq_in_index: 1
 Column_name: actor_id
   Collation: A
 Cardinality: 200
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
*************************** 2. row ***************************
       Table: actor
  Non_unique: 1
    Key_name: idx_actor_last_name
Seq_in_index: 1
 Column_name: last_name
   Collation: A
 Cardinality: 200
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
```

Эта команда выдает довольно много информации об индексах, подробно описанной в руководстве по MySQL. Однако мы хотим обратить ваше внимание на строку Cardinality. Она показывает приблизительное число различных значений, обнаруженных подсистемой хранения в индексе. Эти данные вы также можете получить из таблицы INFORMATION_SCHEMA.STATISTICS в MySQL 5.0 и более новых версиях, что может оказаться очень удобным. Например, можно написать запросы к таблицам INFORMATION_SCHEMA, чтобы найти индексы с очень низкой селективностью.

Уменьшение фрагментации индекса и данных

B-Tree-индексы могут становиться фрагментированными, что приводит к уменьшению производительности. Фрагментированные индексы бывают плохо заполнены и/или располагаются на диске не в последовательном порядке.

B-Tree-индексы по своей сути требуют произвольного доступа к диску, а как иначе добраться до листовой страницы? Поэтому произвольный доступ здесь является правилом, а не исключением. Однако производительность работы с листовыми страницами можно улучшить, когда они являются физически последовательными и плотно упакованными. Если это не так, то мы говорим о фрагментации, и в подобном случае поиск по диапазону и полное сканирование индекса могут оказаться во много раз медленнее. Это особенно справедливо для запросов, покрываемых индексами.

Фрагментации подвержены и данные, хранящиеся в таблицах. Однако механизм фрагментации данных сложнее, чем индексов. Существуют два типа фрагментации данных:

Фрагментация строки

Этот тип фрагментации наблюдается тогда, когда строка хранится в виде нескольких фрагментов в разных местах. Фрагментация строки уменьшает производительность даже если запросу требуется только одна строка из индекса.

Межстрочная фрагментация

Этот тип фрагментации наблюдается тогда, когда логически последовательные страницы или строки хранятся на диске не последовательно. Она влияет на такие операции, как полное сканирование таблицы и сканирование диапазона кластерного индекса, для которых последовательное размещение данных на диске выгоднее.

Таблицы MyISAM могут страдать от фрагментации обоих типов, но InnoDB никогда не фрагментирует короткие строки.

Чтобы дефрагментировать данные, можно запустить команду `OPTIMIZE TABLE`, либо выгрузить данные в файл и заново загрузить их в базу.

Эти методы работают в большинстве подсистем хранения. В некоторых подсистемах, например в MyISAM, можно дефрагментировать индексы, перестраивая их с помощью специального алгоритма, который создает индексы в отсортированном порядке. На данный момент способа дефрагментации индексов в InnoDB не существует, поскольку в версии MySQL 5.0 InnoDB не умеет строить индексы путем сортировки[1]. Даже удаление индексов с повторным их созданием в InnoDB может привести к фрагментации – это зависит от самих данных.

Для подсистем хранения, которые не поддерживают команду `OPTIMIZE TABLE`, вы можете перестроить таблицу с помощью «пустой» команды `ALTER TABLE`. Достаточно указать ту же подсистему хранения, которая используется для таблицы в данный момент:

```
mysql> ALTER TABLE <table> ENGINE=<engine>;
```

[1] На момент написания этой книги разработчики InnoDB работали над этой проблемой.

Нормализация и денормализация

Обычно существует много способов представить имеющиеся данные: от полной нормализации до полной денормализации со всеми промежуточными вариантами. В нормализованной базе данных каждый факт представлен один и только один раз. В денормализованной базе данных, наоборот, информация дублируется или хранится в нескольких местах.

Если вы не знакомы с нормализацией, вам стоит изучить ее. На эту тему существует много хороших книг и ресурсов в Интернете, а здесь мы только дадим краткое введение в те аспекты, которые вы должны знать для понимания этой главы. Приведем для начала классический пример с сотрудниками (employee), подразделениями (department) и руководителями подразделений (department head):

EMPLOYEE	DEPARTMENT	HEAD
Jones	Accounting	Jones
Smith	Engineering	Smith
Brown	Accounting	Jones
Green	Engineering	Smith

Проблема с этой схемой заключается в том, что при модификации данных могут возникать аномалии. Скажем, Brown получил должность начальника отдела Accounting. Чтобы отразить эти изменения, нам нужно обновить много строк, а пока эти обновления выполняются, данные пребывают в несогласованном состоянии. Если строка «Jones» говорит, что начальником отдела является не тот, кто указан в строке «Brown», невозможно выяснить, где правда. Это как в поговорке «Человек с двумя часами никогда не знает точного времени». Кроме того, мы не можем сохранить информацию о подразделении без сотрудников – если мы удалим все записи о сотрудниках в отделе Accounting, то потеряем информацию о самом отделе. Чтобы избежать таких проблем, нам нужно нормализовать таблицу, разделив списки сотрудников и подразделений. Этот процесс приведет к появлению двух следующих таблиц – для сотрудников:

EMPLOYEE_NAME	DEPARTMENT
Jones	Accounting
Smith	Engineering
Brown	Accounting
Green	Engineering

и подразделений:

DEPARTMENT	HEAD
Accounting	Jones
Engineering	Smith

Теперь эти таблицы приведены ко второй нормальной форме, вполне подходящей для многих задач. Однако вторая нормальная форма является только одной их многих возможных нормальных форм.

Здесь мы использовали фамилию как первичный ключ только для иллюстрации, поскольку это «естественный идентификатор» данных. Однако на практике мы так делать не будем. Нет гарантии, что фамилии не повторяются, а использование длинных строк в качестве первичных ключей обычно не оптимально.

Достоинства и недостатки нормализованной схемы

Людям, просящим помощи в борьбе с проблемами производительности, часто советуют нормализовать схемы, особенно, если рабочая нагрузка характеризуется большим числом операций записи. Часто этот совет оказывается верным. Он хорошо работает по следующим причинам:

- Нормализованные таблицы обычно обновляются быстрее, чем ненормализованные.
- Когда данные хорошо нормализованы, они либо редко дублируются, либо не дублируются совсем. Так что изменять приходится меньше данных.
- Нормализованные таблицы обычно меньше по размеру, поэтому лучше помещаются в памяти и их производительность выше.
- Из-за отсутствия избыточных данных реже возникает необходимость в запросах с фразами `DISTINCT` или `GROUP BY` для извлечения списков значений. Рассмотрим предыдущий пример: из денормализованной схемы невозможно получить отдельный список подразделений без `DISTINCT` или `GROUP BY`, а при наличии отдельной таблицы `DEPARTMENT` запрос становится тривиальным.

Недостатки нормализованной схемы обычно проявляются при извлечении данных. Любой нетривиальный запрос к хорошо нормализованной схеме, скорее всего, потребует, по крайней мере одного, а то и нескольких соединений. Это не только дорого, но и делает некоторые стратегии индексирования невозможными. Например, из-за нормализации в разных таблицах могут оказаться столбцы, которые хорошо было бы иметь в одном индексе.

Достоинства и недостатки денормализованной схемы

Достоинством денормализованной схемы является то, что все данные находятся в одной и той же таблице, что позволяет избежать соединений.

Если вам не нужно соединять таблицы, то худшим случаем для большинства запросов – даже тех, которые не используют индексы, – будет полное сканирование таблицы. Это может быть значительно быстрее соединения, когда данные не помещаются в память, поскольку в дан-

ном случае удается избежать операций ввода/вывода с произвольным доступом.

Единая таблица также допускает более эффективные стратегии индексирования. Предположим, у вас есть веб-сайт, на котором посетители могут помещать свои сообщения, причем некоторые пользователи являются привилегированными. Допустим, что требуется показать последние десять сообщений от привилегированных пользователей. Если вы нормализовали схему и индексировали даты публикации сообщений, запрос может выглядеть так:

```
mysql> SELECT message_text, user_name
    -> FROM message
    -> INNER JOIN user ON message.user_id=user.id
    -> WHERE user.account_type='premium'
    -> ORDER BY message.published DESC LIMIT 10;
```

Для эффективного выполнения этого запроса MySQL потребуется просканировать индекс `published` над таблицей `message`. Для каждой найденной строки придется заглядывать в таблицу `user` и проверять, является ли пользователь привилегированным. Если доля привилегированных пользователей невелика, такая обработка будет неэффективной.

Другой возможный план запроса заключается в том, чтобы начать с таблицы `user`, выбрать список привилегированных пользователей, получить все сообщения для них и выполнить сортировку без индекса (filesort). Это, вероятно, будет еще хуже.

Проблемой является соединение, которое не позволяет сортировать и фильтровать одновременно с помощью единственного индекса. Если вы денормализуете данные, объединив таблицы и добавив индекс по (`account_type, published`), то сможете написать запрос без соединения. Это будет очень эффективно:

```
mysql> SELECT message_text,user_name
    -> FROM user_messages
    -> WHERE account_type='premium'
    -> ORDER BY published DESC
    -> LIMIT 10;
```

Сочетание нормализации и денормализации

Как, зная о преимуществах и недостатках нормализованных и денормализованных схем, выбрать лучший вариант?

Истина заключается в том, что полностью нормализованные и полностью денормализованные схемы подобны лабораторным мышам: они редко имеют что-то общее с реальным миром. На практике часто приходится сочетать оба подхода, применяя частично нормализованные схемы, кэширующие таблицы, и другие приемы.

Самым общим способом денормализации данных является дублирование или кэширование отдельных столбцов из одной таблицы в другую. В MySQL 5.0 и более новых версиях вы можете использовать для обновления кэшированных значений триггеры, что облегчает реализацию задачи.

Например, в нашем примере веб-сайта вместо выполнения полной денормализации можно сохранять значение `account_type` и в таблице `user`, и в таблице `message`. Это поможет избежать проблем при вставке и удалении, которые возникли бы в случае полной денормализации, поскольку вы никогда не потеряете информацию о пользователе, даже если сообщений нет. Это не сильно увеличит размеры таблицы `user_message`, зато позволит эффективно выбирать данные.

Однако теперь обновление типа учетной записи пользователя становится дороже, поскольку вам придется изменять ее в обеих таблицах. Чтобы понять, станет ли это проблемой, нужно оценить, насколько часто будут выполняться такие изменения, и сколько времени они будут занимать, а затем сравнить с тем, насколько часто будут выполняться запросы `SELECT`.

Еще одна возможная причина переноса некоторых данных из родительской таблицы в дочернюю – сортировка. Например, в нормализованной схеме сортировка сообщений по имени автора будет чрезвычайно дорогой операцией, но вы можете осуществлять такую сортировку очень эффективно, если кэшируете столбец `author_name` в таблице `message` и проиндексируете ее.

Также может оказаться полезным кэшировать производные значения. Если вам нужно показывать, сколько сообщений написал каждый пользователь (как это делается на многих форумах), то можно либо запустить дорогой подзапрос, который будет подсчитывать сообщения при каждом отображении, либо добавить в таблицу `user` столбец `num_messages`, который будет обновляться каждый раз, когда пользователь отправляет новое сообщение.

Кэширующие и сводные таблицы

В некоторых случаях наилучший способ увеличения производительности состоит в том, чтобы сохранить избыточные данные в той же таблице, что и данные, от которых они произошли. Однако иногда требуется построить совершенно отдельную сводную или кэширующую таблицу, специально настроенную под ваши потребности для извлечения данных. Этот метод работает хорошо, если вы готовы смириться с некоторым устареванием информации, но иногда нет другого выбора (например, когда нужно избежать сложных и дорогостоящих обновлений в режиме реального времени).

Термины «кэширующая таблица» и «сводная таблица» не являются общепринятыми. Мы используем понятие «кэширующая таблица» для

обозначения таблиц, содержащих данные, которые можно легко, хотя и не так быстро, извлечь из схемы (т. е. логически избыточные данные). Под «сводными таблицами» мы понимаем таблицы, в которых хранятся агрегированные данные из запросов с фразой GROUP BY (т. е. данные, не являющиеся логически избыточными). Сводные таблицы иногда называют roll-up таблицы (таблицы-свертки), поскольку данные, которые они хранят, свернуты (по одному или нескольким измерениям).

Продолжая наш пример с веб-сайтом, предположим, что требуется подсчитать количество сообщений, размещенных за последние 24 часа. На загруженном сайте поддерживать точный счетчик в режиме реального времени было бы невозможно. Вместо этого гораздо лучше каждый час генерировать сводную таблицу. Часто хватает единственного запроса, и это получается эффективнее, чем подсчитывать каждое сообщение. Недостаток же заключается в том, что значения такого счетчика не являются стопроцентно точными.

Если необходим точный счетчик сообщений, размещенных за предыдущие 24 часа, то есть и другой вариант. Начнем с почасовой сводной таблицы. Затем вычислим точное количество сообщений, размещенных за данный 24-часовой период, суммируя число сообщений за 23 целых часа этого периода, за частичный час в начале периода и за частичный час в конце периода. Предположим, что сводная таблица называется msg_per_hr и определена следующим образом:

```
CREATE TABLE msg_per_hr (
    hr DATETIME NOT NULL,
    cnt INT UNSIGNED NOT NULL,
    PRIMARY KEY(hr)
);
```

Вы можете найти количество сообщений, размещенных за предыдущие 24 часа, путем сложения результатов следующих трех запросов[1]:

```
mysql> SELECT SUM(cnt) FROM msg_per_hr
    -> WHERE hr BETWEEN
    ->   CONCAT(LEFT(NOW( ), 14), '00:00') - INTERVAL 23 HOUR
    ->   AND CONCAT(LEFT(NOW( ), 14), '00:00') - INTERVAL 1 HOUR;
mysql> SELECT COUNT(*) FROM message
    -> WHERE posted >= NOW( ) - INTERVAL 24 HOUR
    ->   AND posted < CONCAT(LEFT(NOW( ), 14), '00:00') - INTERVAL 23 HOUR;
mysql> SELECT COUNT(*) FROM message
    -> WHERE posted >= CONCAT(LEFT(NOW( ), 14), '00:00');
```

Любой из подходов – неточный подсчет или точное вычисление с запросами по коротким диапазонам – эффективнее, чем подсчет всех строк в таблице message. Это основная причина создания сводных таблиц. Высчитывание этой статистики в режиме реального времени обходит-

[1] Мы используем LEFT(NOW(), 14) для округления текущей даты и времени до ближайшего часа.

ся слишком дорого, поскольку требуется либо просматривать большой объем данных, либо использовать запросы, эффективные только при наличии специальных индексов, которые вы не хотите добавлять, чтобы избежать их влияния на обновления строк. Вычисление наиболее активных пользователей или самых распространенных «тегов» являются типичными примерами таких операций.

В свою очередь кэширующие таблицы полезны для оптимизации поисковых запросов и извлечения данных. Такие запросы часто требуют специальной структуры таблиц и индексов, отличной от структуры, которая необходима для оперативной обработки транзакций.

Например, вам может потребоваться много комбинаций индексов для ускорения различных типов запросов. Эти конфликтующие требования иногда приводят к необходимости создать кэширующую таблицу, содержащую только некоторые столбцы главной таблицы. Полезным приемом является использование для кэширующей таблицы другой подсистемы хранения. Например, если главная таблица использует InnoDB, то при использовании для кэширующей таблицы MyISAM вы добьетесь меньшего размера индекса и возможности выполнять запросы с полнотекстовым поиском. В некоторых случаях имеет смысл полностью вынести таблицу из MySQL и поместить ее в специализированную систему, которая может осуществлять более эффективный поиск, например в поисковые системы Lucene или Sphinx.

При использовании кэширующих и сводных таблиц вам придется решать, поддерживать их в режиме реального времени или периодически перестраивать. Что лучше, зависит от приложения, но периодическое перестроение не только экономит ресурсы, но и может дать более эффективную таблицу без фрагментации и с полностью отсортированными индексами.

При перестроении итоговых и кэширующих таблиц часто требуется, чтобы хранящиеся в них данные на время этой операции оставались доступными. Этого можно добиться, используя «теневую таблицу». Закончив ее построение, вы можете поменять таблицы местами, мгновенно переименовав их. Например, если нужно перестроить таблицу `my_summary`, можно создать таблицу `my_summary_new`, заполнить ее данными и поменять местами с реальной таблицей:

```
mysql> DROP TABLE IF EXISTS my_summary_new, my_summary_old;
mysql> CREATE TABLE my_summary_new LIKE my_summary;
-- заполнить таблицу my_summary_new
mysql> RENAME TABLE my_summary TO my_summary_old, my_summary_new TO my_summary;
```

Если перед присвоением имени `my_summary` вновь построенной таблице вы переименуете исходную таблицу `my_summary` в `my_summary_old`, как мы и поступили, то сможете хранить старую версию до тех пор, пока не придет время следующего перестраивания. Это позволяет выполнить быстрый откат в случае возникновения проблем с новой таблицей.

Таблицы счетчиков

Приложения, хранящие счетчики в таблицах, могут испытывать проблемы совместного доступа при обновлении счетчиков. В веб-приложениях такие таблицы применяются очень часто. Вы можете использовать их для кэширования количества друзей пользователя, количества загрузок файла и т. п. Часто имеет смысл создать для счетчиков отдельную таблицу, что обеспечит ее малый размер и скорость работы. Использование отдельной таблицы поможет избежать вытеснения из кэша запросов и позволит использовать некоторые приемы, которые мы продемонстрируем в этом разделе.

Для простоты представим, что у нас есть таблица счетчиков с одной строкой, в которой просто подсчитывается количество обращений к сайту:

```
mysql> CREATE TABLE hit_counter (
    ->   cnt int unsigned not null
    -> ) ENGINE=InnoDB;
```

При каждом обращении к сайту счетчик обновляется:

```
mysql> UPDATE hit_counter SET cnt = cnt + 1;
```

Проблема заключается в том, что эта единственная строка по существу становится глобальным «мьютексом» для любой транзакции, которая обновляет счетчик. Транзакции оказываются сериализованными. Увеличить уровень конкуренции можно, создав несколько строк и обновляя случайно выбранную строку. Это потребует следующего изменения в таблице:

```
mysql> CREATE TABLE hit_counter (
    ->   slot tinyint unsigned not null primary key,
    ->   cnt int unsigned not null
    -> ) ENGINE=InnoDB;
```

Заранее добавьте в таблицу сто строк. Теперь запрос может просто выбирать случайную строку и обновлять ее:

```
mysql> UPDATE hit_counter SET cnt = cnt + 1 WHERE slot = RAND( ) * 100;
```

Для извлечения статистики просто используйте агрегатный запрос:

```
mysql> SELECT SUM(cnt) FROM hit_counter;
```

Часто требуется время от времени (например, раз в день) начинать подсчет заново. Для этого можно слегка изменить схему:

```
mysql> CREATE TABLE daily_hit_counter (
    ->   day date not null,
    ->   slot tinyint unsigned not null,
    ->   cnt int unsigned not null,
    ->   primary key(day, slot)
    -> ) ENGINE=InnoDB;
```

В этом случае не нужно создавать строки заранее. Вместо этого вы можете использовать фразу ON DUPLICATE KEY UPDATE:

```
mysql> INSERT INTO daily_hit_counter(day, slot, cnt)
    -> VALUES(CURRENT_DATE, RAND( ) * 100, 1)
    -> ON DUPLICATE KEY UPDATE cnt = cnt + 1;
```

Если хочется уменьшить количество строк, чтобы таблица не разрасталась, можно написать периодически выполняемое задание, которое объединит все результаты в слоте 0 и удалит остальные слоты:

```
mysql> UPDATE daily_hit_counter as c
    ->   INNER JOIN (
    ->     SELECT day, SUM(cnt) AS cnt, MIN(slot) AS mslot
    ->     FROM daily_hit_counter
    ->     GROUP BY day
    ->   ) AS x USING(day)
    -> SET c.cnt = IF(c.slot = x.mslot, x.cnt, 0),
    -> c.slot = IF(c.slot = x.mslot, 0, c.slot);
mysql> DELETE FROM daily_hit_counter WHERE slot <> 0 AND cnt = 0;
```

Ускоренное чтение, замедленная запись

Чтобы ускорить работу запросов на чтение, вам часто требуется вводить дополнительные индексы, избыточные поля и даже кэширующие и сводные таблицы. Из-за этого приходится писать лишние запросы и обслуживающие задания, зато повышается производительность: замедление записи компенсируется значительным ускорением операций чтения.

Однако это не единственная цена, которую вы платите за ускорение запросов на чтение. Увеличивается также сложность разработки как для операций чтения, так и для операций записи.

Ускорение работы команды ALTER TABLE

Производительность команды ALTER TABLE в MySQL может стать проблемой при работе с очень большими таблицами. Как правило, MySQL создает в этом случае пустую таблицу с новой структурой, вставляет в нее все данные из старой таблицы и удаляет ее. Если таблица велика и над ней построено много индексов, то это может занимать большой промежуток времени, особенно при недостатке памяти. Многие сталкивались с ситуацией, когда операция ALTER TABLE выполнялась несколько часов или даже дней.

Компания MySQL AB работает над разрешением этой проблемы. В число грядущих усовершенствований входит поддержка «оперативных» процедур, которые не блокируют таблицу на весь период выполнения.

Разработчики InnoDB также трудятся над поддержкой построения индексов путем сортировки. MyISAM уже поддерживает эту технологию, которая делает построение индексов значительно более быстрым и приводит к компактному их размещению (на данный момент InnoDB строит индексы, добавляя по одной строке в порядке первичного ключа, а это означает, что индексные деревья создаются не в оптимальном порядке и оказываются фрагментированными).

Не все операции ALTER TABLE приводят к перестроению таблицы. Например, можно изменить или удалить значение по умолчанию для столбца двумя способами (один быстрый, другой медленный). Предположим, вы хотите изменить продолжительность проката фильма по умолчанию с трех до пяти дней. Вот дорогой способ:

```
mysql> ALTER TABLE sakila.film
    -> MODIFY COLUMN rental_duration TINYINT(3) NOT NULL DEFAULT 5;
```

Профилирование этого оператора с помощью SHOW STATUS показывает, что он осуществляет 1000 считываний и 1000 вставок на уровне подсистемы хранения. Другими словами, он копирует старую таблицу в новую, несмотря на то, что тип, размер и допустимость NULL в столбце не меняются.

Теоретически MySQL может пропускать построение новой таблицы. Значение по умолчанию для столбца фактически хранится в *frm*-файле таблицы, так что его можно изменить, не затрагивая саму таблицу. Пока что MySQL не использует эту оптимизацию, однако любой оператор MODIFY COLUMN приводит к перестраиванию таблицы.

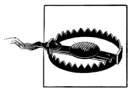
Однако вы можете изменить значение по умолчанию для столбца с помощью команды ALTER COLUMN[1]:

```
mysql> ALTER TABLE sakila.film
    -> ALTER COLUMN rental_duration SET DEFAULT 5;
```

Эта команда модифицирует *frm*-файл, не затрагивая таблицу

Модификация одного лишь frm-файла

Мы видели, что модификация *frm*-файла таблицы происходит быстро и что MySQL иногда без надобности перестраивает таблицу. Если вы готовы пойти на некоторый риск, можете убедить MySQL сделать несколько других типов модификации без перестроения.



Прием, который мы хотим продемонстрировать, не поддерживается официально, недокументирован и может не работать. Используйте его на свой страх и риск. Мы советуем сначала сделать резервную копию данных!

[1] Команда ALTER TABLE позволяет модифицировать столбцы с помощью ALTER COLUMN, MODIFY COLUMN и CHANGE COLUMN. Все три варианта делают разные вещи.

Потенциально вы можете выполнять операции следующих типов без перестроения таблицы:

- Удалить (но не добавить) атрибут столбца AUTO_INCREMENT.
- Добавить, удалить или изменить константы ENUM и SET. Если вы удалите константу, а некоторые строки содержат это значение, запросы будут возвращать для этого значения пустую строку.

Суть приема заключается в том, чтобы создать *frm*-файл для желаемой структуры таблицы и скопировать его на место существующего следующим образом:

1. Создайте пустую таблицу с *точно такой же структурой*, за исключением желаемой модификации (например, с добавленными константами ENUM).
2. Выполните команду FLUSH TABLES WITH READ LOCK. Она закроет все используемые таблицы и предотвратит открытие любых таблиц.
3. Поменяйте местами *frm*-файлы.
4. Выполните команду UNLOCK TABLES, чтобы снять блокировку чтения.

В качестве примера мы добавили константу к столбцу rating в таблице *sakila.film*. Текущее состояние столбца выглядит так:

```
mysql> SHOW COLUMNS FROM sakila.film LIKE 'rating';
+--------+-----------------------------------+------+-----+---------+-------+
| Field  | Type                              | Null | Key | Default | Extra |
+--------+-----------------------------------+------+-----+---------+-------+
| rating | enum('G','PG','PG-13','R','NC-17') | YES  |     | G       |       |
+--------+-----------------------------------+------+-----+---------+-------+
```

Мы добавляем категорию PG-14 для родителей, которые более настороженно относятся к фильмам:

```
mysql> CREATE TABLE sakila.film_new LIKE sakila.film;
mysql> ALTER TABLE sakila.film_new
    -> MODIFY COLUMN rating ENUM('G','PG','PG-13','R','NC-17', 'PG-14')
    -> DEFAULT 'G';
mysql> FLUSH TABLES WITH READ LOCK;
```

Обратите внимание, что мы добавляем новое значение *в конец списка констант*. Если разместить его в середине, после PG-13, то изменятся значения существующих данных: R превратится в PG-14, NC-17 – в R и т. д.

Теперь с помощью команды операционной системы поменяем местами *frm*-файлы:

```
root:/var/lib/mysql/sakila# mv film.frm film_tmp.frm
root:/var/lib/mysql/sakila# mv film_new.frm film.frm
root:/var/lib/mysql/sakila# mv film_tmp.frm film_new.frm
```

Вернувшись к приглашению MySQL, мы можем теперь разблокировать таблицу и увидеть, что изменения вступили в силу:

```
mysql> UNLOCK TABLES;
mysql> SHOW COLUMNS FROM sakila.film LIKE 'rating'\G
*************************** 1. row ***************************
Field: rating
 Type: enum('G','PG','PG-13','R','NC-17','PG-14')
```

Осталось лишь удалить таблицу, которую мы создали для этой операции:

```
mysql> DROP TABLE sakila.film_new;
```

Быстрое построение индексов MyISAM

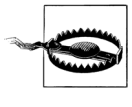
Обычным приемом для эффективной загрузки таблиц MyISAM являются отключение ключей, загрузка данных и повторное включение ключей:

```
mysql> ALTER TABLE test.load_data DISABLE KEYS;
-- загрузка данных
mysql> ALTER TABLE test.load_data ENABLE KEYS;
```

Это работает, поскольку позволяет СУБД MyISAM задержать построение ключей до того момента, когда все данные будут загружены, после чего она может построить индексы путем сортировки. Это происходит намного быстрее и приводит к дефрагментированному, компактному индексному дереву[1].

Увы, данный прием не работает для уникальных индексов, поскольку модификатор `DISABLE KEYS` применяется только к неуникальным. MyISAM строит уникальные индексы в памяти и проверяет их уникальность при загрузке каждой строки. Как только размер индекса превысит объем доступной памяти, загрузка становится чрезвычайно медленной.

Как и в случае с `ALTER TABLE` в предыдущем разделе, вы можете ускорить этот процесс, если готовы проделать дополнительную работу и пойти на некоторый риск. Это может быть полезно для загрузки данных из резервных копий, например когда заранее известно, что все значения корректны и проверять уникальность нет необходимости.



Этот прием также не поддерживается официально и отсутствует в документации. Используйте его на свой страх и риск, не забывая сначала сделать резервную копию данных!

Вот последовательность необходимых действий:

1. Создайте таблицу с нужной структурой, но без всяких индексов.
2. Загрузите данные в таблицу, чтобы построить *MYD*-файл.
3. Создайте другую пустую таблицу с нужной структурой, на сей раз с индексами. Будут сгенерированы необходимые файлы с расширениями *frm* и *MYI*.

[1] MyISAM будет также строить индексы путем сортировки, когда вы используете команду `LOAD DATA INFILE`, а таблица пуста.

4. Сбросьте таблицы, захватив блокировку чтения.
5. Переименуйте *frm*- и *MYI*-файлы второй таблицы, чтобы СУБД MySQL использовала их для первой таблицы.
6. Снимите блокировку чтения.
7. Используйте команду `REPAIR TABLE` для построения индексов над таблицей. Индексы, в том числе и уникальные, будут построены путем сортировки.

Для очень больших таблиц эта процедура может оказаться значительно более быстрой.

Замечания о подсистемах хранения

Мы заканчиваем эту главу некоторыми соображениями о проектировании схемы, специфичными для каждой подсистемы хранения. Мы не пытаемся представить исчерпывающий список, а лищь хотим отметить ключевые факторы, относящиеся к проектированию схемы.

Подсистема хранения MyISAM

Табличные блокировки

Подсистема MyISAM ставит блокировки на уровне таблиц. Убедитесь, что это не создает узких мест.

Отсутствие автоматического восстановления данных

Если происходит сбой сервера MySQL или потеря электропитания, вам нужно проверить и, возможно, восстановить таблицы MyISAM до их использования. Если таблицы велики, этот процесс может занять несколько часов.

Отсутствие поддержки транзакций

Таблицы в этой подсистеме не поддерживают транзакций. Фактически, MyISAM не гарантирует, что и одна-то команда будет доведена до конца. Если, например, ошибка произойдет в процессе выполнения команды `UPDATE`, обновляющей несколько строк, то некоторые строки будут обновлены, а некоторые нет.

В памяти кэшируются только индексы

MyISAM кэширует внутри процесса MySQL только индексы – в буфере ключей. Данные таблицы кэшируются на уровне операционной системы, так что в версии MySQL 5.0 для их извлечения требуется дорогостоящий вызов операционной системы.

Компактное хранение

Строки хранятся упакованными, одна за другой, так что занимают немного места на диске, а полное сканирование таблицы происходит быстро.

Подсистема хранения Memory

Табличные блокировки

Подобно MyISAM подсистема Memory ставит блокировки на уровне таблиц. Однако это не проблема, поскольку запросы к таблицам типа Memory обычно выполняются быстро.

Отсутствие динамических строк

В подсистеме Memory не поддерживаются динамические строки (то есть строки переменной длины), поэтому поля типа `BLOB` и `TEXT` не поддерживаются вовсе. Даже тип `VARCHAR(5000)` превращается в `CHAR(5000)` – очень большой расход памяти, если длина большинства значений мала.

По умолчанию строятся хеш-индексы

В отличие от остальных подсистем хранения по умолчанию строятся хеш-индексы, если явно не оговорено противное.

Отсутствие статистики индексов

Подсистема Memory не поддерживает статистику индексов, поэтому для некоторых сложных запросов вы можете получить плохие планы выполнения.

При перезапуске содержимое теряется

Подсистема Memory не сохраняет данные на диске, в связи с чем при перезапуске сервера данные теряются, хотя определения таблиц сохраняются.

Подсистема хранения InnoDB

Транзакционность

InnoDB поддерживает транзакции и четыре уровня изоляции транзакций.

Внешние ключи

В версии MySQL 5.0 InnoDB является единственной подсистемой хранения, поддерживающей внешние ключи. Другие подсистемы допускают их в команде `CREATE TABLE`, но не проверяют ограничение при работе. Некоторые продукты сторонних разработчиков, например solidDB для MySQL и PBXT, поддерживают их также на уровне подсистемы хранения. Компания MySQL AB планирует добавить в будущем поддержку внешних ключей на уровне сервера.

Блокировки строк

Блокировки устанавливаются на уровне строк. Эскалация блокировок не производится, выборки не блокируют данные – стандартная команда `SELECT` вообще не захватывает никаких блокировок, что обеспечивает очень высокую степень совместной работы.

Многоверсионность

InnoDB использует многоверсионное управление совместным доступом, так что по умолчанию команда `SELECT` может считать устаревшие данные. Фактически архитектура MVCC увеличивает сложность этой подсистемы хранения и может приводить к неожиданному поведению. Если вы пользуетесь InnoDB, внимательно прочитайте документацию.

Кластеризация по первичному ключу

Все таблицы InnoDB кластеризованы по первичному ключу, при проектировании схемы вы можете использовать это обстоятельство в своих интересах.

Все индексы содержат столбцы первичного ключа

Индексы ссылаются на строки по первичному ключу, поэтому удлинение первичного ключа приводит к резкому увеличению размера индексов.

Оптимизированное кэширование

InnoDB кэширует данные и индексы в пуле буферов. Она также автоматически строит хеш-индексы для ускорения выборки строк.

Неупакованные индексы

Префиксное сжатие индексов в этой подсистеме не предусмотрено, поэтому размер индексов может быть значительно больше, чем в MyISAM.

Медленная загрузка данных

В версии MySQL 5.0 подсистема хранения InnoDB не оптимизирует операции загрузки данных. Она строит индексы построчно, а не методом сортировки. Это может значительно уменьшить скорость загрузки данных.

Блокировка AUTO_INCREMENT

В версиях, предшествующих MySQL 5.1, InnoDB захватывает табличную блокировку для порождения очередного значения автоинкрементного столбца.

Значения COUNT(*) не кэшируются

В отличие от таблиц MyISAM и Memory, InnoDB не хранит информацию о количестве строк в таблице, а это означает, что запросы `COUNT(*)` без фразы `WHERE` не могут быть оптимизированы и требуют полного сканирования таблицы или индекса. Подробности приведены в разделе «Оптимизация запросов COUNT()» в главе 4 на стр. 242.

4

Оптимизация запросов

В предыдущей главе мы рассказали об оптимизации схемы. Такая оптимизация является одним из необходимых условий достижения высокой производительности. Но одного этого недостаточно – следует еще правильно конструировать запросы. Если запрос составлен плохо, то даже самая лучшая схема не поможет.

Оптимизация запросов, индексов и схемы идут рука об руку. Постепенно накапливая опыт работы в MySQL, вы начнете понимать, как проектировать схемы для поддержки эффективных запросов. И наоборот, знания о создании оптимальных схем будут отражаться на том, какие запросы вы пишете. Все это, конечно, придет не сразу, поэтому мы призываем вас возвращаться к этой и предыдущей главам по мере освоения нового материала.

Мы начнем данную главу с общих замечаний о конструировании запросов – на что в первую очередь обращать внимание, если запрос выполняется неэффективно. Затем мы углубимся в механизмы оптимизации запросов и детали внутреннего устройства сервера. Мы поможем вам узнать, как именно MySQL выполняет конкретный запрос и как можно изменить план его выполнения. Наконец, мы рассмотрим несколько случаев, когда MySQL оптимизирует запросы не слишком хорошо, и исследуем типичные методы, помогающие СУБД выполнять их более эффективно.

Мы хотим, чтобы вы глубже разобрались в механизмах выполнения запросов, понимали, что считать продуктивным, а что нет, обращали себе во благо сильные стороны MySQL и обходили слабые.

Основная причина замедления: оптимизируйте доступ к данным

Главная причина, из-за которой запрос может выполняться медленно, – слишком большой объем обрабатываемых данных. Конечно, существу-

ют запросы, которые по своей природе должны «перемалывать» очень много всевозможных значений, и тут ничего не поделаешь. Но это довольно редкая ситуация; большинство запросов можно изменить так, чтобы они обращались к меньшему объему данных. Анализ медленно выполняющегося запроса нужно производить в два этапа:

1. Понять, не извлекает ли *приложение* больше данных, чем нужно. Обычно это означает, что слишком велико количество отбираемых строк, но не исключено, что отбираются также лишние столбцы.
2. Понять, не анализирует ли *сервер MySQL* больше строк, чем это необходимо.

Не запрашиваете ли вы лишние данные у базы?

Иногда запрос отбирает больше данных, чем необходимо, а потом отбрасывает некоторые из них. Это требует дополнительной работы от сервера MySQL, приводит к росту накладных расходов на передачу по сети[1], а также увеличивает потребление памяти и процессорного времени на стороне сервера приложений.

Вот несколько типичных ошибок:

Выборка ненужных строк

Широко распространено заблуждение, будто MySQL передает результаты по мере необходимости, а не формирует и возвращает весь результирующий набор целиком. Подобную ошибку мы часто встречали в приложениях, написанных людьми, знакомыми с другими СУБД. Например, применяется такой прием: выполнить команду `SELECT`, которая возвращает много строк, затем выбрать первые строки и закрыть результирующий набор (скажем, отобрать 100 последних по времени статей на новостном сайте, хотя на начальной странице нужно показать только 10). Разработчик полагает, что MySQL вернет первые 10 строк, после чего прекратит выполнение запроса. Но на самом деле MySQL генерирует весь результирующий список. А клиентская библиотека получит полный набор данных и большую часть отбросит. Было бы гораздо лучше включить в запрос фразу `LIMIT`.

Выборка всех столбцов из соединения нескольких таблиц

Если нужно отобрать всех актеров, снимавшихся в фильме *Academy Dinosaur*, не пишите такой запрос:

```
mysql> SELECT * FROM sakila.actor
    -> INNER JOIN sakila.film_actor USING(actor_id)
    -> INNER JOIN sakila.film USING(film_id)
    -> WHERE sakila.film.title = 'Academy Dinosaur';
```

[1] Особенно велики сетевые издержки, когда приложение и сервер работают на разных компьютерах, но даже передача данных между MySQL и приложением на том же компьютере не обходится бесплатно.

Он возвращает все столбцы из всех трех таблиц. Правильнее составить этот запрос следующим образом:

```
mysql> SELECT sakila.actor.* FROM sakila.actor...;
```

Выборка всех столбцов

Наличие `SELECT *` должно вас насторожить. Неужели действительно нужны все столбцы без исключения? Скорее всего, нет. Выборка всех столбцов может воспрепятствовать таким методам оптимизации, как использование покрывающих индексов, и к тому же увеличит потребление сервером ресурсов: ввода/вывода, памяти и ЦП.

По этой причине, а также чтобы уменьшить риск возникновения ошибок при изменении перечня столбцов таблицы, некоторые администраторы базы данных вообще запрещают применять `SELECT *`.

Разумеется, запрашивать больше данных, чем необходимо, не всегда плохо. Во многих изученных случаях нам говорили, что такой расточительный подход упрощает разработку, так как дает возможность использовать один и тот же код в разных местах. Это разумное соображение, если только вы точно знаете, во что оно обходится с точки зрения производительности. Извлечение лишних данных может быть полезно и для организации некоторых видов кэширования на уровне приложения, или если вы видите в этом еще какое-то преимущество. Выборка и кэширование полных объектов иногда предпочтительнее выполнения ряда отдельных запросов, извлекающих части объекта.

Не слишком ли много данных анализирует MySQL?

Если вы уверены, что все запросы *отбирают* лишь необходимые данные, можно поискать запросы, которые *анализируют* слишком много данных для получения результата. В MySQL простейшими метриками стоимости запроса являются:

- Время выполнения
- Количество проанализированных строк
- Количество возвращенных строк

Ни одна из этих метрик не является идеальной мерой стоимости запроса, но все они дают грубое представление о том, сколько данных MySQL должен прочитать для его выполнения, и приблизительно показывают, насколько быстро работает этот запрос. Все три метрики записываются в журнал медленных запросов, так что его просмотр – один из лучших способов выявления запросов, анализирующих слишком много данных.

Время выполнения

В главе 2 мы говорили о том, что в MySQL 5.0 и более ранних версиях у механизма протоколирования медленных запросов были серьезные ограничения, в том числе недостаточно высокое разрешение. К сча-

стью, существуют заплаты, позволяющие протоколировать и измерять время выполнения медленных запросов с микросекундной точностью. Они включены в MySQL 5.1, но при желании можно наложить их и на более ранние версии. Однако не стоит уделять слишком много внимания времени выполнения запроса. Эта метрика полезна по причине своей объективности, но она зависит от текущей загрузки. Другие факторы – блокировки подсистемы хранения (табличные и строковые), высокая степень конкурентности и особенности оборудования – тоже могут оказывать существенное влияние на время выполнения запроса. Эта метрика полезна для отыскания запросов, которые особенно сказываются на скорости реакции приложения или заметно нагружают сервер, но она ничего не говорит о том, является ли наблюдаемое время выполнения разумным для запроса заданной сложности. Время выполнения может быть как симптомом, так и причиной проблемы, и не всегда понятно, с чем именно мы столкнулись.

Количество проанализированных и возвращенных строк

При анализе запросов полезно принимать во внимание, какое количество строк было просмотрено сервером, так как это показывает, насколько эффективно запрос находит нужные вам данные.

Однако, как и время выполнения, эта метрика не идеальна для выявления плохих запросов. Не все обращения к строкам одинаковы. Доступ к коротким строкам занимает меньше времени, а выборка строк из памяти происходит гораздо быстрее, чем чтение их с диска.

В идеале количество проанализированных строк должно совпадать с количеством возвращенных, но на практике так бывает редко. Например, когда в запросе производится соединение нескольких таблиц, для порождения одной строки в результирующем наборе приходится обращаться к исходным строкам. Отношение проанализированных строк к возвращенным обычно мало, скажем, между 1:1 и 10:1, но иногда бывает на несколько порядков больше.

Проанализированные строки и типы доступа

Прикидывая стоимость запроса, учитывайте расходы на поиск одиночной строки в таблице. В MySQL применяется несколько методов поиска и возврата строки. Иногда требуется просмотреть много строк, а иногда удается создать результирующий набор, не анализируя ни одной из них.

Методы доступа отображаются в столбце `type` результата, возвращаемого командой `EXPLAIN`. Это может быть и полное сканирование таблицы, и сканирование индекса, и сканирование диапазона, и поиск по уникальному индексу, и возврат константы. Каждый из вышеперечисленных методов быстрее предыдущего, поскольку требует меньшего количества операций чтения. Помнить все типы доступа необязательно, но очень важно понимать, что означает сканирование таблицы, сканирование индекса, доступ по диапазону и доступ к единственному значению.

Если тип доступа не оптимален, то для решения проблемы лучше всего добавить подходящий индекс. Подробно мы рассматривали вопрос об индексировании в предыдущей главе, а теперь вы видите, почему индексы так важны для оптимизации запросов. Наличие индекса позволяет MySQL применять гораздо более эффективный метод доступа, при котором приходится анализировать меньше данных.

Рассмотрим для примера простой запрос к демонстрационной базе данных Sakila:

```
mysql> SELECT * FROM sakila.film_actor WHERE film_id = 1;
```

Этот запрос возвращает 10 строк, и команда `EXPLAIN` показывает, что MySQL применяет для выполнения запроса тип доступа `ref` по индексу `idx_fk_film_id` `index`:

```
mysql> EXPLAIN SELECT * FROM sakila.film_actor WHERE film_id = 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: idx_fk_film_id
          key: idx_fk_film_id
      key_len: 2
          ref: const
         rows: 10
        Extra:
```

Как показывает `EXPLAIN`, MySQL оценил, что понадобится прочитать всего 10 строк. Иными словами, оптимизатор запросов знает: выбранный тип доступа позволит эффективно выполнить запрос. А если бы подходящего индекса не было? Тогда MySQL был бы вынужден воспользоваться менее эффективным типом доступа. Чтобы убедиться в этом, достаточно удалить индекс и повторить запрос:

```
mysql> ALTER TABLE sakila.film_actor DROP FOREIGN KEY fk_film_actor_film;
mysql> ALTER TABLE sakila.film_actor DROP KEY idx_fk_film_id;
mysql> EXPLAIN SELECT * FROM sakila.film_actor WHERE film_id = 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 5073
        Extra: Using where
```

Как и следовало ожидать, теперь типом доступа является полное сканирование таблицы (`ALL`), и MySQL определила, что для выполнения за-

проса придется проанализировать 5073 строки. Слова «Using where» в столбце Extra говорят о том, что MySQL использует фразу WHERE, чтобы отбросить строки после того, как их считает подсистема хранения.

Вообще говоря, MySQL может применять фразу WHERE тремя способами, которые перечислены ниже в порядке от наилучшего к наихудшему.

- Применить указанные условия к операции поиска по индексу с целью исключить неподходящие строки. Это происходит на уровне подсистемы хранения.
- Использовать покрывающий индекс (слова «Using index» в столбце Extra), чтобы избежать доступа к самим строкам и отфильтровать неподходящие строки после выборки результатов из индекса. Это происходит на уровне сервера, но не требует чтения строк из таблицы.
- Выбрать строки из таблицы, а затем отфильтровать неподходящие (слова «Using where» в столбце Extra). Это происходит на уровне сервера, причем фильтрации предшествует чтение строк из таблицы.

На этом примере видно, насколько важно иметь хорошие индексы. Они позволяют применять при обработке запроса наиболее эффективный тип доступа и анализировать лишь те строки, которые необходимы. Однако добавление индекса не всегда означает, что количество проанализированных и возвращенных строк совпадает. Вот, например, запрос с агрегатной функцией COUNT()[1]:

```
mysql> SELECT actor_id, COUNT(*) FROM sakila.film_actor GROUP BY actor_id;
```

Этот запрос возвращает всего 200 строк, но для построения результирующего набора сервер должен прочитать тысячи. Для подобных запросов индекс не может сократить объем анализируемых данных.

К сожалению, MySQL ничего не говорит о том, какое количество проанализированных строк использовано для построения результирующего набора; сообщается лишь, сколько всего строк было проанализировано. Возможно, многие из них оказались отброшены условием WHERE и не внесли никакого вклада в результирующий набор. Если в предыдущем примере удалить индекс по столбцу sakila.film_actor, то сервер будет обращаться к каждой строке таблицы, но из-за условия WHERE окажутся отброшены все, кроме 10. И лишь оставшиеся 10 строк используются для построения результирующего набора. Чтобы понять, к какому числу строк обращается сервер и сколько из них реально нужны, следует внимательно проанализировать запрос.

Если оказывается, что для получения сравнительно небольшого результирующего набора пришлось проанализировать несоразмерно много строк, то попробуйте применить некоторые ухищрения.

- Воспользуйтесь покрывающими индексами, в которых данные хранятся таким образом, что подсистема хранения вообще не должна

[1] Дополнительную информацию по этому поводу см. в разделе «Оптимизация запросов с функцией **COUNT()» на стр. 242.**

извлекать полные строки (мы обсуждали эту тему в предыдущей главе).

- Измените схему. Например, можно ввести сводные таблицы (см. предыдущую главу).
- Перепишите сложный запрос, так чтобы оптимизатор MySQL мог выполнить его наиболее оптимальным способом (данную технику мы рассмотрим ниже в этой главе).

Способы реструктуризации запросов

Целью оптимизации проблемных запросов должно стать отыскание альтернативных способов получения требуемого результата, хотя далеко не всегда это означает получение точно такого же результирующего набора от MySQL. Иногда удается преобразовать запрос в эквивалентную форму, добившись более высокой производительности. Но следует подумать и о приведении запроса к виду, дающему иной результат, если это позволяет повысить скорость выполнения. Можно даже изменить не только запрос, но и код приложения. В этом разделе мы рассмотрим различные приемы реструктуризации широкого круга запросов и покажем, когда применять каждый из них.

Один сложный или несколько простых запросов?

При конструировании запросов часто приходится отвечать на важный вопрос: не лучше ли будет разбить сложный запрос на несколько более простых? Традиционно при проектировании базы данных стараются сделать как можно больше работы с помощью наименьшего числа запросов. Исторически такой подход был оправдан из-за высокой стоимости сетевых коммуникаций и накладных расходов на разбор и оптимизацию.

Но к MySQL данная рекомендация относится в меньшей степени, поскольку эта СУБД изначально проектировалась так, чтобы установление и разрыв соединения происходили максимально эффективно, а обработка небольших простых запросов выполнялась очень быстро. Современные сети гораздо быстрее, чем раньше, поэтому и сетевые задержки заметно сократились. MySQL способна выполнять свыше 50 000 простых запросов в секунду на типичном серверном оборудовании и свыше 2000 запросов в секунду от одиночного клиента в гигабитной сети, поэтому выполнение нескольких запросов может оказаться вполне приемлемой альтернативой.

Передача информации с использованием соединения все же происходит значительно медленнее по сравнению с тем, какой объем находящихся в памяти данных сам MySQL может перебрать в секунду, – это число измеряется миллионами строк. Так что с учетом всех факторов по-преж-

нему лучше бы ограничиться минимальным количеством запросов, но иногда можно повысить скорость выполнения сложного запроса, разложив его на несколько более простых. Не пугайтесь этого; взвесьте все затраты и выберите ту стратегию, которая уменьшает общий объем работы. Чуть ниже мы приведем примеры применения такой техники.

Несмотря на все вышесказанное, чрезмерно большое количество запросов – одна из наиболее часто встречающихся ошибок при проектировании приложений. Например, в некоторых приложениях выполняется 10 запросов, возвращающих по одной строке, вместо одного запроса, отбирающего 10 строк. Нам даже встречались приложения, в которых каждый столбец выбирался по отдельности, для чего одна и та же строка запрашивалась многократно!

Разбиение запроса на части

Другой способ уменьшить сложность запроса состоит в применении тактики «разделяй и властвуй», когда выполняется по существу один и тот же запрос, но каждый раз из него возвращается меньшее число строк.

Отличный пример – удаление старых данных. В процессе периодической чистки иногда приходится удалять значительные объемы информации. Если делать это одним большим запросом, то возможны всяческие неприятные последствия: блокировки большого числа строк на длительное время, переполнение журналов транзакций, истощение ресурсов, блокировка небольших запросов, которые не допускают прерывания. Разбив команду DELETE на части, каждая из которых удаляет умеренное число строк, мы заметно повысим производительность и уменьшим отставание реплики в случае репликации запроса. Например, вместо следующего монолитного запроса:

```
mysql> DELETE FROM messages WHERE created < DATE_SUB(NOW( ),INTERVAL 3 MONTH);
```

ваши действия можно было бы описать таким псевдокодом:

```
rows_affected = 0
do {
   rows_affected = do_query(
   "DELETE FROM messages WHERE created < DATE_SUB(NOW( ),INTERVAL 3 MONTH)
   LIMIT 10000")
} while rows_affected > 0
```

Обычно удаление 10000 строк за раз – слишком объемная операция, чтобы быть эффективной, и вместе с тем достаточно короткая, чтобы минимизировать негативное воздействие на сервер[1] (транзакционные

[1] Инструмент mk-archiver, входящий в состав продукта Maatkit, упрощает реализацию такого рода задач.

подсистемы хранения могут работать эффективнее при меньшем размере транзакции). Кроме того, имеет смысл вставить небольшую паузу между последовательными командами DELETE, чтобы распределить нагрузку по времени и не удерживать блокировки слишком долго.

Декомпозиция соединения

На многих высокопроизводительных сайтах применяется техника *декомпозиции соединений (join decomposition)*. Смысл ее заключается в том, чтобы выполнить несколько однотабличных запросов вместо одного запроса к нескольким объединенным таблицам, а соединение выполнить уже в приложении. Например, следующий запрос:

```
mysql> SELECT * FROM tag
    -> JOIN tag_post ON tag_post.tag_id=tag.id
    -> JOIN post ON tag_post.post_id=post.id
    -> WHERE tag.tag='mysql';
```

можно было бы заменить такими:

```
mysql> SELECT * FROM tag WHERE tag='mysql';
mysql> SELECT * FROM tag_post WHERE tag_id=1234;
mysql> SELECT * FROM post WHERE post.id in (123,456,567,9098,8904);
```

На первый взгляд, это расточительство, поскольку мы просто увеличили количество запросов, не получив ничего взамен. Тем не менее, такая реструктуризация может дать ощутимый выигрыш в производительности.

- Можно более эффективно реализовать кэширование. Во многих приложениях кэшируются «объекты», которые напрямую соответствуют таблицам. В данном примере, если объект, для которого поле tag равно mysql, уже кэширован, то приложение может пропустить первый запрос. Если выясняется, что в кэше уже есть записи из таблицы post с идентификаторами post_id, равными 123, 567 или 9098, то соответствующие значения можно исключить из списка IN(). Кэш запросов от такой стратегии также выигрывает. Если часто изменяется только одна таблица, то декомпозиция соединения может уменьшить количество перезагрузок записей в кэш (cache invalidations).
- Для подсистемы MyISAM запросы, обращающиеся только к одной таблице, позволяют более эффективно использовать блокировки, поскольку таблицы блокируются по отдельности и на краткий промежуток времени, а не коллективно и надолго.
- Соединение результатов на уровне приложения упрощает масштабирование базы данных путем размещения разных таблиц на различных серверах.
- Сами запросы также могут стать более эффективными. В приведенном примере использование списка IN() вместо соединения позволяет MySQL более эффективно сортировать идентификаторы и более

оптимально извлекать строки, чем это было бы возможно в процессе соединения. Подробнее на этом вопросе мы остановимся ниже.

- Можно избавиться от лишних обращений к строкам. Если соединение производится на уровне приложения, то каждая строка извлекается ровно один раз, тогда как на уровне сервера эта операция по существу сводится к денормализации, в ходе которой обращение к одним и тем же данным может производиться многократно. По той же причине описанная реструктуризация может сократить общий сетевой трафик и потребление памяти.
- В какой-то мере эту технику можно считать ручной реализацией хеш-соединений вместо стандартного применяемого в MySQL алгоритма вложенных циклов. Иногда хеш-соединение оказывается более эффективным (ниже в этой главе мы еще обсудим стратегию выполнения соединений в MySQL).

Резюме: когда соединение на уровне приложения может оказаться эффективнее

Соединение в приложении может оказаться эффективнее в следующих случаях:

- Организован кэш и вы повторно используете ранее запрошенные данные
- Часто используются таблицы типа MyISAM
- Данные распределены по нескольким серверам
- Вместо соединения с большой таблицей используется список `IN()`
- В соединении несколько раз встречается одна и та же таблица

Основные принципы выполнения запросов

Если вы хотите получить максимальную производительность от своего сервера MySQL, то настоятельно рекомендуем потратить время на изучение того, как СУБД оптимизирует и выполняет запросы. Разобравшись в этом, вы обнаружите, что оптимизация запросов основана на следовании четким принципам и реализована весьма логично.

Ниже предполагается, что вы уже прочли главу 1, в которой заложены основы для понимания того, как работает подсистема выполнения запросов в MySQL.

На рис. 4.1 показано, как MySQL в общем случае обрабатывает запрос.

Давайте посмотрим, что происходит, когда вы отправляете запрос на выполнение.

1. Клиент отправляет SQL-команду серверу.
2. Сервер смотрит, есть ли эта команда в кэше запросов. Если да, то возвращается сохраненный результат из кэша; в противном случае выполняется следующий шаг.
3. Сервер осуществляет разбор, предварительную обработку (preprocesing) и оптимизацию SQL-команды, преобразуя ее в план выполнения запроса.
4. Подсистема выполнения запросов выполняет этот план, обращаясь к подсистеме хранения.
5. Сервер отправляет результат клиенту.

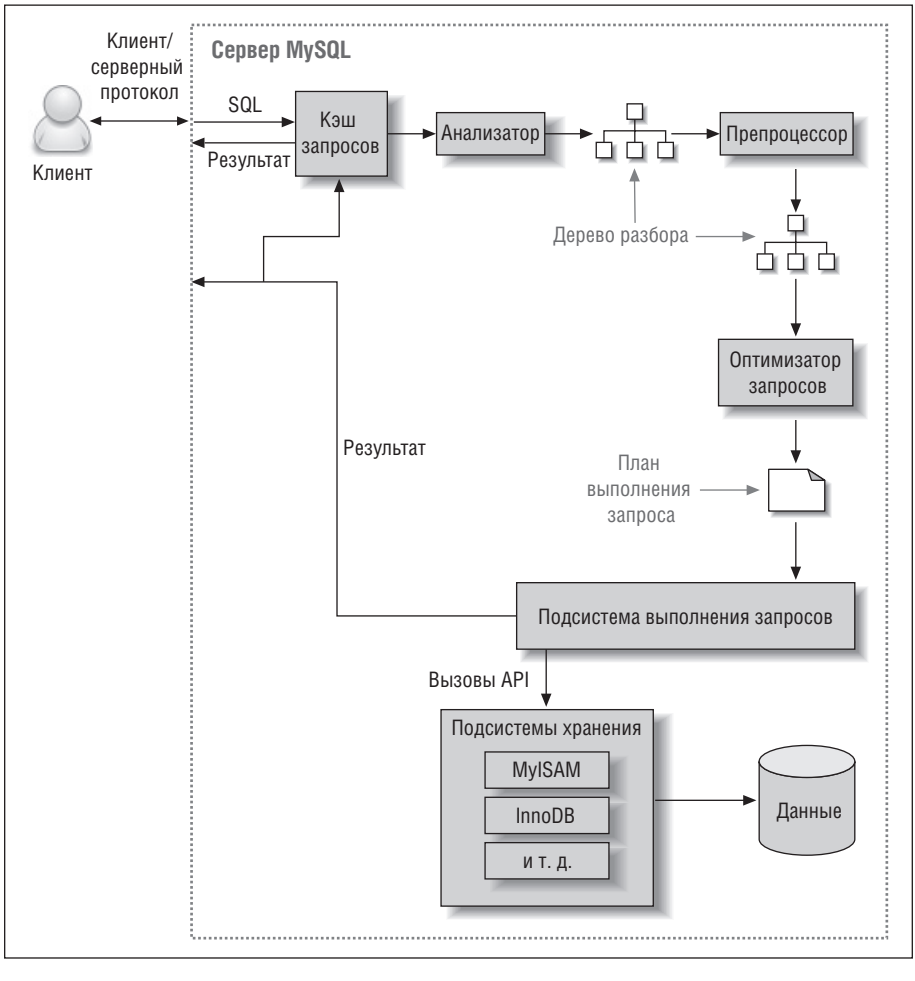


Рис. 4.1. *Порядок выполнения запроса*

На каждом из этих шагов есть свои тонкости, которые мы рассмотрим в последующих разделах. Мы также объясним, в каком состоянии запрос находится на каждом шаге. Особенно сложен процесс оптимизации, но именно его важно понимать.

Клиент–серверный протокол MySQL

Хотя детали клиент-серверного протокола MySQL вам понимать необязательно, но иметь общее представление о том, как он работает, необходимо. Это полудуплексный протокол, то есть в любой момент времени сервер либо отправляет, либо принимает сообщения, но не то и другое вместе. Кроме того, это означает, что невозможно оборвать сообщение «на полуслове».

Данный протокол обеспечивает простое и очень быстрое взаимодействие с MySQL, но имеет кое-какие ограничения. Во-первых, в нем отсутствует механизм управления потоком данных: после того как одна сторона отправила сообщение, другая должна получить его целиком и только потом сможет ответить. Это как перекидывание мячика: в каждый момент времени мячик находится только на одной стороне, невозможно перебросить его сопернику (отправить сообщение), предварительно не получив.

Клиент отправляет запрос в виде одного пакета данных. Поэтому так важна конфигурационная переменная `max_allowed_packet` в случаях, когда встречаются длинные запросы[1]. После отправки запроса мяч уже на другой стороне; клиенту остается только дожидаться результатов.

Напротив, ответ сервера обычно состоит из нескольких пакетов данных. Клиент обязан получить *весь* результирующий набор, отправленный сервером. Нельзя выбрать только первые несколько строк и попросить сервер не посылать остальное. Если клиенту все-таки нужны именно первые строки, то у него есть два варианта действий: дождаться прихода всех отправленных сервером пакетов и отбросить ненужные, или бесцеремонно разорвать соединение. Оба метода не слишком привлекательны, поэтому фраза `LIMIT` так важна.

Можно было бы подумать, что клиент, извлекающий строки с сервера, *вытягивает* (*pull*) их. Но на самом деле это не так: сервер MySQL *выталкивает* (*push*) строки по мере их порождения. Клиент лишь получает передаваемые данные, но не может заставить сервер прекратить передачу. Образно говоря, клиент «пьет из пожарного шланга» (drinking from the fire hose – кстати, это технический термин).

Большинство клиентских библиотек для MySQL позволяют либо получить весь результирующий набор и разместить его в памяти, либо вы-

[1] Если запрос слишком длинный, сервер отказывается принимать его целиком и возвращает ошибку.

бирать по одной строке. Как правило, по умолчанию набор целиком буферизуется в памяти. Это важно, поскольку до тех пор, пока все строки не будут получены, сервер MySQL не освобождает блокировки и другие ресурсы, потребовавшиеся для выполнения запроса. Запрос будет находиться в состоянии Sending data (отправка данных; состояния запросов рассматриваются в следующем разделе). Если клиентская библиотека извлекает все результаты сразу, то на долю сервера остается меньше работы: сервер может закончить выполнение запроса и освободить ресурсы настолько быстро, насколько это вообще возможно.

Как правило, клиентские библиотеки создают впечатление, будто вы получаете данные с сервера, тогда как на самом деле строки выбираются из буфера в памяти самой библиотеки. Обычно в этом нет ничего плохого, если только речь не идет о гигантских наборах данных, на получение которых уходит много времени, а на хранение – много ресурсов. Потребляемый объем памяти можно сократить и приступить к обработке результата скорее, если задать режим работы без буферизации. Минус такого подхода заключается в том, что сервер удерживает блокировки и другие ресурсы на все время, пока приложение взаимодействует с библиотекой[1].

Рассмотрим пример с использованием PHP. Сначала покажем, как обычно отправляется запрос к MySQL из PHP:

```
<?php
$link = mysql_connect('localhost', 'user', 'p4ssword');
$result = mysql_query('SELECT * FROM HUGE_TABLE', $link);
while ( $row = mysql_fetch_array($result) ) {
   // Что-то сделать с результатом
}
?>
```

Код выглядит так, будто строки выбираются по мере необходимости в цикле `while`. Но в действительности весь результирующий набор копируется в буфер при обращении к функции `mysql_query( )`. В цикле `while` мы просто обходим этот буфер. Напротив, в следующем коде результаты не буферизуются, потому что вместо функции `mysql_query( )` вызывается `mysql_unbuffered_query( )`:

```
<?php
$link = mysql_connect('localhost', 'user', 'p4ssword');
$result = mysql_unbuffered_query('SELECT * FROM HUGE_TABLE', $link);
while ( $row = mysql_fetch_array($result) ) {
   // Что-то сделать с результатом
}
?>
```

[1] Это ограничение можно обойти с помощью указания оптимизатору SQL_BUFFER_RESULT, которое описывается ниже.

В разных языках программирования приняты разные способы подавления буферизации. Например, в драйвере `DBD::mysql` на Perl необходимо задавать атрибут `mysql_use_result` (по умолчанию предполагается наличие атрибута `mysql_store_result`):

```
#!/usr/bin/perl
use DBI;
my $dbh = DBI->connect('DBI:mysql:;host=localhost', 'user', 'p4ssword');
my $sth = $dbh->prepare('SELECT * FROM HUGE_TABLE', { mysql_use_result => 1
});
$sth->execute( );
while ( my $row = $sth->fetchrow_array( ) ) {
   # Что-то сделать с результатом
}
```

Обратите внимание, что в вызове метода `prepare( )` атрибут `mysql_use_result` говорит о том, что результат следует «использовать», а не «буферизовать». Можно было бы задать этот атрибут и на этапе установления соединения, тогда режим буферизации отключался бы для каждой команды:

```
my $dbh = DBI->connect('DBI:mysql:;mysql_use_result=1', 'user', 'p4ssword');
```

Состояния запроса

У каждого соединения, или *потока* MySQL имеется состояние, показывающее, что происходит в текущий момент времени. Есть несколько способов узнать состояние, и самый простой из них – воспользоваться командой `SHOW FULL PROCESSLIST` (состояния выводятся в столбце `Command`). На протяжении жизненного цикла запроса состояние меняется много раз, а всего насчитывается несколько десятков состояний. Авторитетным источником информации по состояниям является руководство по MySQL, но некоторые из них мы опишем здесь.

`Sleep`

Поток ожидает поступления нового запроса от клиента.

`Query`

Поток либо занят выполнением запроса, либо отправляет клиенту результаты.

`Locked`

Поток ожидает предоставления табличной блокировки на уровне сервера. Блокировки, реализованные подсистемой хранения, например блокировки строк в InnoDB, не вызывают перехода в состояние `Locked`.

`Analyzing` *и* `Statistics`

Поток проверяет статистику, собранную подсистемой хранения, и оптимизирует запрос.

`Copying to tmp table [on disk]`

Поток обрабатывает запрос и копирует результаты во временную таблицу; скорее всего, для группировки, затребованной во фразе `GROUP BY`, для сортировки (filesort) или для удовлетворения запроса, содержащего фразу `UNION`. Если название состояния оканчивается словами «on disk», значит, MySQL преобразует таблицу в памяти в таблицу на диске.

`Sorting result`

Поток занят сортировкой результирующего набора.

`Sending data`

Поток в этом состоянии выполняет одно из следующих действий: пересылает данные между различными стадиями обработки запроса, генерирует результирующий набор или возвращает результаты клиенту.

Знать хотя бы основные состояния полезно, поскольку это позволяет понять, «на чьей стороне мячик». На очень сильно загруженных серверах можно заметить, что состояние, которое обычно появляется крайне редко или на короткое время, вдруг начинает «висеть» довольно долго. Обычно это свидетельствует о возникновении какой-то аномалии.

Кэш запросов

Еще перед тем как приступать к разбору запроса, MySQL проверяет, нет ли его в кэше запросов (если режим кэширования включен). При этом производится поиск в хеш-таблице с учетом регистра ключа. Если поступивший запрос отличается от хранящегося в кэше хотя бы в одном байте, запросы считаются разными, и сервер переходит к следующей стадии обработки запроса.

Если MySQL находит запрос в кэше, то перед тем, как возвратить сохраненный результат, он должен проверить привилегии. Это можно сделать, даже не разбирая запрос, так как вместе с кэшированным запросом MySQL хранит информацию о таблицах. Если с привилегиями все в порядке, MySQL выбирает из кэша ассоциированный с запросом результат и отправляет его клиенту, минуя все остальные стадии. В этом случае для запроса не производятся разбор, оптимизация и выполнение.

Подробнее о кэше запросов рассказывается в главе 5.

Процесс оптимизации запроса

Следующий шаг в жизненном цикле запроса – преобразование SQL-команды в план выполнения, необходимый подсистеме выполнения запросов. Он состоит из нескольких этапов: разбор, предварительная обработка (preprocessing) и оптимизация. Ошибки (например, синтаксические) возможны в любой точке этого процесса. Поскольку в нашу задачу не входит строгое документирование внутреннего устройства

MySQL, мы можем позволить себе некоторые вольности, например описывать шаги по отдельности, хотя часто они ради эффективности целиком или частично совмещены. У нас лишь одна цель – помочь вам разобраться в том, как MySQL выполняет запросы, чтобы вы могли составлять их более оптимально.

Анализатор и препроцессор

Прежде всего, *анализатор* MySQL разбивает запрос на лексемы и строит «дерево разбора». Для интерпретации и проверки запроса анализатор использует грамматику языка SQL. В частности, проверяется, что все лексемы допустимы, следуют в нужном порядке и нет других ошибок, например непарных кавычек.

Затем получившееся дерево разбора передается *препроцессору*, который контролирует дополнительную семантику, не входящую в компетенцию анализатора. К примеру, проверяется, что указанные таблицы и столбцы существуют, а ссылки на столбцы не допускают неоднозначного толкования.

Далее препроцессор проверяет привилегии. Обычно этот шаг выполняется очень быстро, если только на сервере определено не слишком много привилегий. Дополнительную информацию о привилегиях и безопасности см. в главе 12.

Оптимизатор запросов

Теперь, когда дерево разбора тщательно проверено, наступает очередь *оптимизатора*, который превращает его в план выполнения запроса. Часто существует множество способов выполнить запрос, и все они дают один и тот же результат. Задача оптимизатора – выбрать лучший из них.

В MySQL используется стоимостный оптимизатор, пытающийся предсказать стоимость различных планов выполнения и выбрать из них наиболее дешевый. В качестве единицы стоимости принимаются затраты на считывание случайной страницы данных размером 4 Кбайт. Чтобы узнать, как оптимизатор оценил запрос, выполните этот запрос, а затем посмотрите на сеансовую переменную `Last_query_cost`:

```
mysql> SELECT SQL_NO_CACHE COUNT(*) FROM sakila.film_actor;
+----------+
| count(*) |
+----------+
|     5462 |
+----------+
mysql> SHOW STATUS LIKE 'last_query_cost';
+-----------------+-------------+
| Variable_name   | Value       |
+-----------------+-------------+
| Last_query_cost | 1040.599000 |
+-----------------+-------------+
```

Этот результат означает, что согласно оценке оптимизатора для выполнения запроса потребуется выполнить примерно 1040 случайных чтений страниц данных. Оценка вычисляется на основе различной статистической информации: количество страниц в таблице или в индексе, *кардинальность* (количество различных значений) индекса, длина строк и ключей, распределение ключей. Оптимизатор не учитывает влияния кэширования – предполагается, что любое чтение сводится к операции дискового ввода/вывода.

Не всегда оптимизатор выбирает наилучший план, и тому есть много причин.

- Некорректная статистика. Сервер получает статистическую информацию от подсистемы хранения, и тут есть масса вариантов: от абсолютно верных до не имеющих ничего общего с действительностью. Например, подсистема хранения InnoDB не ведет точную статистику количества строк в таблице, так уж устроена архитектура многоверсионного управления конкурентным доступом (MVCC).
- Принятая метрика стоимости не всегда эквивалентна истинной стоимости выполнения запроса, поэтому даже когда статистика точна, запрос может оказаться дороже или дешевле оценки MySQL. В некоторых случаях план, предполагающий чтение большего количества страниц, будет дешевле, потому, например, что чтение с диска производится последовательно или страницы уже находятся в памяти.
- Представление MySQL о том, что такое «оптимально», может расходиться с вашим представлением. Вы, вероятно, хотите получить результат как можно быстрее, но для MySQL понятия «быстро» не существует, он оперирует лишь «стоимостью», а вычисление стоимости, как мы только что видели, – неточная наука.
- MySQL не берет в расчет другие одновременно выполняющиеся запросы, а они могут повлиять на время обработки оптимизируемого.
- MySQL не всегда выполняет оптимизацию по стоимости. Иногда он просто следует правилам, например: «если запрос содержит фразу `MATCH()`, то используется полнотекстовый индекс, если таковой существует». Подобное решение будет принято, даже если быстрее было бы воспользоваться другим индексом и неполнотекстовым запросом с фразой `WHERE`.
- Оптимизатор не учитывает стоимость операций, которые ему неподконтрольны, например выполнение хранимых или определенных пользователем функций.
- Позже мы увидим, что не всегда оптимизатор способен рассмотреть все возможные планы выполнения, поэтому оптимальный план он может просто не увидеть.

Оптимизатор запросов MySQL – это очень сложный код, в котором для преобразования запроса в план выполнения применяется много раз-

ных операций. Но существует всего два основных вида оптимизации: *статическая* и *динамическая*. Для выполнения *статической оптимизации* достаточно одного лишь исследования дерева разбора. Например, оптимизатор может преобразовать фразу `WHERE` в эквивалентную форму, применяя алгебраические правила. Статическая оптимизация не зависит от конкретных значений, таких как константы в условии `WHERE`. Будучи один раз произведена, статическая оптимизация всегда остается в силе, даже если запрос будет повторно выполнен с другими значениями. Можно считать, что это «оптимизация на этапе компиляции».

С другой стороны, *динамические оптимизации* зависят от контекста и могут определяться многими факторами, скажем, конкретным значением в условии `WHERE` или количеством строк в индексе. Их приходится заново вычислять при каждом выполнении запроса. Можно считать, что это «оптимизация на этапе выполнения».

Данное различие важно при выполнении подготовленных (prepared) команд и хранимых процедур. MySQL может произвести статическую оптимизацию однократно, но динамические оптимизации должен заново вычислять при каждом выполнении запроса. Иногда MySQL даже повторно производит оптимизацию во время выполнения запроса[1].

Ниже перечислены несколько типов оптимизаций, поддерживаемых в MySQL.

Изменение порядка соединения

Таблицы не обязательно соединять именно в том порядке, который указан в запросе. Определение наилучшего порядка соединения – важная оптимизация; подробнее мы рассмотрим ее ниже в разделе «Оптимизатор соединений» на стр. 226.

Преобразование `OUTER JOIN` *в* `INNER JOIN`

Оператор `OUTER JOIN` необязательно выполнять как внешнее соединение. При определенных условиях, зависящих, например, от фразы `WHERE` и схемы таблицы, запрос с `OUTER JOIN` эквивалентен запросу с `INNER JOIN`. MySQL умеет распознавать и переписывать такие запросы, после чего они могут быть подвергнуты оптимизации типа «изменение порядка соединения».

Применение алгебраических правил эквивалентности

MySQL применяет алгебраические преобразования для упрощения выражений и приведения их к каноническому виду. Она умеет также вычислять константные выражения, исключая заведомо невы-

[1] Например, план выполнения с проверкой диапазона повторно оценивает индексы для каждой строки в соединении `JOIN`. Опознать такой план можно по наличию слов «range checked for each record» в столбце `Extra`, формируемом командой `EXPLAIN`. При выборе подобного плана также увеличивается серверная переменная `Select_full_range_join server`.

полнимые и всегда выполняющиеся условия. Например, терм (5=5 AND a>5) приводится к более простому: a>5. Аналогично условие (a<b AND b=c) AND a=5 принимает вид b>5 AND b=c AND a=5. Эти правила очень полезны при написании условных запросов, о чем пойдет речь ниже в настоящей главе.

Оптимизации COUNT(), MIN() *и* MAX()

Наличие индексов и сведений о возможности хранения NULL-значений в столбцах часто позволяет вообще не вычислять эти выражения. Например, чтобы найти минимальное значение в столбце, который является самой левой частью ключа индекса типа B-Tree, MySQL может просто запросить первую строку из этого индекса. Это можно сделать даже на стадии оптимизации и далее рассматривать полученное значение как константу. Аналогично для поиска максимального значения в индексе типа B-Tree сервер считывает последнюю строку. Если применена такая оптимизация, то в плане, выведенном командой EXPLAIN, будет присутствовать фраза «Select tables optimized away» (некоторые таблицы исключены при оптимизации). Это означает, что оптимизатор полностью исключил таблицу из плана выполнения, подставив вместо нее константу.

Похожим образом некоторые подсистемы хранения могут оптимизировать запросы, содержащие COUNT(*)без фразы WHERE (к примеру, MyISAM, где количество строк в таблице всегда известно точно). Дополнительную информацию см. в разделе «Оптимизация запросов с COUNT()» этой главы на стр. 242.

Вычисление и свертка константных выражений

Если MySQL обнаруживает, что выражение можно свернуть в константу, то делает это на стадии оптимизации. Например, определенную пользователем переменную можно преобразовать в константу, если она не изменяется в запросе. Другим примером могут служить арифметические выражения.

Как ни странно, даже такие вещи, которые вы, скорее всего, назвали бы запросом, можно свернуть в константу во время оптимизации. Например, вычисление функции MIN() по индексу. Этот пример можно даже обобщить на поиск констант по первичному ключу или по уникальному индексу. Если во фразе WHERE встречается константное условие для такого индекса, то оптимизатор знает, что MySQL могла бы найти значение в самом начале выполнения запроса. Впоследствии найденное значение можно трактовать как константу. Приведем пример:

```
mysql> EXPLAIN SELECT film.film_id, film_actor.actor_id
    -> FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id)
    -> WHERE film.film_id = 1;
+----+-------------+------------+-------+----------------+-------+------+
```

```
| id | select_type | table      | type  | key            | ref   | rows |
+----+-------------+------------+-------+----------------+-------+------+
| 1 | SIMPLE       | film       | const | PRIMARY        | const |    1 |
| 1 | SIMPLE       | film_actor | ref   | idx_fk_film_id | const |   10 |
+----+-------------+------------+-------+----------------+-------+------+
```

MySQL выполняет этот запрос в два этапа, о чем свидетельствуют две строки в выведенной таблице. На первом этапе находится нужная строка в таблице `film`. Оптимизатор MySQL знает, что такая строка единственная, поскольку столбец `film_id` – это первичный ключ. Так как оптимизатору известна точная величина (значение во фразе `WHERE`), которая будет возвращена в результате поиска, то в столбце `ref` для этой таблицы стоит `const`.

На втором шаге MySQL считает столбец `film_id` из строки, найденной на первом шаге, известной величиной. Он может так поступить, потому что в момент перехода ко второму шагу оптимизатор уже знает все значения, определенные ранее. Отметим, что тип `ref` для таблицы `film_actor` равен `const` точно так же, как и для таблицы `film`.

Константные условия могут применяться также вследствие распространения «константности» значения из одного места в другое при наличии фразы `WHERE`, `USING` или `ON` с ограничением типа «равно». В приведенном выше примере фраза `USING` гарантирует, что значение `film_id` будет одинаково на протяжении всего запроса – оно должно быть равно константе, заданной во фразе `WHERE`.

Покрывающие индексы

Если индекс содержит все необходимые запросу столбцы, то MySQL может воспользоваться им, вообще не читая данные таблицы. Мы подробно рассматривали покрывающие индексы в главе 3.

Оптимизация подзапросов

MySQL умеет преобразовывать некоторые виды подзапросов в более эффективные эквивалентные формы, сводя их к поиску по индексу.

Раннее завершение

MySQL может прекратить обработку запроса (или какой-то шаг обработки), как только поймет, что этот запрос или шаг полностью выполнен. Очевидный пример – фраза `LIMIT`, но есть и еще несколько случаев раннего завершения. Например, встретив заведомо невыполнимое условие, MySQL может прекратить обработку всего запроса. Взгляните на следующий пример:

```
mysql> EXPLAIN SELECT film.film_id FROM sakila.film WHERE film_id = -1;
+----+...+------------------------------------------------+
| id |...| Extra                                          |
+----+...+------------------------------------------------+
| 1 |...| Impossible WHERE noticed after reading const tables |
+----+...+------------------------------------------------+
```

Этот запрос остановлен на шаге оптимизации, но иногда MySQL прерывает запрос вскоре после начала его выполнения. Сервер может применить такую оптимизацию, когда подсистема выполнения приходит к выводу, что нужно извлекать только различающиеся значения либо остановиться, если значения не существует. Например, следующий запрос находит все фильмы, в которых нет ни одного актера:[1]

```
mysql> SELECT film.film_id
    -> FROM sakila.film
    ->    LEFT OUTER JOIN sakila.film_actor USING(film_id)
    -> WHERE film_actor.film_id IS NULL;
```

Запрос исключает все фильмы, в которых есть хотя бы один актер. В фильме может быть задействовано много актеров, но, обнаружив первого, сервер прекращает обработку текущего фильма и переходит к следующему, поскольку знает, что условию `WHERE` такой фильм заведомо не удовлетворяет. Похожую оптимизацию «различающиеся/не существует» можно применить к некоторым запросам, включающим операторы `DISTINCT`, `NOT EXISTS( )`и `LEFT JOIN`.

Распространение равенства

MySQL распознает ситуации, когда в некотором запросе два столбца должны быть равны, – например, в условии `JOIN`, и распространяет условие `WHERE` на эквивалентные столбцы. В частности, следующий запрос

```
mysql> SELECT film.film_id
    -> FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id)
    -> WHERE film.film_id > 500;
```

демонстрирует MySQL, что условие `WHERE` применяется не только к таблице `film`, но и к таблице `film_actor`, поскольку в силу наличия фразы `USING` оба столбца должны совпадать.

Если вы привыкли к другой СУБД, в которой такая оптимизация не реализована, то вам, вероятно, рекомендовали «помочь оптимизатору», самостоятельно задав во фразе `WHERE` условия для обеих таблиц, например:

```
... WHERE film.film_id > 500 AND film_actor.film_id > 500
```

В MySQL это необязательно и лишь усложняет сопровождение запросов.

---

[1] Согласны, фильм без актеров выглядит странно, но в демонстрационной базе данных Sakila такой фильм есть. Он называется «SLACKER LIAISONS» и аннотирован как «стремительно развивающаяся история о мошеннике и студенте, которые должны встретиться с крокодилом в Древнем Китае».

Сравнение по списку IN( )

Во многих СУБД оператор `IN()` – не более чем синоним нескольких условий `OR`, поскольку логически они эквивалентны. Но не в MySQL, здесь перечисленные в списке `IN()`значения сортируются, и для работы с ним применяется быстрый двоичный поиск. Вычислительная сложность при этом составляет `O(log n)`, где `n` – размер списка, тогда как сложность эквивалентной последовательности условий `OR` равна `O(n)` (т. е. гораздо медленнее для больших списков).

Приведенный выше перечень, конечно, неполон, так как MySQL умеет применять гораздо больше оптимизаций, чем поместилось бы во всей этой главе, но представление о сложности и изощренности оптимизатора вы все же получили. Главная мысль, которую следует вынести из этого обсуждения, – *не пытайтесь перехитрить оптимизатор.* В конечном итоге вы просто потерпите неудачу или сделаете свои запросы чрезмерно запутанными и трудными для сопровождения, не получив ни малейшей выгоды. Позвольте оптимизатору заниматься своим делом.

Разумеется, каким бы «умным» ни был оптимизатор, иногда он не находит наилучший результат. Бывает так, что вы знаете о своих данных что-то такое, о чем оптимизатору неизвестно, например, некое условие, которое гарантированно истинно вследствие логики приложения. Кроме того, оптимизатор не наделен кое-какой функциональностью, например хеш-индексами, а временами алгоритм оценки стоимости выбирает план, оказывающийся более дорогим, чем возможная альтернатива.

Если вы уверены, что оптимизатор дает плохой результат и знаете, почему, то можете ему помочь. Вариантов тут несколько: включить в запрос подсказку (hint), переписать запрос, перепроектировать схему или добавить индексы.

Статистика по таблицам и индексам

Вспомните архитектурные уровни сервера MySQL, показанные на рис. 1.1. На уровне сервера, где расположен оптимизатор запросов, не содержится статистика по данным и индексам. Эта обязанность возлагается на подсистемы хранения, поскольку каждая из них может собирать различную статистическую информацию (и хранить ее по-разному). Некоторые подсистемы, к примеру Archive, вообще не собирают статистику!

Поскольку сервер статистику не содержит, оптимизатор запросов MySQL должен обращаться к подсистемам хранения за статистическими данными по участвующим в запросе таблицам. Подсистема может предоставить такую информацию, как количество страниц в таблице или индексе, кардинальность таблиц и индексов, длина строк и ключей, гистограмма распределения ключей. На основе этих сведений оптимизатор выбирает наилучший план выполнения. В последующих разделах мы увидим, какое влияние статистика оказывает на решения оптимизатора.

Стратегия выполнения соединений в MySQL

В MySQL термин «соединение» (join) используется шире, чем вы, возможно, привыкли. Под соединениями понимаются все запросы, а не только те, что сопоставляют строки из двух таблиц. Еще раз подчеркнем: имеются в виду все без исключения запросы, в том числе подзапросы и даже выборка `SELECT` из одной таблицы[1]. Следовательно, очень важно понимать, как в MySQL выполняется операция соединения.

Возьмем, к примеру, запрос `UNION`. MySQL реализует операцию объединения (`UNION`) в виде последовательности отдельных запросов, результаты которых сохраняются во временной таблице, а затем снова читаются из нее. Каждый запрос представляет собой соединение в терминологии MySQL, и точно также соединением является операция чтения результирующей временной таблицы.

В текущей версии стратегия соединения в MySQL проста: все соединения реализуются алгоритмом вложенных циклов. Это означает, что MySQL в цикле перебирает строки из одной таблицы, а затем во вложенном цикле ищет соответствующие строки в следующей. И так со всеми соединяемыми таблицами. После этого СУБД строит и возвращает строку, составленную из перечисленных в списке `SELECT` столбцов. Далее MySQL пытается найти следующую строку в последней таблице. Если такой не оказывается, то производится возврат на одну таблицу назад и попытка найти дополнительные строки в ней. MySQL продолжает выполнять возвраты, пока в какой-то таблице не обнаружится подходящая строка, после чего ищет соответствующую ей строку в следующей таблице и т. д.[2]

Этот процесс, состоящий из поиска строк, проверки следующей таблицы и возврата, можно представить в плане выполнения в виде последовательности вложенных циклов, отсюда и названия «соединение методом вложенных циклов».

В качестве примера рассмотрим простой запрос:

---

[1] В оригинальной документации по MySQL операция соединения описывается следующим образом:

```
SELECT ...
select_expr [, select_expr ...]
[FROM table_references
[WHERE where_condition]...]
```

Во фразе `FROM table_references` мы указываем таблицу или таблицы, из которых будут извлекаться строки. Если мы указываем более чем одну таблицу, то выполняется операция соединения. Для более полного понимания операций соединения мы рекомендуем прочитать соответствующие разделы реляционной алгебры. – *Прим. науч. ред.*

[2] Ниже мы увидим, что выполнение запроса на самом деле происходит не так просто; существует немало оптимизаций, усложняющих алгоритм.

```
mysql> SELECT tbl1.col1, tbl2.col2
    -> FROM tbl1 INNER JOIN tbl2 USING(col3)
    -> WHERE tbl1.col1 IN(5,6);
```

Предположим, что MySQL решает соединять таблицы в порядке, указанном в запросе, тогда следующий псевдокод описывает возможный алгоритм выполнения запроса:

```
outer_iter = iterator over tbl1 where col1 IN(5,6)
outer_row = outer_iter.next
while outer_row
   inner_iter = iterator over tbl2 where col3 = outer_row.col3
   inner_row = inner_iter.next
   while inner_row
      output [ outer_row.col1, inner_row.col2 ]
      inner_row = inner_iter.next
   end
   outer_row = outer_iter.next
end
```

Этот план выполнения с тем же успехом можно применить к запросу, в котором участвует всего одна таблица. Именно поэтому запросы к одной таблице считаются соединениями; такое однотабличное соединение представляет собой базовую операцию, на основе которой конструируются более сложные соединения. Данный алгоритм обобщается и на внешние соединения. Слегка изменим предыдущий запрос:

```
mysql> SELECT tbl1.col1, tbl2.col2
    -> FROM tbl1 LEFT OUTER JOIN tbl2 USING(col3)
    -> WHERE tbl1.col1 IN(5,6);
```

Вот как выглядит соответствующий ему псевдокод (изменения выделены полужирным шрифтом):

```
outer_iter = iterator over tbl1 where col1 IN(5,6)
outer_row = outer_iter.next
while outer_row
   inner_iter = iterator over tbl2 where col3 = outer_row.col3
   inner_row = inner_iter.next
   if inner_row
      while inner_row
         output [ outer_row.col1, inner_row.col2 ]
         inner_row = inner_iter.next
      end
   else
      output [ outer_row.col1, NULL ]
   end
   outer_row = outer_iter.next
end
```

Еще один способ визуализировать план выполнения запроса – воспользоваться тем, что разработчики оптимизаторов называют «диа-

граммой плавательных дорожек». На рис. 4.2 показана такая диаграмма для исходного запроса с INNER JOIN. Читать ее следует слева направо и сверху вниз.

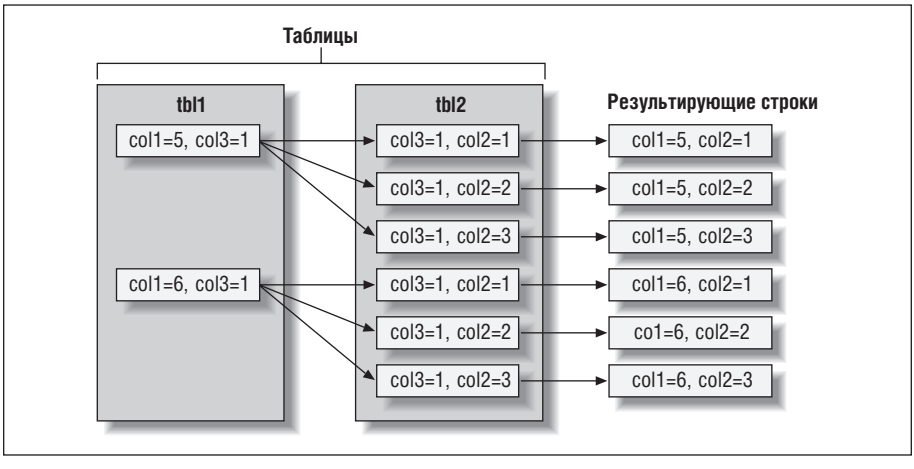


***Рис. 4.2.** Диаграмма плавательных дорожек, иллюстрирующая выборку строк при обработке соединения*

MySQL выполняет запросы любого вида практически одинаково. Например, подзапрос во фразе FROM выполняется первым, его результаты сохраняются во временной таблице[1], которая затем трактуется как самая обычная таблица (отсюда и название «производная таблица» (derived table)). При обработке запросов с UNION тоже используются временные таблицы, а запросы с RIGHT OUTER JOIN преобразуются в эквивалентную форму с LEFT OUTER JOIN. Короче говоря, MySQL приводит запросы любого вида к этому плану выполнения.

Но таким способом можно выполнить не все допустимые SQL-запросы. Например, запрос, включающий полное внешнее соединение (FULL OUTER JOIN), не удастся выполнить методом вложенных циклов с возвратами, коль скоро обнаружится таблица, не содержащая подходящих строк, поскольку алгоритм может начать работу именно с такой таблицы. Вот поэтому MySQL и не поддерживает оператор FULL OUTER JOIN. Есть и другие запросы, которые, хотя и можно выполнить методом вложенных циклов, но это оказывается крайне неэффективно. Мы рассмотрим такие примеры ниже.

[1] Над производными таблицами не строятся индексы, и об этом следует помнить при написании сложных соединений с результатами подзапросов. То же самое относится и к запросам с UNION.

План выполнения

В отличие от многих других СУБД, MySQL не генерирует байт-код для выполнения запроса. Вместо этого план представляет собой дерево инструкций, которые выполняются для получения результата. Окончательный план содержит достаточно информации, позволяющей реконструировать исходный запрос. Выполнив команду `EXPLAIN EXTENDED`, а вслед за ней команду `SHOW WARNINGS`, мы увидим реконструированный запрос[1].

Любой запрос с несколькими таблицами концептуально можно представить в виде дерева. Например, соединение четырех таблиц можно выполнить так, как показано на рис. 4.3.

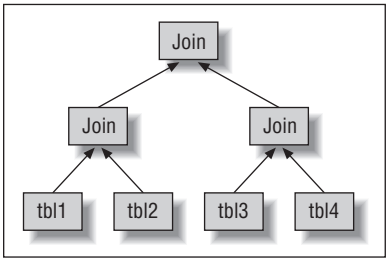


Рис. 4.3. *Один из способов соединить несколько таблиц*

Такое дерево в информатике называется *сбалансированным*. Но MySQL обрабатывает запрос иначе. В предыдущем разделе мы рассказали, что MySQL всегда начинает с какой-то одной таблицы и ищет соответствующие строки в следующей. Поэтому планы выполнения запросов, формируемые MySQL, всегда имеют вид, показанный на рис. 4.4. Такие деревья называются «левоглубокими» (left-deep tree).

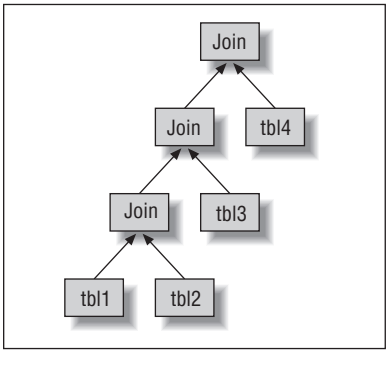


Рис. 4.4. *Как соединяются несколько таблиц в MySQL*

[1] Сервер генерирует выходную информацию, исходя из плана выполнения. Поэтому выведенный запрос будет иметь ту же семантику, что исходный, но текстуально оба запроса могут отличаться.

Оптимизатор соединений

Важнейшая часть оптимизатора запросов в MySQL – это оптимизатор соединений, который определяет наилучший порядок выполнения запросов с несколькими таблицами. Этот компонент вычисляет стоимости различных планов достижения искомого результата и пытается выбрать самый дешевый.

Вот пример запроса, в котором таблицы можно соединить в разном порядке с одним и тем же результатом:

```
mysql> SELECT film.film_id, film.title, film.release_year, actor.actor_id,
    ->    actor.first_name, actor.last_name
    ->    FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id)
    ->    INNER JOIN sakila.actor USING(actor_id);
```

Нетрудно придумать несколько подходящих планов выполнения. Например, MySQL мог бы начать с таблицы `film`, воспользовавшись индексом таблицы `film_actor` по полю `film_id` для поиска значений `actor_id`, а затем искать строки в индексе, построенном над таблицей `actor` по первичному ключу. Это должно быть эффективно, не правда ли? Однако посмотрим на вывод команды `EXPLAIN`, объясняющей, как MySQL в действительности собирается выполнять этот запрос:

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 200
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: PRIMARY,idx_fk_film_id
          key: PRIMARY
      key_len: 2
          ref: sakila.actor.actor_id
         rows: 1
        Extra: Using index
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
```

```
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: sakila.film_actor.film_id
         rows: 1
        Extra:
```

Это совсем не тот план, что был предложен выше. MySQL хочет начать с таблицы `actor` (мы знаем это, потому что она идет первой в списке, выданном командой `EXPLAIN`) и двигаться в обратном порядке. Действительно ли это более эффективно? Давайте разберемся. Ключевое слово `STRAIGHT_JOIN` заставляет сервер выполнять соединение в той последовательности, которая указана в запросе. Вот что говорит `EXPLAIN` о таком модифицированном запросе:

```
mysql> EXPLAIN SELECT STRAIGHT_JOIN film.film_id...\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 951
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: PRIMARY,idx_fk_film_id
          key: idx_fk_film_id
      key_len: 2
          ref: sakila.film.film_id
         rows: 1
        Extra: Using index
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: sakila.film_actor.actor_id
         rows: 1
        Extra:
```

Теперь мы видим, почему MySQL предпочла обратный порядок соединения: это позволяет исследовать меньше строк в первой таблице[1]. В обоих случаях во второй и третьей таблицах можно вести быстрый поиск по индексу. Разница лишь в том, сколько придется выполнить таких операций поиска.

- Если начать с таблицы `film`, то потребуется 951 обращение к таблицам `film_actor` и `actor`, по одному для каждой строки в первой таблице.
- Если же сначала искать в таблице `actor`, то для последующих таблиц придется выполнить всего 200 операций поиска по индексу.

Таким образом, изменив порядок соединения, можно уменьшить количество возвратов и повторных операций чтения. Чтобы убедиться в правильности выбранного оптимизатором решения, мы выполнили оба варианта запроса и посмотрели на значение переменной `Last_query_cost`. Оценка стоимости запроса с измененным порядком соединения составила 241, а запроса с навязанным нами порядком – 1154.

Это простой пример того, как оптимизатор соединений в MySQL может добиваться более низкой стоимости выполнения, изменяя порядок выполнения запросов. Изменение порядка соединений обычно оказывается весьма эффективной оптимизацией. Иногда она не дает оптимальный план, и в этих случаях вы можете употребить ключевое слово `STRAIGHT_JOIN`, чтобы записать запрос в том порядке, который вам представляется наилучшим, но такие случаи редки. Как правило, оптимизатор оказывается эффективнее человека.

Оптимизатор соединений пытается построить дерево плана выполнения запроса с минимально достижимой стоимостью. Если возможно, он исследует все потенциальные перестановки поддеревьев.

К сожалению, для полного исследования соединения n таблиц нужно перебрать n-факториал возможных перестановок. Это называется *пространством поиска* всех возможных планов выполнения, и его размер растет очень быстро – соединить 10 таблиц можно 3 628 800 способами! Если пространство поиска оказывается слишком большим, то для оптимизации запроса потребуется чересчур много времени, поэтому сервер отказывается от полного анализа. Если количество соединяемых таблиц превышает значение параметра `optimizer_search_depth`, то сервер применяет сокращенные алгоритмы, например «жадный» поиск.

Для ускорения стадии оптимизации в MySQL реализовано множество эвристических приемов, выработанных за годы экспериментов. Это, конечно, полезно, но означает также, что иногда (в редких случаях) MySQL может пропустить оптимальный план и выбрать менее удачный, поскольку СУБД не пыталась исследовать все возможные планы.

---

[1] Строго говоря, **MySQL не стремится минимизировать количество прочитанных строк**. Она пытается оптимизировать количество считываемых страниц. Однако счетчик строк часто дает грубое представление о стоимости запроса.

Бывает так, что запросы нельзя выполнить в другом порядке, тогда, воспользовавшись этим фактом, оптимизатор соединений может сократить пространство поиска, исключив из него некоторые перестановки. Хорошим примером могут служить запросы с LEFT JOIN и коррелированные подзапросы (к подзапросам мы еще вернемся). Объясняется это тем, что результаты для одной таблицы зависят от данных, извлеченных из другой. Наличие таких зависимостей позволяет оптимизатору соединений сократить пространство поиска.

Оптимизации сортировки

Сортировка результатов может оказаться дорогостоящей операцией, поэтому зачастую производительность можно повысить, избежав сортировки вовсе или уменьшив количество сортируемых строк.

В главе 3 мы показали, как индексы могут быть использованы для сортировки. Если MySQL не может найти подходящего индекса, то ей приходится сортировать строки самостоятельно. Это можно сделать в памяти или на диске, но сама процедура всегда называется *файловой сортировкой* (filesort), пусть даже в действительности файл не используется.

Если обрабатываемые данные умещаются в буфер, то MySQL может выполнить сортировку целиком в памяти, применяя алгоритм *быстрой сортировки* (quicksort). В противном случае сортировка выполняется на диске поблочно. Каждый блок обрабатывается методом быстрой сортировки, а затем уже отсортированные блоки сливаются.

Существует два алгоритма файловой сортировки.

Двухпроходный (старый)

Читает указатели на строки и столбцы, упомянутые во фразе ORDER BY, сортирует их, затем проходит по отсортированному списку и снова читает исходные строки, чтобы вывести результат.

Двухпроходный алгоритм обходится довольно дорого, поскольку читает строки из таблицы дважды, и второе чтение вызывает много непоследовательных операций ввода/вывода. Особенно накладно это в случае таблиц типа MyISAM, когда для выборки каждой строки необходим вызов операционной системы (в вопросах кэширования данных MyISAM полагается ОС). С другой стороны, для сортировки этим способом используется минимальный объем памяти, поэтому, если все сортируемые строки уже находятся в ОЗУ, то может оказаться дешевле хранить меньше данных и перечитывать строки для генерации окончательного результата.

Однопроходный (новый)

Читает все необходимые запросу столбцы, сортирует строки по столбцам, упомянутым во фразе ORDER BY, проходит по отсортированному списку и выводит заданные столбцы.

Этот алгоритм реализован, начиная с версии MySQL 4.1. Он может показывать гораздо более высокую скорость, особенно на больших наборах данных. Однопроходный алгоритм не читает строки из таблицы дважды, а случайный ввод/вывод в нем заменяется последовательным чтением. Но при этом необходимо больше памяти, так как для каждой строки приходится хранить все запрошенные столбцы, а не только те, по которым производится сортировка. Следовательно, в буфер сортировки поместится меньше строк и надо будет выполнить больше циклов слияния.

Для файловой сортировки серверу MySQL может потребоваться гораздо больше места на диске, чем вы ожидаете, с целью хранения временных файлов, так как для каждого сортируемого кортежа выделяется запись фиксированной длины. Ее размер должен быть достаточен, чтобы сохранить максимально длинный кортеж, в котором для столбцов типа `VARCHAR` зарезервировано место в соответствии с указанным в схеме размером. Кроме того, для кодировки UTF-8 MySQL выделяет на каждый символ три байта. Так что нам доводилось встречать плохо оптимизированные схемы, для которых размер временного файла сортировки во много раз превышал размер исходной таблицы на диске.

При выполнении соединения MySQL может производить файловую сортировку на двух стадиях выполнения запроса. Если во фразе `ORDER BY` упомянуты только столбцы из первой (в порядке соединения) таблицы, то MySQL может отсортировать ее, а затем приступить к соединению. В таком случае команда `EXPLAIN` поместит в столбец `Extra` фразу «Using temporary; Using filesort» (используется временная таблица, используется файловая сортировка). Если задана фраза `LIMIT`, то она применяется после сортировки, поэтому временная таблица может быть очень велика.

Дополнительную информацию о том, как настроить сервер для выполнения быстрой сортировки и оказать влияние на выбор алгоритма, см. в разделе «Оптимизация для сортировки» на стр. 229.

Подсистема выполнения запросов

На стадиях разбора и оптимизации вырабатывается план выполнения запроса, который затем передается подсистеме выполнения. План представляет собой структуру данных, а не исполняемый байт-код, как во многих других СУБД.

Как правило, стадия выполнения гораздо проще, чем стадия оптимизации: MySQL просто следует инструкциям, содержащимся в плане. Для многих указанных в плане операций нужно вызывать методы, реализованные в подсистеме хранения; интерфейс с этой подсистемой еще называют *API обработчика* (*handler API*). Каждая таблица в запросе представлена экземпляром обработчика. Если таблица встречается в запросе трижды, то сервер создаст три экземпляра обработчика. Хотя раньше

ше мы об этом не упоминали, фактически MySQL создает экземпляры обработчика на ранних шагах стадии оптимизации. Оптимизатор использует эти экземпляры для получения информации о таблицах, например об именах столбцов и статистике по индексам.

В интерфейсе подсистемы хранения определено множество функций, но для выполнения большинства запросов хватает примерно десятка основных операций-«кирпичиков». Например, имеются операции для чтения первой и следующей строки в индексе. Этого вполне достаточно для запроса, в котором производится просмотр индекса. Благодаря такому упрощенному подходу к выполнению запроса и стала возможной архитектура подсистемы хранения в MySQL, но это, в свою очередь, налагает определенные ограничения на оптимизатор.

Не все, что есть в плане выполнения запроса, является операциями обработчика. Например, табличными блокировками управляет сам сервер. Обработчик может реализовать собственную схему блокировки на уровне строк, как делает, например, InnoDB, но она не заменяет реализацию блокировок внутри сервера. В главе 1 отмечалось, что функции, общие для всех подсистем хранения, реализованы в сервере, например функции работы с датой и временем, представления и триггеры.

Для выполнения запроса сервер просто повторяет инструкции, пока не будут проанализированы все строки.

Возврат результатов клиенту

Последняя стадия выполнения запроса – отправка ответа клиенту. Даже для тех запросов, которые не возвращают результирующий набор, сервер должен послать клиенту информацию о результате обработки, например сообщить, сколько строк было изменено.

Если запрос можно кэшировать, то на этой стадии MySQL помещает результаты в кэш запросов.

Сервер генерирует и отправляет данные инкрементно. Давайте вернемся к рассмотренному выше алгоритму соединения нескольких таблиц. Как только сервер обработает последнюю таблицу и успешно сгенерирует очередную строку, он может и должен отправить ее клиенту. У такого решения есть два достоинства: серверу не обязательно хранить строку в памяти, а клиент начинает получать результаты настолько быстро, насколько это вообще возможно[1].

[1] При желании это поведение можно изменить, например, включив в запрос указание SQL_BUFFER_RESULT. См. раздел «Подсказки оптимизатору запросов» на стр. 250.

Ограничения оптимизатора MySQL

Подход MySQL к выполнению запросов по принципу «всякий запрос – это соединение методом вложенных циклов» идеален не для всех видов запросов. К счастью, есть не так уж много случаев, с которыми оптимизатор MySQL справляется заведомо плохо, и обычно удается переписать такие запросы более эффективно.

Приведенная в этом разделе информация относится к тем версиям MySQL, которые были доступны на момент работы над этой книгой, т. е. вплоть до версии MySQL 5.1. Возможно, в будущих редакциях некоторые ограничения будут ослаблены или вообще сняты, а кое-какие уже исправлены в версиях, которые существуют, но официально еще не выпущены. В частности, в исходный код MySQL 6 включен ряд оптимизаций подзапросов, и работа над ними продолжается.

Коррелированные подзапросы

Иногда MySQL очень плохо оптимизирует подзапросы. Самый неприятный случай – подзапросы в операторе `IN()` во фразе `WHERE`. Давайте, к примеру, найдем в демонстрационной базе Sakila все фильмы, в которых играла Пенелопа Гинесс (`actor_id=1`). На первый взгляд, кажется естественным написать такой подзапрос:

```
mysql> SELECT * FROM sakila.film
    -> WHERE film_id IN(
    ->    SELECT film_id FROM sakila.film_actor WHERE actor_id = 1);
```

Хочется думать, что MySQL будет выполнять этот запрос «изнутри наружу», то есть сначала найдет список значений `film_id` с заданным `actor_id`, а затем подставит их в список `IN()`. Выше мы отмечали, что в общем случае списки `IN()` обрабатываются очень быстро, поэтому можно ожидать, что запрос будет оптимизирован следующим образом:

```
-- SELECT GROUP_CONCAT(film_id) FROM sakila.film_actor WHERE actor_id = 1;
-- Result: 1,23,25,106,140,166,277,361,438,499,506,509,605,635,749,832,939,
970,980
SELECT * FROM sakila.film
WHERE film_id
IN(1,23,25,106,140,166,277,361,438,499,506,509,605,635,749,832,939,970,980);
```

К сожалению, на деле имеет место нечто прямо противоположное. MySQL пытается «помочь» подзапросу, перенеся в него корреляцию из внешней таблицы. Сервер полагает, что так подзапрос будет искать строки более эффективно. В результате запрос переписывается следующим образом:

```
SELECT * FROM sakila.film
WHERE EXISTS (
   SELECT * FROM sakila.film_actor WHERE actor_id = 1
   AND film_actor.film_id = film.film_id);
```

Теперь подзапросу необходимо значение поля film_id из внешней таблицы film, поэтому первым его выполнить не получится. EXPLAIN показывает в качестве типа запроса DEPENDENT SUBQUERY (воспользовавшись командой EXPLAIN EXTENDED, вы можете точно узнать, как был переписан запрос).

```
mysql> EXPLAIN SELECT * FROM sakila.film ...;
+----+--------------------+------------+--------+-----------------------+
| id | select_type        | table      | type   | possible_keys         |
+----+--------------------+------------+--------+-----------------------+
|  1 | PRIMARY            | film       | ALL    | NULL                  |
|  2 | DEPENDENT SUBQUERY | film_actor | eq_ref | PRIMARY,idx_fk_film_id |
+----+--------------------+------------+--------+-----------------------+
```

Согласно результатам EXPLAIN MySQL производит полное сканирование таблицы film и для каждой найденной строки выполняет подзапрос. В случае с небольшими таблицами это не приведет к заметному падению производительности, но если внешняя таблица велика, то замедление окажется весьма существенным. Впрочем, такой запрос легко переписать с использованием оператора JOIN:

```
mysql> SELECT film.* FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id)
    -> WHERE actor_id = 1;
```

Еще одна неплохая оптимизация – вручную сгенерировать список IN(), выполнив вместо подзапроса отдельный запрос с функцией GROUP_CONCAT(). Иногда это оказывается быстрее, чем JOIN.

MySQL не раз подвергалась суровой критике именно за планы выполнения подзапросов такого вида. Хотя этот недочет, безусловно, следует исправить, критики часто путают две разные проблемы: порядок выполнения и кэширование. Выполнение запроса «изнутри наружу» – один из способов его оптимизации, кэширование результата внутреннего запроса – другой. Самостоятельное переписывание запроса в другой форме позволяет вам контролировать оба аспекта. В будущих версиях MySQL оптимизация запросов такого вида, скорее всего, улучшится, хотя это и непростая задача. Для любого плана выполнения существуют граничные случаи, когда поведение оказывается очень плохим; это касается и плана «изнутри наружу», который, как многим кажется, будет несложно оптимизировать.

Когда коррелированный подзапрос – благо

Не всегда MySQL так уж плохо оптимизирует коррелированные подзапросы. Если вам рекомендуют любой ценой избегать их, не слушайте! Измеряйте производительность и принимайте решение самостоятельно. Иногда коррелированный подзапрос – вполне приемлемый и даже оптимальный способ получения результата. Рассмотрим пример:

```
mysql> EXPLAIN SELECT film_id, language_id FROM sakila.film
    -> WHERE NOT EXISTS(
```

```
    ->    SELECT * FROM sakila.film_actor
    ->    WHERE film_actor.film_id = film.film_id
    -> )\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: film
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 951
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: film_actor
         type: ref
possible_keys: idx_fk_film_id
          key: idx_fk_film_id
      key_len: 2
          ref: film.film_id
         rows: 2
        Extra: Using where; Using index
```

Обычно рекомендуют записывать такой запрос с помощью LEFT OUTER JOIN, а не подзапроса. Теоретически в обоих случаях план выполнения должен быть одинаков. Посмотрим.

```
mysql> EXPLAIN SELECT film.film_id, film.language_id
    -> FROM sakila.film
    ->    LEFT OUTER JOIN sakila.film_actor USING(film_id)
    -> WHERE film_actor.film_id IS NULL\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 951
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
```

```
possible_keys: idx_fk_film_id
          key: idx_fk_film_id
      key_len: 2
          ref: sakila.film.film_id
         rows: 2
        Extra: Using where; Using index; Not exists
```

Планы почти идентичны, но различия все же есть.

- Тип `SELECT` для таблицы `film_actor` равен `DEPENDENT SUBQUERY` в одном запросе и `SIMPLE` – в другом. Это различие просто отражает синтаксис, так как в первом запросе используется подзапрос, а во втором – нет. С точки зрения операций обработчика это не слишком существенно.
- В столбце `Extra` для второго запроса нет фразы «Using where». Но и это неважно, поскольку фраза `USING` во втором запросе реализует ту же семантику, что и `WHERE`.
- В столбце `Extra` для второго запроса стоит «Not exists» (не существует). Это пример работы алгоритма раннего завершения, который мы упоминали выше, он означает, что MySQL применил оптимизацию типа «не существует», чтобы не читать более одной строки из индекса `idx_fk_film_id` над таблицей `film_actor`. Это эквивалентно коррелированному подзапросу `NOT EXISTS( )`, поскольку обработка текущей строки прекращается, как только ей найдено соответствие.

Таким образом, в теории MySQL выполняет запросы почти одинаково. Но на практике только измерение может показать, какой способ быстрее. Мы провели эталонное тестирование в стандартной конфигурации системы. Результаты сведены в табл. 4.1.

Таблица 4.1. Сравнение NOT EXISTS и LEFT OUTER JOIN

Запрос	Результаты (запросов/сек) (QPS)
Подзапрос `NOT EXISTS`	360 QPS
`LEFT OUTER JOIN`	425 QPS

Измерение показало, что подзапрос немного медленнее!

Однако так бывает не всегда. Например, подзапрос может оказаться вполне приемлемым, если вам нужно только увидеть строки из одной таблицы, которые соответствуют строкам в другой таблице. Хотя это описание, на первый взгляд, в точности совпадает с определением соединения, на самом деле это не всегда одно и то же. Следующий запрос с соединением предназначен для поиска фильмов, в которых есть хотя бы один актер, но он возвращает дубликаты, поскольку в некоторых фильмах играют несколько актеров:

```
mysql> SELECT film.film_id FROM sakila.film
    ->     INNER JOIN sakila.film_actor USING(film_id);
```

Чтобы устранить дубликаты, нужно включить в запрос фразу DISTINCT или GROUP BY.

```
mysql> SELECT DISTINCT film.film_id FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id);
```

Но что же мы на самом деле хотим выразить этим запросом, и так ли очевидно наше намерение из SQL-кода? Оператор EXISTS выражает логическую идею «имеет соответствие», не порождая строк-дубликатов и не требуя операций DISTINCT или GROUP BY, для выполнения которых может понадобиться временная таблица. Вот как этот запрос можно переписать с использованием подзапроса вместо соединения:

```
mysql> SELECT film_id FROM sakila.film
    ->    WHERE EXISTS(SELECT * FROM sakila.film_actor
    ->    WHERE film.film_id = film_actor.film_id);
```

И снова мы протестировали обе стратегии, чтобы выявить более эффективную. Результаты показаны в табл. 4.2.

Таблица 4.2. Сравнение EXISTS и INNER JOIN

Запрос	Результаты (запросов/сек) (QPS)
INNER JOIN	185 QPS
Подзапрос EXISTS	325 QPS

В этом примере подзапрос выполняется намного быстрее, чем соединение.

Мы привели этот длинный пример, чтобы проиллюстрировать две вещи: во-первых, не стоит слепо доверять категорическим заявлениям относительно неприемлемости подзапросов, а, во-вторых, тестируйте, чтобы убедиться в верности своих предположений относительно планов и скорости выполнения запросов.

Ограничения UNION

Иногда MySQL не может «опустить» условия с внешнего уровня UNION на внутренний, где их можно было бы использовать с целью ограничения результата или создания возможностей для дополнительных оптимизаций. Если вы полагаете, что какой-то из запросов, составляющих UNION, будет выполняться быстрее при наличии LIMIT или если вы знаете, что после объединения с другими запросами результаты будут подвергнуты сортировке, то имеет смысл поместить соответствующие фразы в каждую часть UNION. Например, в ситуации, когда вы объединяете две очень большие таблицы и ограничиваете результат первыми 20 строками, MySQL сначала запишет обе таблицы во временную, а из нее выберет всего 20 строк. Этого можно избежать, включив ограничение LIMIT 20 в каждую часть UNION.

Оптимизация слияния индексов

Алгоритмы слияния индексов (index merge), появившиеся в версии MySQL 5.0, позволяют использовать при выполнении запроса несколько индексов по одной таблице. В более ранних версиях можно было задействовать только один индекс, а если ни один отдельный индекс не подходил для учета всех условий во фразе WHERE, то MySQL частенько выбирала полное сканирование таблицы. Например, по каждому из столбцов film_id и actor_id таблицы film_actor построены индексы, но ни один их них не позволяет эффективно обработать оба условия в следующем запросе:

```
mysql> SELECT film_id, actor_id FROM sakila.film_actor
    -> WHERE actor_id = 1 OR film_id = 1;
```

В прежних версиях MySQL этот запрос привел бы к полному сканированию таблицы, если не переписать его в виде объединения (UNION) двух запросов:

```
mysql> SELECT film_id, actor_id FROM sakila.film_actor WHERE actor_id = 1
    -> UNION ALL
    -> SELECT film_id, actor_id FROM sakila.film_actor WHERE film_id = 1
    ->    AND actor_id <> 1;
```

В MySQL 5.0 и старше при выполнении запроса можно использовать оба индекса: они просматриваются одновременно, после чего результаты сливаются. Существует три варианта этого алгоритма: объединение для условий с OR, пересечение для условий с AND и объединение пересечений для случая, когда встречаются как OR, так и AND. В следующем примере производится объединение результатов просмотра двух индексов, в чем можно убедиться, посмотрев на содержимое столбца Extra:

```
mysql> EXPLAIN SELECT film_id, actor_id FROM sakila.film_actor
    -> WHERE actor_id = 1 OR film_id = 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: index_merge
possible_keys: PRIMARY,idx_fk_film_id
          key: PRIMARY,idx_fk_film_id
      key_len: 2,2
          ref: NULL
         rows: 29
        Extra: Using union(PRIMARY,idx_fk_film_id); Using where
```

MySQL умеет применять эту технику к сложным фразам WHERE, поэтому для некоторых запросов в столбце Extra можно встретить вложенные операции. Результат зачастую бывает очень хорошим, но иногда для выполнения операций буферизации, сортировки и слияния требу-

ется довольно много ресурсов ЦП и памяти. Так, в частности, обстоит дело, когда селективность некоторых индексов оставляет желать лучшего и, следовательно, в результате параллельных просмотров возвращается много строк, подлежащих слиянию. Напомним, что оптимизатор не включает этот расход в стоимость, он старается минимизировать лишь количество операций чтения случайных страниц. Поэтому запрос может оказаться «недооцененным» и будет выполняться даже медленнее, чем полное сканирование таблицы. К тому же интенсивное потребление памяти и ЦП может оказать влияние на одновременно выполняющиеся запросы, но, запустив данный запрос изолированно, вы этого влияния не заметите. Вот и еще одна причина производить эталонное тестирование в реальных условиях.

Если ваш запрос работает медленнее, чем ожидается, из-за этой особенности оптимизатора, то можно попробовать обходной путь – запретить использование некоторых индексов с помощью подсказки `IGNORE INDEX` или просто прибегнуть к старой тактике декомпозиции запроса с помощью `UNION`.

Распространение равенства

Иногда распространение равенства может иметь неожиданные последствия для стоимости. Рассмотрим, к примеру, гигантский список `IN()` для некоторого столбца, относительно которого оптимизатору известно, что он равен каким-то другим столбцам в других таблицах, – благодаря наличию фраз `WHERE`, `ON` или `USING`, устанавливающих равенство столбцов.

Оптимизатор прибегнет к «разделению» списка, скопировав его во все связанные столбцы в соединенных таблицах. Обычно это полезно, поскольку предоставляет оптимизатору запросов и подсистеме выполнения больше свободы в определении того, где именно выполнять проверку сравнения со списком `IN()`. Но, если список очень велик, то и оптимизация, и выполнение могут занять больше времени. Нам неизвестен никакой обходной путь решения этой проблемы; если вы с ней столкнулись, придется изменить исходный код (для большинства разработчиков это не составляет труда).

Параллельное выполнение

MySQL не умеет распараллеливать выполнение одного запроса на нескольких ЦП. Эту возможность предлагают многие СУБД, но только не MySQL. Мы упоминаем о ней лишь для того, чтобы вы не тратили время в попытках понять, как же заставить MySQL выполнять запрос параллельно!

Хеш-соединения

В настоящее время MySQL не умеет выполнять настоящие хеш-соединения, любое соединение производится методом вложенных циклов. Но

можно эмулировать хеш-соединения с помощью хеш-индексов. Правда, если вы работаете не с подсистемой хранения Memory, то и сами хеш-индексы придется эмулировать. Как это делается, мы продемонстрировали в разделе «Построение собственных хеш-индексов» на стр. 143.

Непоследовательный просмотр индекса

Исторически MySQL никогда не умела выполнять непоследовательный просмотр индекса (loose index scan), то есть просмотр несмежных диапазонов индекса. При просмотре индекса всегда необходимо было задавать начальную и конечную точку, даже если для запроса нужно всего несколько несмежных строк в середине диапазона. MySQL просматривает весь диапазон строк между двумя заданными точками.

Поясним это на примере. Предположим, что имеется таблица с индексом, построенным по столбцам (a, b), и мы хотим выполнить такой запрос:

```
mysql> SELECT ... FROM tbl WHERE b BETWEEN 2 AND 3;
```

Поскольку ключ индекса начинается со столбца a, а в условии WHERE этот столбец не фигурирует, то MySQL вынуждена прибегнуть к полному сканированию таблицы, чтобы исключить неподходящие строки (рис. 4.5).

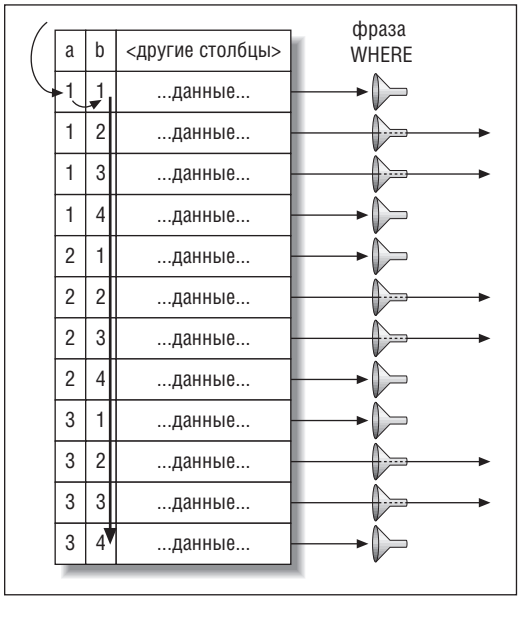


Рис. 4.5. *MySQL сканирует всю таблицу в поисках нужных строк*

Легко видеть, что существует более быстрый способ выполнения этого запроса. Структура индекса (но не API подсистемы хранения MySQL) позволяет найти начало каждого диапазона значений, просмотреть

этот диапазон до конца, затем вернуться и перейти в начало следующего диапазона. На рис. 4.6 показано, как выглядела бы такая стратегия, если бы MySQL мог претворить ее в жизнь.

Обратите внимание на отсутствие фразы `WHERE`, которая здесь ни к чему, потому что сам индекс позволяет пропустить ненужные строки (повторим, MySQL пока не умеет это делать).

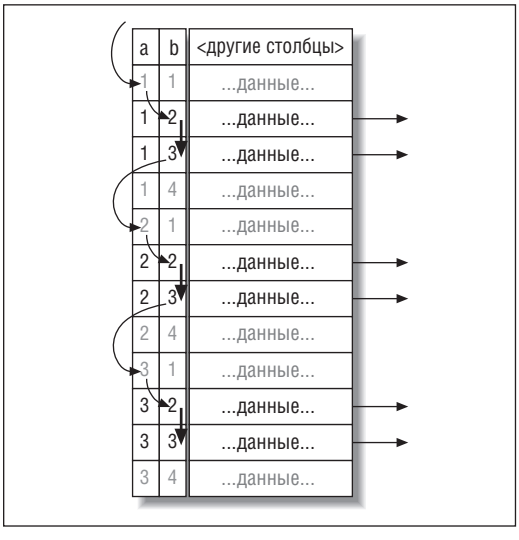


Рис. 4.6. *Непоследовательный просмотр индекса, который в MySQL пока не реализован, позволил бы выполнить запрос более эффективно*

Конечно, это упрощенный пример; данный запрос легко было бы оптимизировать, добавив еще один индекс. Однако существует немало случаев, когда добавление индекса не решает проблему. Например, запрос, в котором для первого индексного столбца задано условие в виде диапазона, а для второго – сравнение на равенство.

Начиная с версии MySQL 5.0, непоследовательный просмотр индекса стал возможен в некоторых ситуациях, например для отыскания минимального и максимального значений в запросе с группировкой:

```
mysql> EXPLAIN SELECT actor_id, MAX(film_id)
    -> FROM sakila.film_actor
    -> GROUP BY actor_id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: range
possible_keys: NULL
          key: PRIMARY
      key_len: 2
```

```
          ref: NULL
         rows: 396
        Extra: Using index for group-by
```

Наличие слов «Using index for group-by» (Используется индекс для group-by) в плане, представленном командой EXPLAIN, свидетельствует о непоследовательном просмотре индекса. Для данного частного случая эта оптимизация хороша, но назвать ее универсальным алгоритмом непоследовательного просмотра индекса невозможно. Лучше подошел бы термин «непоследовательное взятие проб из индекса» (loose index probe).

До тех пор пока в MySQL не будет реализован универсальный алгоритм, можно применять обходное решение: задавать константу или список констант для столбца, указанного первым в ключе индекса. В предыдущей главе мы привели несколько примеров, показывающих, как этот подход позволяет добиться неплохой производительности.

Функции MIN() и MAX()

MySQL не очень хорошо оптимизирует некоторые запросы, содержащие агрегатные функции MIN() и MAX(). Вот пример:

```
mysql> SELECT MIN(actor_id) FROM sakila.actor WHERE first_name = 'PENELOPE';
```

Поскольку по столбцу first_name нет индекса, то этот запрос приводит к полному просмотру таблицы. Когда MySQL сканирует первичный ключ, теоретически можно прекратить поиск после нахождения первой подходящей строки, так как значения первичного ключа строго возрастают, и, значит, во всех последующих строках значение actor_id будет больше уже встретившегося. Однако в этом случае MySQL продолжит сканирование до конца, в чем легко убедиться с помощью профилирования запроса. Обходное решение – исключить MIN() и переписать запрос с применением LIMIT:

```
mysql> SELECT actor_id FROM sakila.actor USE INDEX(PRIMARY)
    -> WHERE first_name = 'PENELOPE' LIMIT 1;
```

Эта общая стратегия зачастую неплохо работает в тех случаях, когда без нее MySQL стала бы просматривать больше строк, чем необходимо. Пуристы могут возразить, что такой запрос входит в противоречие с самой идеей SQL. Предполагается, что мы говорим серверу, *что* мы хотим получить, а он уж решает, *как* найти нужные данные. В данном же случае мы говорим MySQL, *как* выполнять запрос, при этом из самого запроса неясно, *что* мы ищем минимальное значение. Это все правда, но иногда для достижения высокой производительности приходится поступаться принципами.

SELECT и UPDATE для одной и той же таблицы

MySQL не позволяет производить выборку из таблицы (SELECT) одновременно с ее обновлением (UPDATE). На самом деле, это не ограничение опти-

мизатора, но, зная, как MySQL выполняет запросы, вы сможете обойти проблему. Ниже приведен пример запроса, который написан в полном соответствии со стандартом SQL, но в MySQL недопустим. Этот запрос обновляет каждую строку, записывая в нее количество похожих строк в той же таблице:

```
mysql> UPDATE tbl AS outer_tbl
    ->    SET cnt = (
    ->       SELECT count(*) FROM tbl AS inner_tbl
    ->       WHERE inner_tbl.type = outer_tbl.type
    ->    );
ERROR 1093 (HY000): You can't specify target table
'outer_tbl' for update in FROM clause
```

Чтобы обойти это ограничение, можно воспользоваться производной таблицей, так как MySQL материализует ее в виде временной таблицы. В результате выполняются два запроса: `SELECT` в подзапросе и `UPDATE` с двумя таблицами – соединением исходной и результатом подзапроса. Подзапрос открывает и закрывает таблицу перед тем, как внешний `UPDATE` открывает таблицу, поэтому запрос проходит успешно.

```
mysql> UPDATE tbl
    ->    INNER JOIN(
    ->       SELECT type, count(*) AS cnt
    ->       FROM tbl
    ->       GROUP BY type
    ->    ) AS der USING(type)
    -> SET tbl.cnt = der.cnt;
```

Оптимизация запросов конкретных типов

В этом разделе мы дадим рекомендации по оптимизации некоторых частных случаев запросов. Большей частью эти темы рассмотрены в других местах книги, но мы хотели представить общий перечень типичных проблем оптимизации, чтобы потом на него можно было ссылаться.

Большинство советов в этом разделе зависят от версии MySQL, и к последующим версиям, возможно, будут неприменимы. Нет никаких препятствий тому, чтобы в будущем сервер сам бы смог производить некоторые или даже все описанные оптимизации.

Оптимизация запросов с COUNT()

Агрегатная функция `COUNT()` и способы оптимизации содержащих ее запросов – пожалуй, один из десяти хуже всего понимаемых аспектов MySQL. Поиск в Сети даст столько неправильной информации, что мы даже думать об этом не хотим.

Но прежде чем переходить к вопросу об оптимизации, неплохо бы понять, что в действительности делает функция `COUNT()`.

Что делает COUNT()

COUNT() – это особая функция, которая решает две очень разные задачи: подсчитывает *значения* и *строки*. Значение – это выражение, отличное от NULL (NULL означает отсутствие какого бы то ни было значения). Если указать имя столбца или какое-нибудь другое выражение в скобках, то COUNT( ) посчитает, сколько раз это выражение имеет значение (т. е. сколько раз оно не равно NULL). Многих это путает, отчасти потому, что вопрос о значениях и NULL сам по себе не прост. Если вы ощущаете потребность понять, как все это работает, рекомендуем прочитать хорошую книгу об основных SQL (интернет вовсе не всегда можно считать надежным источником информации по этому поводу.)

Вторая форма COUNT() просто подсчитывает количество строк в результирующем наборе. Так MySQL поступает, когда точно знает, что выражение внутри скобок не может быть равно NULL. Наиболее очевидный пример – выражение COUNT(*), специальная форма COUNT(), которая вовсе не сводится к подстановке вместо метасимвола * полного списка столбцов таблицы, как вы, возможно, подумали. На самом деле столбцы вообще игнорируются, а подсчитываются сами строки.

Одна из наиболее часто встречающихся ошибок – задание имени столбца в скобках, когда требуется подсчитать строки. Если вы хотите знать, сколько строк в результирующем наборе, *всегда* употребляйте COUNT(*). Тем самым вы недвусмысленно сообщите о своем намерении и избежите возможного падения производительности.

Мифы о MyISAM

Распространено неверное представление о том, что для таблиц типа MyISAM запросы, содержащие функцию COUNT(), выполняются очень быстро. Так-то оно так, но лишь в очень частном случае: COUNT(*) без фразы WHERE, то есть при подсчете общего количества строк в таблице. MySQL может оптимизировать такой запрос, поскольку подсистеме хранения в любой момент известно, сколько в таблице строк. Если MySQL знает, что столбец col не может содержать NULL, то СУБД оптимизирует и выражение COUNT(col), самостоятельно преобразовав его в COUNT(*).

MyISAM не обладает никакими магическими возможностями для подсчета строк, когда в запросе есть фраза WHERE, равно как и для подсчета значений, а не строк. Возможно, конкретный запрос она выполнит быстрее, чем другая подсистема хранения, а, возможно, и нет. Это зависит от множества факторов.

Простые оптимизации

Иногда оптимизацию COUNT(*) в MyISAM можно применить во благо, если требуется подсчитать все строки, кроме очень небольшого числа, при условии наличия высокоселективного индекса. В следующем примере мы воспользовались стандартной базой данных World, чтобы по-

казать, как можно эффективно подсчитать количество городов с идентификаторами больше 5. Можно было бы записать запрос так:

```
mysql> SELECT COUNT(*) FROM world.City WHERE ID > 5;
```

Выполнив профилирование этого запроса с помощью SHOW STATUS, вы обнаружите, что он просматривает 4079 строк. Если изменить условие на противоположное и вычесть количество городов с идентификаторами, меньшими либо равными 5, из общего количества, то число просмотренных строк сократится до пяти:

```
mysql> SELECT (SELECT COUNT(*) FROM world.City) - COUNT(*)
    -> FROM world.City WHERE ID <= 5;
```

В этом варианте читается меньше строк, поскольку на стадии оптимизации подзапрос преобразуется в константу, что и показывает EXPLAIN:

```
+----+-------------+-------+...+------+-----------------------------+
| id | select_type | table |...| rows | Extra                       |
+----+-------------+-------+...+------+-----------------------------+
| 1  | PRIMARY     | City  |...|    6 | Using where; Using index    |
| 2  | SUBQUERY    | NULL  |...| NULL | Select tables optimized away |
+----+-------------+-------+...+------+-----------------------------+
```

В списках рассылки и в IRC-каналах часто спрашивают, как с помощью одного запроса подсчитать, сколько раз встречаются несколько разных значений в одном столбце. Пусть, скажем, вы хотите написать запрос, подсчитывающий предметы разных цветов. Использовать OR (например, SELECT COUNT(color= 'blue' OR color= 'red') FROM items;) нельзя, потому что невозможно будет разделить счетчики разных цветов. Указать цвета во фразе WHERE (например, SELECT COUNT(*) FROM items WHERE color= 'blue' AND color= 'red';) тоже нельзя, поскольку цвета являются взаимно исключающими. Задачу, тем не менее, можно решить с помощью такого запроса:

```
mysql> SELECT SUM(IF(color = 'blue', 1, 0)) AS blue,
SUM(IF(color = 'red', 1, 0)) -> AS red FROM items;
```

А вот еще один эквивалентный запрос, в котором вместо функции SUM( ) используется COUNT( ). Он основан на том, что выражение не будет иметь значения, когда условие ложно:

```
mysql> SELECT COUNT(color = 'blue' OR NULL) AS blue, COUNT(color = 'red' OR
NULL) -> AS red FROM items;
```

Более сложные оптимизации

В общем случае запросы, содержащие COUNT(), с трудом поддаются оптимизации, поскольку обычно они должны подсчитать много строк (то есть «прошерстить» большой объем данных). Единственная возможность внутри самого сервера MySQL – воспользоваться покрывающим индексом (см. главу 3). Если этого недостаточно, рекомендуется

внести изменения в архитектуру приложения. Можно рассмотреть вариант заведения сводных таблиц (тоже описаны в главе 3) или применить внешнюю систему кэширования, скажем, *memcached*. Не исключено, что придется столкнуться с известной дилеммой: «быстро, точно, просто – выбирайте любые два варианта».

Оптимизация запросов с JOIN

Эта тема всплывает на протяжении всей книги, но несколько основополагающих моментов мы все же упомянем в настоящем разделе:

- Стройте индексы по столбцам, используемым во фразах ON или USING. Подробнее об индексировании см. в разделе «Основы индексирования» на стр. 135. При добавлении индексов учитывайте порядок соединения. Если таблицы A и B соединяются по столбцу C и оптимизатор решит, что их надо соединять в порядке B, A, то индексировать таблицу B необязательно. Неиспользуемые индексы только влекут за собой лишние накладные расходы. В общем случае следует индексировать только вторую таблицу в порядке соединения, если, конечно, индекс не нужен для каких-то других целей.
- Старайтесь, чтобы в выражениях `GROUP BY` и `ORDER BY` встречались столбцы только из одной таблицы, тогда у MySQL появится возможность воспользоваться для этой операции индексом.
- Будьте внимательны при переходе на новую версию MySQL, поскольку в разные моменты изменялись синтаксис соединения, приоритеты операторов и другие аспекты поведения. Конструкция, которая раньше была обычным соединением, иногда вдруг становится декартовым произведением (а это совершенно другой тип соединения, который возвращает иные результаты), а то и вовсе оказывается синтаксически некорректной.

Оптимизация подзапросов

Самая важная рекомендация, которую мы можем дать относительно подзапросов, – стараться по возможности использовать вместо них соединение, по крайней мере, в текущих версиях MySQL. Эту тему мы подробно рассматривали в настоящей главе.

Обработка подзапросов – предмет напряженного труда создателей оптимизатора. Ожидается, что в будущих версиях MySQL появятся новые идеи в этой области. Интересно будет посмотреть, какие из тех оптимизаций, что мы видели, останутся в коде релиза и насколько их код будет отличаться от предварительного. Стоит подчеркнуть, что совет «предпочитайте соединения» в будущих версиях может утратить актуальность. Сервер все время становится «умнее», и говорить ему, что делать, а не просто указывать, какой результат требуется получить, приходится все реже.

Оптимизация GROUP BY и DISTINCT

Во многих случаях MySQL оптимизирует эти два вида запросов схожим образом; по существу, зачастую он внутренне переключается между ними на стадии оптимизации. Как обычно, выполнение того и другого запроса можно ускорить при наличии подходящих индексов.

Если подходящего индекса не существует, то MySQL может применить одну из двух стратегий реализации GROUP BY: воспользоваться временной таблицей или прибегнуть к файловой сортировке. Для конкретного запроса более эффективным может оказаться тот или другой подход. Чтобы заставить оптимизатор выбрать нужный вам метод, включите в запрос подсказки SQL_BIG_RESULT или SQL_SMALL_RESULT.

Если нужна группировка по значению столбца, который извлекается при соединении из справочной таблицы, то обычно более продуктивно группировать по идентификатору из этой таблицы, а не по его значению. Например, следующий запрос написан не очень удачно:

```
mysql> SELECT actor.first_name, actor.last_name, COUNT(*)
    -> FROM sakila.film_actor
    ->    INNER JOIN sakila.actor USING(actor_id)
    -> GROUP BY actor.first_name, actor.last_name;
```

Эффективнее было бы переписать его в таком виде:

```
mysql> SELECT actor.first_name, actor.last_name, COUNT(*)
    -> FROM sakila.film_actor
    ->    INNER JOIN sakila.actor USING(actor_id)
    -> GROUP BY film_actor.actor_id;
```

Группировка по столбцу actor.actor_id может оказаться производительнее, чем по film_actor.actor_id. Чтобы выбрать правильное решение, профилируйте запрос и тестируйте его на реальных данных.

В этом примере используется тот факт, что имя и фамилия актера зависят от значения поля actor_id, поэтому результат получится одинаковым. Однако не всегда можно так беззаботно включить в список SELECT столбцы, по которым не производится группировка, в надежде получить тот же самый эффект. Более того, в конфигурационном файле сервера может быть задан параметр SQL_MODE, который вообще запрещает такой нестандартный режим. Чтобы обойти эту сложность, можно воспользоваться функциями MIN()или MAX(), если точно известно, что значения в группе различны, так как зависят от группируемого столбца, или если неважно, какое значение будет получено:

```
mysql> SELECT MIN(actor.first_name), MAX(actor.last_name), ...;
```

Сторонники чистоты идеи могли бы возразить, что группировка производится не по тому столбцу, что нужно, и они были бы правы. Фиктивные MIN( ) или MAX( ) – признак того, что запрос структурирован неправильно. Но иногда требуется, чтобы MySQL выполнила запрос макси-

мально быстро. Пуристы были бы удовлетворены такой формой записи этого запроса:

```
mysql> SELECT actor.first_name, actor.last_name, c.cnt
    -> FROM sakila.actor
    ->    INNER JOIN (
    ->       SELECT actor_id, COUNT(*) AS cnt
    ->       FROM sakila.film_actor
    ->       GROUP BY actor_id
    ->    ) AS c USING(actor_id) ;
```

Однако временами стоимость создания и заполнения временной таблицы, необходимой для обработки подзапроса, слишком высока по сравнению с мелким отступлением от принципов реляционной теории. Напомним, что временная таблица, создаваемая в процессе выполнения подзапроса, не имеет индексов.

В общем случае включать в список SELECT столбцы, по которым не производится группировка, – плохая идея, так как результаты получаются недетерминированными и легко могут измениться, если вы создадите другой индекс или оптимизатор выберет иную стратегию. Большинство встречавшихся нам запросов такого рода либо появились случайно (поскольку сервер не возражает) или в результате лености программиста, а не потому, что были специально задуманы ради оптимизации. Лучше явно выражать свои намерения. На самом деле, мы даже рекомендуем включать в конфигурационную переменную SQL_MODE режим ONLY_FULL_GROUP_BY, чтобы сервер выдавал сообщение об ошибке, а не разрешал писать плохие запросы.

MySQL автоматически упорядочивает результат запроса с группировкой по столбцам, перечисленным во фразе GROUP BY, если фраза ORDER BY явно не указана. Если для вас порядок не имеет значения, но вы видите, что в плане выполнения присутствует файловая сортировка (filesort), то можно включить фразу ORDER BY NULL, которая подавляет автоматическую сортировку. Можно также поместить сразу после GROUP BY необязательное ключевое слово DESC или ASC, задающее направление сортировки (по убыванию или по возрастанию).

Оптимизация GROUP BY WITH ROLLUP

Вариацией на тему группирующих запросов является суперагрегирование результатов. Для этого предназначена фраза WITH ROLLUP, но ее оптимизация может оставлять желать лучшего. С помощью команды EXPLAIN проверьте, как выполняется запрос, обращая особое внимание на то, производится ли группировка посредством файловой сортировки или создания временной таблицы; попробуйте убрать фразу WITH ROLLUP и посмотрите, будет ли применен тот же способ группировки. Возможно, придется принудительно указать метод группировки, включив в запрос вышеупомянутые подсказки.

Иногда оказывается эффективнее выполнить суперагрегирование в самом приложении, даже если для этого придется запросить у сервера больше строк. Можно также воспользоваться подзапросом, вложенным во фразу FROM, или создать временную таблицу для хранения промежуточных результатов.

Оптимизация LIMIT со смещением

Запросы, содержащие ключевые слова LIMIT и OFFSET, часто встречаются в системах, производящих разбиение на страницы, и неизменно в сочетании с ORDER BY. Полезно иметь индекс, поддерживающий нужное упорядочение, иначе серверу придется слишком часто прибегать к файловой сортировке.

Типичная проблема – слишком большое смещение. Если в запросе встречается фраза LIMIT 10000, 20, то сервер сгенерирует 10 020 строк и отбросит первые 10 000, а это очень дорого. В предположении, что доступ ко всем страницам производится с одинаковой частотой, такой запрос в среднем просматривает половину таблицы. Для оптимизации можно либо наложить ограничения на то, сколько страниц разрешено просматривать, или попытаться реализовать обработку больших смещений более эффективно.

Один из простых приемов повышения производительности – выполнять смещение, пользуясь покрывающим индексом, а не исходной таблицей. Затем полученные результаты можно соединить с полными строками, чтобы дополнительно выбрать интересующие вас столбцы. Такой подход может оказаться намного эффективнее. Рассмотрим следующий запрос:

```
mysql> SELECT film_id, description FROM sakila.film ORDER BY title LIMIT 50, 5;
```

Если таблица очень велика, то его лучше переписать в следующем виде:

```
mysql> SELECT film.film_id, film.description
    -> FROM sakila.film
    ->    INNER JOIN (
    ->       SELECT film_id FROM sakila.film
    ->       ORDER BY title LIMIT 50, 5
    ->    ) AS lim USING(film_id);
```

Выигрыш достигается за счет того, что серверу приходится просматривать только индекс, не обращаясь к самим строкам. А после того, как нужные строки найдены, они соединяются с исходной таблицей для выборки недостающих столбцов. Аналогичная техника применима к соединениям с фразой LIMIT.

Иногда можно также преобразовать запрос с LIMIT в позиционный запрос, который сервер сможет выполнить путем просмотра диапазона индекса. Например, если предварительно вычислить столбец position и построить по нему индекс, то предыдущий запрос можно будет переписать так:

```
mysql> SELECT film_id, description FROM sakila.film
    -> WHERE position BETWEEN 50 AND 54 ORDER BY position;
```

Похожая проблема возникает при ранжировании данных, причем обычно к этой задаче еще примешивается фраза GROUP BY. Почти наверняка вам придется заранее вычислять и сохранять ранги.

Если вы хотите по-настоящему оптимизировать разбиение на страницы, то имеет смысл предварительно вычислять итоги. В качестве альтернативы можно предложить соединение со вспомогательными таблицами, которые содержат только первичный ключ и столбцы, необходимые для выполнения ORDER BY. Можно также воспользоваться поисковой системой Sphinx (дополнительную информацию см. в приложении С).

Оптимизация SQL_CALC_FOUND_ROWS

Еще один широко распространенный способ оптимизации разбиения на страницы – включить в запрос с фразой LIMIT подсказку SQL_CALC_FOUND_ROWS, чтобы знать, сколько строк сервер вернул бы, если бы этой фразы не было. Может показаться, будто здесь скрыто какое-то волшебство, позволяющее серверу предсказать, сколько строк он смог бы найти. Увы, на деле все обстоит не так: сервер не умеет подсчитывать строки, которые не отбирал. Это ключевое слово означает лишь, что сервер должен сгенерировать и отбросить оставшуюся часть результирующего набора, а не останавливаться, выбрав затребованное количество строк. Это очень дорого.

Лучше включить в состав элемента разбиения на страницы ссылку «следующая». В предположении, что на странице выводится 20 результатов, мы отбираем с помощью фразы LIMIT 21 строку, а выводим только 20. Если 21-я строка существует, значит, имеется следующая страница, и мы формируем ссылку «следующая».

Еще одна возможность – выбрать и закэшировать намного больше строк, чем необходимо, скажем 1000, и генерировать последующие страницы из кэша. Такая стратегия позволяет приложению узнать размер полного результирующего набора. Если в нем меньше 1000 строк, то точно известно, сколько формировать ссылок на страницы; если больше, то можно просто вывести сообщение «найдено более 1000 результатов». Обе стратегии гораздо эффективнее, чем генерация полного набора с последующим отбрасыванием значительной его части.

Даже если вы не можете воспользоваться ни одним из описанных выше приемов, выполнение отдельного запроса COUNT(*) для нахождения количества строк может оказаться намного быстрее режима SQL_CALC_FOUND_ROWS, если удастся воспользоваться покрывающим индексом.

Оптимизация UNION

MySQL всегда выполняет запросы с UNION путем создания и заполнения временной таблицы. К подобным запросам MySQL может применить

не так уж много оптимизаций. Но вы в состоянии помочь оптимизатору, «опустив вниз» фразы WHERE, LIMIT, ORDER BY и другие условия (то есть скопировав их из внешнего запроса в каждый SELECT, входящий в объединение).

Очень важно *всегда* употреблять UNION ALL, если только вы не хотите, чтобы сервер устранял строки-дубликаты. Когда ключевое слово ALL отсутствует, MySQL будет создавать временную таблицу в режиме distinct, а это значит, что для соблюдения уникальности производится сравнение строк целиком. Такая операция обойдется очень недешево. Однако имейте в виду, что наличие слова ALL не отменяет необходимости во временной таблице. MySQL в обязательном порядке помещает в нее результаты, а затем читает их оттуда, даже если без этого и можно было бы обойтись (например, когда результаты можно было вернуть напрямую клиенту).

Подсказки оптимизатору запросов

В СУБД MySQL имеется ряд подсказок оптимизатору запросов, которыми можно воспользоваться, чтобы повлиять на выбор плана выполнения, если тот, что предложен оптимизатором, вас не удовлетворяет. Ниже приведен их перечень с рекомендациями, когда имеет смысл применять данную подсказку. Подсказка включается в запрос, чей план выполнения вы хотите изменить, и действует только для этого запроса. Точный синтаксис подсказок можно найти в документации по MySQL. Некоторые из них зависят от номера версии СУБД.

HIGH_PRIORITY *и* LOW_PRIORITY

Говорят, какой приоритет назначить данной команде относительно других команд, пытающихся обратиться к тем же таблицам.

Подсказка HIGH_PRIORITY означает, что MySQL должен поместить команду SELECT в очередь раньше всех прочих, ожидающих получения блокировок для модификации данных. Иными словами, команда SELECT должна быть выполнена как можно быстрее, а не ожидать своей очереди. Эта подсказка применима и к команде INSERT; в этом случае она просто отменяет действие глобального параметра LOW_PRIORITY, установленного на уровне сервера.

Подсказка LOW_PRIORITY оказывает обратное действие: в этом случае команда будет ждать завершения всех остальных команд, желающих обратиться к тем же таблицам, даже если они были отправлены позже. Можно провести аналогию с излишне вежливым человеком, придерживающим дверь в ресторан, пока есть желающие войти; этак он заморит себе голодом! Данная подсказка применима к командам SELECT, INSERT, UPDATE, REPLACE.

Обе подсказки работают с подсистемами хранения, в которых реализована блокировка на уровне таблиц, но в InnoDB и других подсистемах с более детальным контролем блокировки и конкурентного

доступа необходимости в них возникать не должно. Будьте осторожны, применяя их к таблицам типа MyISAM, поскольку таким образом можно запретить одновременные вставки и существенно понизить производительность.

Подсказки HIGH_PRIORITY и LOW_PRIORITY часто понимают неправильно. Их смысл не в том, чтобы выделить запросу побольше ресурсов, чтобы «сервер уделил ему больше внимания», или поменьше, чтобы «сервер не перетруждался». Они просто влияют на дисциплину обслуживания очереди команд, ожидающих доступа к таблице.

DELAYED

Эта подсказка применяется к командам INSERT и REPLACE. Команда с данной подсказкой возвращает управление немедленно, а подлежащие вставке строки помещаются в буфер и будут реально вставлены все сразу, когда таблица освободится. Чаще всего это бывает полезно для протоколирования и аналогичных приложений, в которых нужно записывать много строк, не заставляя клиента ждать и не выполняя операцию ввода/вывода для каждой команды в отдельности. Однако у данного режима есть много ограничений; так, отложенная вставка реализована не во всех подсистемах хранения, а функция LAST_INSERT_ID( ) в этом случае неприменима.

STRAIGHT_JOIN

Эта подсказка может встречаться сразу после ключевого слова SELECT в команде SELECT или в любой другой команде между двумя соединяемыми таблицами. В первом случае она говорит серверу, что указанные в запросе таблицы нужно соединять в порядке перечисления. Во втором случае она задает порядок соединения таблиц, между которыми находится.

Подсказка STRAIGHT_JOIN полезна, если выбранный MySQL порядок соединения не оптимален или оптимизатор тратит чересчур много времени на выбор порядка. В последнем случае поток слишком много времени проводит в состоянии «Statistics», а, включив эту подсказку, можно будет сократить пространство поиска оптимизатора.

С помощью команды EXPLAIN вы можете посмотреть, какой порядок выбрал оптимизатор, а потом переписать запрос, расположив таблицы именно в этом порядке и добавив подсказку STRAIGHT_JOIN. Указанная идея неплоха, если вы уверены, что фиксированный порядок не приведет к снижению производительности для некоторых условий WHERE. Однако не забывайте пересматривать такие запросы после перехода на новую версию MySQL, поскольку могут появиться другие оптимизации, подавляемые наличием STRAIGHT_JOIN.

SQL_SMALL_RESULT *и* SQL_BIG_RESULT

Эти подсказки применимы только к команде SELECT. Они говорят оптимизатору, когда и как использовать временные таблицы или сор-

тировку при выполнении запросов с GROUP BY или DISTINCT. SQL_SMALL_RESULT означает, что результирующий набор будет невелик, так что его можно поместить в индексированную временную таблицу, чтобы не сортировать для группировки. Напротив, SQL_BIG_RESULT означает, что результат велик, и лучше использовать временные таблицы на диске с последующей сортировкой.

SQL_BUFFER_RESULT

Эта подсказка говорит оптимизатору, что результаты нужно поместить во временную таблицу и как можно скорее освободить табличные блокировки. Это совсем не то, что буферизация на стороне клиента, описанная ранее в разделе «Клиент-серверный протокол MySQL» на стр. 211. Буферизация на стороне сервера может быть полезна, когда вы не используете буферизацию на стороне клиента, поскольку позволяет уменьшить потребление памяти клиентом и вместе с тем быстро освободить блокировки. Компромисс состоит в том, что теперь вместо памяти клиента потребляется память сервера.

SQL_CACHE *и* SQL_NO_CACHE

Эти подсказки говорят серверу о том, что данный запрос является или не является кандидатом на помещение в кэш запросов. О том, как ими пользоваться, рассказано в следующей главе.

SQL_CALC_FOUND_ROWS

Эта подсказка заставляет MySQL вычислить весь результирующий набор, даже если имеется фраза LIMIT, ограничивающая количество возвращаемых строк. Получить общее количество строк позволяет функция FOUND_ROWS( ) (см. однако раздел «Оптимизация SQL_CALC_FOUND_ROWS» на стр. 249, где говорится о том, почему не следует пользоваться этой подсказкой).

FOR UPDATE *и* LOCK IN SHARE MODE

Эти подсказки управляют блокировками для команд SELECT, но только в тех подсистемах хранения, где реализованы блокировки на уровне строк. Они позволяют поставить блокировки на найденные строки, что бывает полезно, когда заранее известно, что эти строки нужно будет обновить, или чтобы избежать эскалации и сразу получить монопольные блокировки.

Данные подсказки не нужны в запросах вида INSERT ... SELECT, поскольку в этом случае MySQL 5.0 по умолчанию ставит блокировки чтения на строки исходной таблицы (это поведение можно отменить, однако мы не рекомендуем так поступать, в главах 8 и 11 объяснено, почему). В версии MySQL 5.1 упомянутое ограничение при определенных условиях может быть снято.

Когда писалась эта книга, лишь подсистема InnoDB позволяла использовать данные подсказки, и пока слишком рано говорить, поддержат ли их в будущем другие подсистемы хранения с блокировка-

ми строк. Применяя эти подсказки в InnoDB, имейте в виду, что они подавляют некоторые оптимизации, например покрывающие индексы. InnoDB не может монопольно заблокировать строки, не обращаясь к индексу по первичному ключу, в котором хранится информация о версиях строк.

`USE INDEX`, `IGNORE INDEX` *и* `FORCE INDEX`

Эти подсказки говорят оптимизатору о том, какие индексы использовать или игнорировать при поиске строк в таблице (например, для выработки решения о порядке соединения). В версии MySQL 5.0 и более ранних они не влияют на выбор сервером индексов для сортировки и группировки. В MySQL 5.1 можно дополнить подсказку ключевыми словами `FOR ORDER BY` или `FOR GROUP BY`.

`FORCE INDEX` — то же самое, что `USE INDEX`, но эта подсказка сообщает оптимизатору о том, что сканирование таблицы обойдется гораздо дороже поиска по индексу, даже если индекс не очень полезен. Вы можете включить данные подсказки, если полагаете, что оптимизатор выбрал неподходящий индекс, или если хотите по какой-то причине воспользоваться конкретным индексом, например для неявного упорядочения без использования `ORDER BY`. В разделе «Оптимизация LIMIT со смещением» (стр. 248) был приведен пример, показывающий, как можно эффективно получить минимальное значение с помощью фразы `LIMIT`.

В версии MySQL 5.0 и более поздних существуют также несколько системных переменных, влияющих на поведение оптимизатора.

`optimizer_search_depth`

Эта переменная говорит оптимизатору, насколько исчерпывающе исследовать частичные планы. Если запрос слишком долго пребывает в состоянии «Statistics», то попробуйте уменьшить это значение.

`optimizer_prune_level`

Эта переменная, которая по умолчанию включена, позволяет оптимизатору пропускать некоторые планы в зависимости от количества исследованных строк.

Обе переменные контролируют режим сокращенного перебора планов выполнения. Такое «срезание углов» бывает полезно для повышения производительности при обработке сложных запросов, но чревато тем, что ради достижения эффективности сервер может пропустить оптимальный план. Поэтому иногда имеет смысл менять их значения.

Переменные, определяемые пользователем

О переменных, определяемых пользователем в MySQL, часто забывают, хотя они могут оказаться мощным инструментом при написании эффективных запросов. Особенно хорошо они работают для запросов,

где можно получить выигрыш от сочетания процедурной и реляционной логики. В чистой реляционной теории все таблицы рассматриваются как неупорядоченные множества, которыми сервер как-то манипулирует целиком. Подход MySQL более прагматичен. Это можно было бы назвать слабой стороной, но, если вы знаете, как ей воспользоваться, то сумеете обратить слабость в силу. И тут могут помочь переменные, определяемые пользователем.

Определяемые пользователем переменные – это временные контейнеры, хранящие некоторые значения. Их существование ограничено временем жизни соединения с сервером. Определяются они простым присвоением с помощью команд `SET` или `SELECT`[1]:

```
mysql> SET @one := 1;
mysql> SET @min_actor := (SELECT MIN(actor_id) FROM sakila.actor);
mysql> SET @last_week := CURRENT_DATE-INTERVAL 1 WEEK;
```

Затем эти переменные можно использовать в различных местах выражения:

```
mysql> SELECT ... WHERE col <= @last_week;
```

Прежде чем начать рассказ о сильных сторонах пользовательских переменных, обратим внимание на некоторые их особенности и недостатки, и покажем, для каких целей их *нельзя* использовать.

- Они подавляют кэширование запроса.
- Их нельзя использовать в тех местах, где требуется литерал или идентификатор, например вместо имени таблицы или столбца либо во фразе `LIMIT`.
- Они связаны с конкретным соединением, поэтому не годятся для передачи информации между соединениями.
- При использовании пула соединений или устойчивых (persistent) соединений они могут привести к интерференции между, казалось бы, изолированными участками кода.
- В версиях, предшествующих MySQL 5.0, они были чувствительны к регистру, поэтому обращайте внимание на совместимость.
- Нельзя явно объявить тип переменной, а точки, в которых MySQL принимает решение о типе неопределенной переменной, в разных версиях различаются. Самое правильное – присвоить начальное значение 0 переменным, предназначенным для хранения целых чисел, 0.0 – переменным, предназначенным для хранения чисел с плавающей точкой, и '' (пустая строка) – переменным, предназначенным для хранения строк. Тип переменной изменяется в момент при-

[1] В некоторых контекстах можно использовать просто знак равенства =, но мы считаем, что во избежание неоднозначности лучше всегда употреблять :=.

сваивания ей значения; в MySQL типизация пользовательских переменных динамическая.

- В некоторых ситуациях оптимизатор может вообще исключить такие переменные, поэтому цель их использования не будет достигнута.
- Порядок и даже момент присваивания переменной значения не всегда детерминирован и может зависеть от выбранного оптимизатором плана выполнения. Ниже вы увидите, что результат может оказаться совершенно обескураживающим.
- Приоритет оператора присваивание := ниже, чем у всех остальных операторов, поэтому следует явно расставлять скобки.
- Наличие неопределенной переменной не приводит к синтаксической ошибке, поэтому легко допустить ошибку, не сознавая этого.

Одна из наиболее важных особенностей переменных состоит в том, что можно одновременно присвоить переменной значение и воспользоваться им. Иными словами, результат операции присваивания – это L-значение. В следующем примере мы вычисляем и выводим «номер строки»:

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum := @rownum + 1 AS rownum
    -> FROM sakila.actor LIMIT 3;
+----------+--------+
| actor_id | rownum |
+----------+--------+
|        1 |      1 |
|        2 |      2 |
|        3 |      3 |
+----------+--------+
```

Сам по себе этот пример не особенно интересен, поскольку просто показывает, как можно продублировать первичный ключ таблицы. Тем не менее, и у него есть применения, одно из них – ранжирование строк. Давайте напишем запрос, который возвращает 10 актеров, сыгравших в наибольшем количестве фильмов, причем значение в столбце rank должно быть одинаково у актеров, сыгравших в одном и том же количестве фильмов. Начнем с запроса, выбирающего актеров и количество фильмов, в которых они сыграли:

```
mysql> SELECT actor_id, COUNT(*) as cnt
    -> FROM sakila.film_actor
    -> GROUP BY actor_id
    -> ORDER BY cnt DESC
    -> LIMIT 10;
+----------+-----+
| actor_id | cnt |
+----------+-----+
|      107 |  42 |
|      102 |  41 |
```

```
|       198 |  40 |
|       181 |  39 |
|        23 |  37 |
|        81 |  36 |
|       106 |  35 |
|        60 |  35 |
|        13 |  35 |
|       158 |  35 |
+-----------+-----+
```

Теперь добавим ранг, который должен быть одинаков у всех актеров, сыгравших в 35 фильмах. Для этого нам понадобятся три переменные: текущий ранг, счетчик фильмов для предыдущего актера и счетчик фильмов для текущего актера. Мы будем изменять ранг при изменении счетчика фильмов. Вот как выглядит первая попытка:

```
mysql> SET @curr_cnt := 0, @prev_cnt := 0, @rank := 0;
mysql> SELECT actor_id,
    ->    @curr_cnt := COUNT(*) AS cnt,
    ->    @rank := IF(@prev_cnt <> @curr_cnt, @rank + 1, @rank) AS rank,
    ->    @prev_cnt := @curr_cnt AS dummy
    -> FROM sakila.film_actor
    -> GROUP BY actor_id
    -> ORDER BY cnt DESC
    -> LIMIT 10;
+-----------+-----+------+-------+
| actor_id | cnt | rank | dummy |
+-----------+-----+------+-------+
|       107 |  42 |    0 |     0 |
|       102 |  41 |    0 |     0 |
...
```

Не получилось: ранг и счетчик все время остаются равными нулю. Почему?

Невозможно дать универсальный ответ. Проблема может быть вызвана просто ошибкой при написании имени переменной (в данном случае это не так) или чем-то более сложным. В нашем примере `EXPLAIN` показывает, что применяются временная таблица и файловая сортировка, поэтому переменные вычисляются не в тот момент, когда мы этого ожидаем.

Такое загадочное поведение довольно часто встречается при работе с пользовательскими переменными в MySQL. Отлаживать подобные ошибки трудно, но усилия окупаются с лихвой. Ранжирование средствами SQL обычно требует квадратичных алгоритмов, тогда как решение с использованием пользовательских переменных оказывается линейным, а это существенное улучшение.

В данном случае проблему можно решить, добавив в запрос еще один уровень временных таблиц за счет подзапроса во фразе `FROM`:

```
mysql> SET @curr_cnt := 0, @prev_cnt := 0, @rank := 0;
```

```
    -> SELECT actor_id,
    ->    @curr_cnt := cnt AS cnt,
    ->    @rank := IF(@prev_cnt <> @curr_cnt, @rank + 1, @rank) AS rank,
    ->    @prev_cnt := @curr_cnt AS dummy
    -> FROM (
    ->    SELECT actor_id, COUNT(*) AS cnt
    ->    FROM sakila.film_actor
    ->    GROUP BY actor_id
    ->    ORDER BY cnt DESC
    ->    LIMIT 10
    -> ) as der;
+----------+-----+------+-------+
| actor_id | cnt | rank | dummy |
+----------+-----+------+-------+
|      107 |  42 |    1 |    42 |
|      102 |  41 |    2 |    41 |
|      198 |  40 |    3 |    40 |
|      181 |  39 |    4 |    39 |
|       23 |  37 |    5 |    37 |
|       81 |  36 |    6 |    36 |
|      106 |  35 |    7 |    35 |
|       60 |  35 |    7 |    35 |
|       13 |  35 |    7 |    35 |
|      158 |  35 |    7 |    35 |
+----------+-----+------+-------+
```

Бóльшая часть проблем при работе с пользовательскими переменными проистекает из-за того, что присваивание и чтение значений происходят на разных стадиях обработки запроса. Например, невозможно предсказать, что произойдет, если присвоить значение во фразе `SELECT`, а прочитать во фразе `WHERE`. Так, может показаться, что следующий запрос вернет одну строку, однако в реальности дело обстоит по-другому:

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum := @rownum + 1 AS cnt
    -> FROM sakila.actor
    -> WHERE @rownum <= 1;
+----------+------+
| actor_id |  cnt |
+----------+------+
|        1 |    1 |
|        2 |    2 |
+----------+------+
```

Так происходит потому, что фразы `WHERE` и `SELECT` обрабатываются на разных стадиях процесса выполнения запроса. Это видно даже более наглядно, если добавить еще одну стадию, включив фразу `ORDER BY`:

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum := @rownum + 1 AS cnt
    -> FROM sakila.actor
```

```
    -> WHERE @rownum <= 1
    -> ORDER BY first_name;
```

Данный запрос возвращает все строки из таблицы, поскольку ORDER BY добавила файловую сортировку (filesort), а WHERE вычисляется до сортировки. Решение состоит в том, чтобы присваивать значения и читать их на *одной и той же* стадии выполнения запроса:

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum AS rownum
    -> FROM sakila.actor
    -> WHERE (@rownum := @rownum + 1) <= 1;
+----------+--------+
| actor_id | rownum |
+----------+--------+
|        1 |      1 |
+----------+--------+
```

Вопрос на засыпку: что произойдет, если включить в этот запрос фразу ORDER BY? Попробуйте сами. Если полученный результат отличается от ожидаемого, то объясните, в чем причина. А как насчет следующего запроса, где значение переменной изменяется во фразе ORDER BY, а считывается во фразе WHERE?

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, first_name, @rownum AS rownum
    -> FROM sakila.actor
    -> WHERE @rownum <= 1
    -> ORDER BY first_name, LEAST(0, @rownum := @rownum + 1);
```

Ответы на большинство вопросов, связанных с неожиданным поведением пользовательских переменных, можно получить, выполнив команду EXPLAIN и поискав фразы «Using where», «Using temporary» или «Using filesort» в столбце Extra.

В последнем примере применен еще один полезный трюк: мы поместили присваивание в функцию LEAST( ), так что ее значение отбрасывается и не искажает результатов ORDER BY (мы вызываем функцию LEAST( ) так, что она всегда возвращает 0). Этот прием очень полезен, когда присваивание переменной выполняется исключительно ради побочного эффекта; он позволяет скрыть возвращаемое значение и избежать появления лишних столбцов (таких, как dummy в примере выше). Для этой цели подходят также функции GREATEST(), LENGTH(), ISNULL(), NULLIF(), COALESCE() и IF(), поодиночке и в сочетании друг с другом, в силу особенностей их поведения. Например, COALESCE() прекращает вычислять свои аргументы, встретив первый, имеющий значение, отличное от NULL.

Присваивание переменной может встречаться в любой команде, а не только в SELECT. Более того, это один из наиболее удачных способов применения пользовательских переменных. Например, накладные расхо-

ды, скажем, вычисление ранга с помощью подзапроса, можно сделать более низкими, переписав запросы в виде команды UPDATE.

Впрочем, добиться требуемого поведения не всегда легко. Иногда оптимизатор считает переменные константами этапа компиляции и отказывается выполнять присваивания. Обычно помогает помещение присваиваний внутрь функции типа LEAST(). Дадим еще один совет: проверяйте, присвоено ли переменной значение, прежде чем выполнять содержащую ее команду. Иногда у переменной должно быть значение, иногда – нет. Немного поэкспериментировав, вы научитесь делать с помощью пользовательских переменных очень интересные вещи. Вот несколько идей:

- Вычисление промежуточных итогов и средних
- Эмуляция функций FIRST() и LAST() с помощью запросов с группировкой
- Математические операции над очень большими числами
- Вычисление MD5-свертки для всей таблицы
- Восстановление выборочного значения, которое «оборачивается», если превышает некоторую границу
- Эмуляция курсоров чтения/записи

Осторожнее при переходе на новые версии MySQL

Как мы уже говорили, попытки перехитрить оптимизатор MySQL обычно ни к чему хорошему не приводят. Вы лишь создаете себе лишнюю работу и увеличиваете стоимость сопровождения, не получая ничего взамен. В особенности это относится к переходу на новую версию MySQL, поскольку включенные в запрос подсказки могут помешать оптимизатору применить новую, более эффективную стратегию.

В частности, способы использования индексов оптимизатором можно уподобить движущейся мишени. В новой версии они могут измениться, и вам придется подстраивать выработанные приемы индексирования под иные реалии. Например, мы отмечали, что в версии MySQL 4.0 и более ранних разрешено было использовать только один индекс на каждую упомянутую в запросе таблицу. А начиная с версии MySQL 5.0 были реализованы стратегии слияния индексов.

Помимо существенных изменений, которые время от времени вносятся в оптимизатор запросов, в каждой второстепенной версии обычно реализовано множество мелких усовершенствований. Как правило, они касаются таких деталей, как условия, при которых некий индекс исключается из рассмотрения, и позволяют лучше оптимизировать некоторые частные случаи.

Хотя в теории все это выглядит прекрасно, на практике после перехода на новую версию может оказаться, что некоторые запросы работают *хуже*, чем раньше. Если вы в течение длительного времени работа-

ли с определенной версией, то, вероятно, вольно или невольно подстроили под нее свои запросы. В новой редакции выполненная вами ручная оптимизация может быть неприменима, а то и станет причиной снижения производительности.

Если вы всерьез намерены добиться максимальной эффективности, то должны подготовить комплекс тестов, описывающих нагрузку в ваших конкретных условиях эксплуатации, прогонять эти тесты на сервере разработки для каждой новой версии СУБД и лишь потом устанавливать новую версию на промышленные сервера. Кроме того, перед тем как переходить на обновленную версию, ознакомьтесь с замечаниями к ней и со списком известных проблем. В руководство по MySQL включен удобно организованный перечь известных серьезных ошибок.

Как правило, с каждой новой версией MySQL общая производительность сервера повышается; мы вовсе не хотим, чтобы у вас сложилось противоположное впечатление. Однако осторожность все равно не повредит.

5

Дополнительные средства MySQL

В версиях MySQL 5.0 и 5.1 появилось немало средств, знакомых пользователям других СУБД. Например, хранимые процедуры, представления и триггеры. Добавление в MySQL этих возможностей привлекло множество новых пользователей. Однако влияние на производительность подобных нововведений оставалось неясным до начала широкого применения.

В настоящей главе мы рассмотрим эти новшества и другие темы, в том числе некоторые возможности, существовавшие еще в версии MySQL 4.1 и даже раньше. Мы сосредоточимся главным образом на производительности, но заодно покажем, как получить максимальную выгоду от использования этих продвинутых средств.

Кэш запросов MySQL

Многие СУБД умеют кэшировать планы выполнения, что позволяет серверу пропустить стадии разбора и оптимизации запросов, уже встречавшихся ранее. MySQL при некоторых условиях тоже может это делать, но, кроме того, у нее имеется кэш особого вида (так называемый *кэш запросов*), в котором хранятся *полные результирующие наборы*, сгенерированные командами `SELECT`. Этот раздел посвящен именно такому кэшу.

В кэше запросов MySQL хранится в точности тот результат, который запрос вернул клиенту. При попадании в кэш запросов сервер сразу же возвращает сохраненные итоги, пропуская стадии разбора, оптимизации и выполнения.

Кэш запросов отслеживает, какие таблицы были использованы в запросе, и, если хотя бы одна из них изменилась, данные в кэше становятся недействительными. Такая грубая политика определения недействительности (инвалидации) может показаться неэффективной, поскольку внесенные изменения, возможно, и не отражаются на сохра-

ненных результатах. Однако накладные расходы, сопряженные с таким упрощенным подходом, невелики, что очень важно для сильно нагруженной системы.

Кэш запросов спроектирован так, что остается абсолютно прозрачным для приложений. Программе не нужно знать, откуда она получила данные: из кэша или в результате реального выполнения запроса. В любом случае результат будут одинаковым. Иными словами, кэш запросов не изменяет семантику – с точки зрения приложения, поведение сервера не зависит от того, активирован кэш или нет[1].

Как MySQL проверяет попадание в кэш

Для проверки попадания в кэш MySQL применяет простую и очень быструю методику: по сути кэш – это справочная таблица (lookup table). Ключом этой таблицы является хеш, рассчитанный по тексту запроса, текущей базе данных, номеру версии клиентского протокола и еще нескольким параметрам, от которых могут зависеть результаты обработки запроса.

При проверке попадания в кэш MySQL не разбирает, не «нормализует» и не параметризует команду; текст команды и прочие данные используются точно в том виде, в каком они получены от клиента. Любое различие, будь то регистр символов, лишние пробелы или комментарии, приведет к тому, что новый запрос не совпадет с кэшированной версией. Об этом следует помнить при написании запросов. Следовать единому стилю или способу форматирования – вообще хорошая привычка, но в данном случае она может еще и ускорить работу системы.

Следует также иметь в виду, что результат не попадет в кэш запросов, если сгенерирован недетерминированным запросом. Иначе говоря, любой запрос, содержащий недетерминированную функцию, например `NOW()` или `CURRENT_DATE()`, не кэшируется. Аналогично, такие функции, как `CURRENT_USER()` и `CONNECTION_ID()`, дают разные результаты в зависимости от того, каким пользователем вызваны, поэтому также препятствуют записи запроса в кэш. Кроме того, кэш не работает для запросов, в которых есть ссылки на определенные пользователем функции, хранимые процедуры, пользовательские переменные, временные таблицы, таблицы в базе данных `mysql` и таблицы, для которых заданы привилегии на уровне столбцов (полный перечень всех причин, по которым запрос не кэшируется, см. в руководстве по MySQL).

Нам часто доводилось слышать, будто «MySQL не проверяет кэш, если запрос содержит недетерминированную функцию». Это неправильно.

---

[1] На самом деле, кэш запросов все-таки изменяет семантику в одном, весьма тонком отношении: по умолчанию запрос может быть обслужен из кэша, если одна из участвующих в нем таблиц заблокирована предложением `LOCK TABLES`. Этот режим можно отменить, установив переменную `query_cache_wlock_invalidate`.

MySQL не может знать, является функция детерминированной или нет, пока не разберет запрос, а поиск в кэше производится *до* разбора. Сервер лишь проверяет, что запрос начинается с букв SEL (без учета регистра), этим и ограничивается.

Однако утверждение «сервер не найдет в кэше результатов, если запрос содержит функцию типа NOW()», правильно, так как даже если бы сервер и выполнял такой запрос раньше, он не поместил бы его в кэш. MySQL помечает запрос как некэшируемый, как только обнаруживает конструкцию, препятствующую кэшированию, и результаты такого запроса не сохраняются.

Есть полезный прием, позволяющий все же закэшировать запросы, ссылающиеся на текущую дату; для этого нужно включить дату в виде литерала, вместо использования функции. Например:

```
... DATE_SUB(CURRENT_DATE, INTERVAL 1 DAY) -- не кэшируется!
... DATE_SUB('2007-07-14', INTERVAL 1 DAY) -- кэшируется
```

Поскольку кэш запросов работает с полным текстом команды SELECT, идентичные запросы, сделанные внутри подзапроса или представления, не могут воспользоваться кэшем, равно как и запросы внутри хранимых процедур. В версиях до MySQL 5.1 подготовленные запросы также не могли быть закэшированы.

Кэш запросов MySQL может повысить производительность, но при работе с ним нужно помнить о некоторых тонкостях. Во-первых, если кэш включен, добавляются накладные расходы как при чтении, так и при записи:

- Перед началом обработки запроса на чтение нужно проверить, есть ли он в кэше
- Если запрос допускает кэширование, но еще не помещен в кэш, то нужно потратить некоторое время на запись в кэш сгенерированных результатов
- Наконец, при обработке любого запроса на запись необходимо сделать недействительными все записи в кэше, в которых встречается измененная таблица

Эти издержки сравнительно невелики, поэтому в целом кэш все же может дать выигрыш. Однако, как мы увидим позже, дополнительные накладные расходы могут суммироваться.

Пользователей InnoDB подстерегает еще одна проблема: полезность кэша ограничена транзакциями. Если какая-то команда внутри транзакции модифицирует таблицу, то сервер делает недействительными все кэшированные запросы, ссылающиеся на эту таблицу, пусть даже реализованная в InnoDB система многоверсионного контроля способна скрыть произведенные в транзакции изменения от других команд. Таблица также считается глобально некэшируемой до тех пор, пока транзакция не будет зафиксирована, поэтому все последующие запросы

к той же таблице – неважно, внутри или вне транзакции, – также не могут кэшироваться, пока транзакция не завершится. Поэтому длинные транзакции увеличивают количество «непопаданий» в кэш.

Инвалидация может стать серьезной проблемой, когда кэш запросов велик. Если в кэше много запросов, то поиск тех результатов, которые нужно объявить недействительными, может занять длительное время, в течение которого вся система будет простаивать. Так происходит потому, что существует единственная глобальная блокировка, разрешающая доступ к кэшу только одному запросу в каждый момент времени. А этот доступ производится как при проверке попадания, так и при выяснении того, какие запросы нужно сделать недействительными.

Как кэш использует память

MySQL хранит кэш запросов целиком в памяти, поэтому нужно понимать, как эта память используется, чтобы правильно настроить механизм кэширования. В кэше содержатся не только результаты обработки запроса. В некоторых отношениях он очень похож на файловую систему: хранит структуры, позволяющие узнать, сколько памяти в пуле свободно, можно ли сопоставить таблицы и тексты команд с результатами запросов.

Если не принимать во внимание некоторые служебные структуры, под которые отведено примерно 40 Кбайт, то весь пул памяти, выделенной под кэш, состоит из *блоков* переменной длины. Каждый блок знает о своем типе, размере, о том, сколько данных он содержит, а также включает указатели на следующий и предыдущий логический и физический блок. Есть несколько типов блоков: для хранения кэшированных результатов, списков таблиц, упоминаемых в запросе, текста запроса и т. д. Однако блоки разных типов трактуются в значительной мере одинаково, поэтому различать их для настройки кэша запросов нет необходимости.

На этапе запуска сервер инициализирует память, выделенную для кэша запросов. Первоначально пул памяти состоит из одного свободного блока. Его размер совпадает с размером сконфигурированной для нужд кэша памяти за вычетом служебных структур.

Когда сервер кэширует результат запроса, он выделяет блок для хранения этого результата. Минимальный размер блока составляет `query_cache_min_res_unit` байтов, хотя может и отличаться, если сервер знает, что для результирующего набора требуется больше памяти. К сожалению, сервер не в состоянии создать блок в точности подходящего размера, так как первоначальное выделение производится еще до того, как результирующий набор полностью сгенерирован. Сервер не строит весь набор в памяти с тем, чтобы позже отправить его целиком, – гораздо эффективнее отправлять клиенту строки по мере их порождения. Следовательно, начиная построение результирующего набора, сервер еще не знает, насколько большим он получится.

Выделение блока – относительно медленный процесс, так как сервер должен просмотреть список свободных блоков и найти достаточно большой из них. Поэтому сервер пытается минимизировать количество операций выделения. Когда возникает необходимость кэшировать результирующий набор, свервер выделяет блок размера, не меньшего минимально допустимого, и начинает помещать в него результаты. Если блок заполняется раньше, чем данные закончились, то сервер выделяет новый блок – снова минимального или большего размера – и продолжает помещать в него данные. Когда результат полностью сгенерирован, сервер смотрит, осталось ли в последнем блоке место, и, если да, усекает блок и объединяет оставшуюся область с соседним свободным блоком. Весь этот процесс показан на рис. 5.1[1].

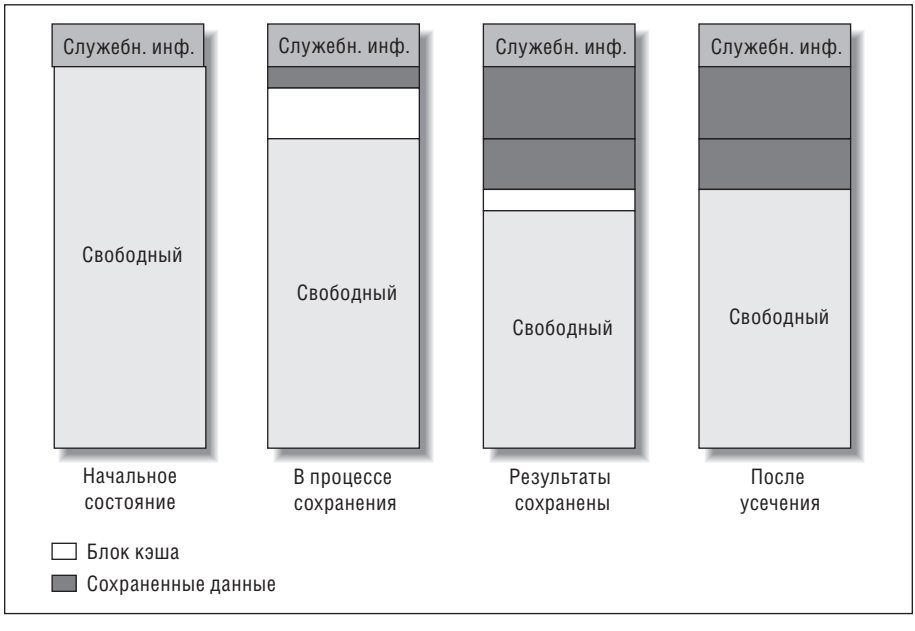


Рис. 5.1. *Как выделяются блоки для хранения результатов запроса*

Говоря, что сервер «выделяет блок», мы не имеем в виду, что он запрашивает память у операционной системы, вызывая функцию `malloc()` или ей подобную. Это делается только один раз, при создании кэша запросов. Затем сервер только просматривает список блоков и либо находит место, где собирается разместить новый блок, либо при необходимости удаляет самый старый кэшированный запрос, чтобы освободить

[1] В целях иллюстрации диаграммы в этом разделе упрощены. На самом деле, алгоритм выделения блоков кэша сложнее, чем здесь показано. Если вас интересуют детали, почитайте комментарии в начале исходного файла *sql/sql_cache.cc*; там все очень понятно описано.

место. Иными словами, сервер MySQL самостоятельно управляет своей памятью, не полагаясь на операционную систему.

До сих пор все выглядело довольно просто. Однако картина может оказаться сложнее, чем представлено на рис. 5.1. Предположим, что средний результирующий набор очень мал, и сервер одновременно отправляет результаты по двум клиентским соединениям. После усечения может остаться блок, который меньше `query_cache_min_res_unit`, и, следовательно, не может в будущем использоваться для хранения результирующих наборов. Последовательность операций выделения блоков может привести к ситуации, показанной на рис. 5.2.

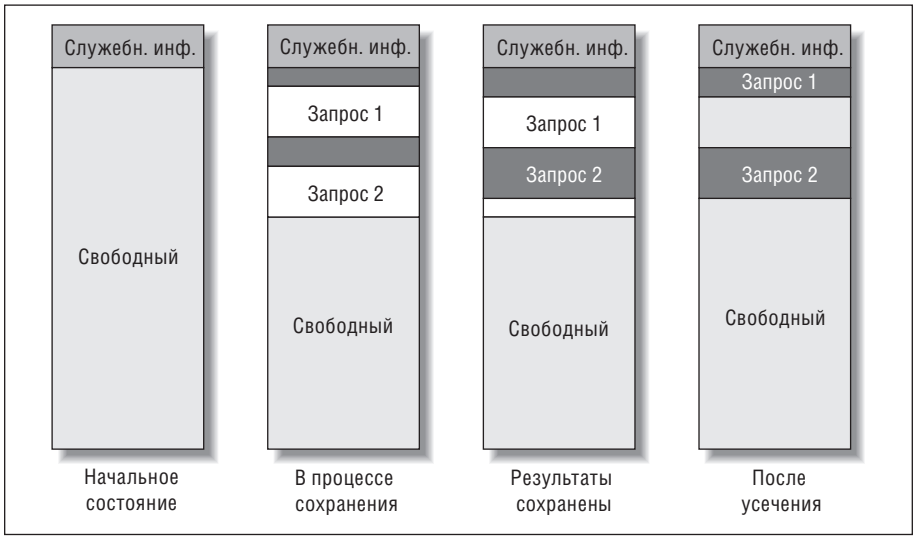


Рис. 5.2. *Фрагментация, вызванная сохранением результатов в кэше запросов*

После усечения первого результата по размеру остался так называемый зазор – блок, слишком маленький для того, чтобы сохранить в нем результат другого запроса. Образование таких зазоров называется *фрагментацией*; это классическая проблема в задачах выделения памяти и в файловых системах. У фрагментации много причин, в том числе объявление записей в кэше недействительными (инвалдиция), при этом тоже могут появляться слишком мелкие блоки.

Когда полезен кэш запросов

Нельзя сказать, что кэширование запросов автоматически оказывается более эффективным, чем его отсутствие. На кэширование уходит время, поэтому выигрыш от кэша можно получить, только если экономия перевешивает издержки. А это зависит от загруженности сервера.

Теоретически определить, полезен кэш или нет, можно, сравнив объем работы, который сервер должен проделать с включенным и отключенным кэшем. Если кэш отключен, то сервер выполняет все запросы на чтение и на запись, причем в первом случае – еще генерирует и возвращает результаты. Если кэш включен, то при обработке любого запроса на чтение необходимо сначала проверить кэш, а затем либо вернуть сохраненный результат, либо (если его нет в кэше) выполнить запрос, сгенерировать результирующий набор и отослать его клиенту. При обработке любого запроса на запись его следует сначала выполнить, а потом проверить, нужно ли объявить недействительными какие-то записи в кэше.

Хотя выглядит все это достаточно просто, точно предсказать выигрыш от использования кэша затруднительно. Необходимо принимать во внимание еще и внешние факторы. Например, кэш запросов может сократить время, необходимое для формирования результата, но не время, требующееся для отправки его клиенту, а именно оно может оказаться доминирующим фактором.

Выигрывают от наличия кэша те запросы, которые долго выполняются, но занимают в кэше немного места, так что их хранение, возврат клиенту и инвалидация обходятся дешево. В эту категорию попадают, в частности, агрегирующие запросы, например, с функцией `COUNT()` для больших таблиц. Но есть и много других типов запросов, которые имеет смысл кэшировать.

Один из самых простых способов понять, дает ли кэш выигрыш, – посмотреть на коэффициент попаданий в кэш. Он показывает, сколько запросов было обслужено из кэша, а не выполнено сервером с нуля. Когда сервер получает команду `SELECT`, он увеличивает одну из двух переменных состояния: `Qcache_hits` или `Com_select`, в зависимости от того, был запрос кэширован или нет. Поэтому коэффициент попаданий в кэш вычисляется по формуле `Qcache_hits / (Qcache_hits+Com_select)`.

Какой коэффициент попаданий считать хорошим? Так сразу и не скажешь. Даже 30% попаданий может быть очень неплохим результатом, поскольку экономия от невыполненных запросов, как правило (в расчете на один запрос), значительно превышает накладные расходы на объявление записей недействительными и на сохранение результатов в кэше. Важно также знать, какие именно запросы кэшируются. Если попадания в кэш относятся к наиболее дорогим запросам, то даже низкий коэффициент может означать существенную экономию на работе сервера.

Любой запрос `SELECT`, не обслуженный из кэша, называется *непопаданием в кэш*. Не попасть в кэш можно по следующим причинам.

- Запрос не допускает кэширования, поскольку либо содержит недетерминированную конструкцию (например, функцию `CURRENT_DATE`), либо потому что результирующий набор слишком велик для сохра-

нения в кэше. В обоих случаях переменная `Qcache_not_cached` увеличивается на единицу.

- Сервер еще не видел такой запрос, поэтому не мог закэшировать его результат.
- Результат запроса был ранее закэширован, но сервер удалил его. Такое может произойти из-за нехватки памяти, из-за того, что кто-то велел серверу удалить запрос из кэша, или потому что запись была объявлена недействительной.

Если количество непопаданий в кэш велико, а количество некэшируемых запросов очень мало, то верно одно из следующих утверждений.

- Кэш еще не «прогрелся», то есть сервер не успел заполнить кэш результатами запросов.
- Сервер постоянно видит запросы, которые раньше не встречались. Если количество повторяющихся запросов невелико, то такое может случиться даже, когда кэш «прогрелся».
- Слишком часто записи в кэше объявляются недействительными.

Запись может стать недействительной из-за фрагментации, нехватки ресурсов или изменения данных. Если вы отвели под кэш достаточно памяти и правильно настроили параметр `query_cache_min_res_unit`, то инвалидация по большей части обусловлена изменением хранимых значений. Узнать, сколько запросов модифицировали данные, позволяют переменные состояния `Com_*` (`Com_update`, `Com_delete` и т. д.), а сколько запросов было объявлено недействительными из-за нехватки памяти, – переменная `Qcache_lowmem_prunes`.

Имеет смысл рассматривать издержки на инвалидацию отдельно от коэффициента попаданий. Предположим, например, что из одной таблицы производится только чтение, при этом коэффициент попадания равен 100%, а в другой – только обновление. Если вычислить коэффициент попадания по переменным состояния, то он составит 100%. Но даже в этом случае использование кэша может оказаться неэффективным, поскольку замедляет выполнение запросов на обновление. При обработке любого запроса на обновление приходится проверять, не нужно ли объявить недействительными какие-то кэшированные данные, но поскольку ответ неизменно будет «нет», то вся эта работа проделана впустую. Такого рода проблему можно и не заметить, если не проанализировать совместно количество некэшируемых запросов и коэффициент попадания.

Сервер, для которого смесь операций записи и кэшируемого чтения из одних и тех же таблиц сбалансирована, тоже не всегда выигрывает от наличия кэша запросов. Любая операция записи делает запросы в кэше недействительными, тогда как любая операция чтения вставляет в кэш новый запрос. А выигрыш можно получить только, если эти запросы потом обслуживаются из кэша.

Если закэшированный результат становится недействительным до того, как сервер снова получает ту же самую команду SELECT, то вся работа по сохранению была пустой тратой времени и памяти. Чтобы понять, имеет ли место такая ситуация, сравните значения переменных Com_select и Qcache_inserts. Если почти все команды SELECT не попадают в кэш (при этом увеличивается значение Com_select), после чего их результирующие наборы кэшируются, то Qcache_inserts будет мало отличаться от Com_select. А хотелось бы, чтобы значение Qcache_inserts было существенно меньше Com_select, по крайней мере, после «прогрева» кэша.

Для каждого приложения существует конечный *потенциальный размер кэша*, даже при отсутствии операций записи. Потенциальный размер – это объем памяти, необходимый для сохранения всех кэшируемых запросов, которые может выполнить приложение. Теоретически для большинства случаев этот размер очень велик. Но на практике у многих приложений он гораздо меньше, чем можно было бы ожидать, из-за того, что большинство записей в кэше становятся недействительными. Даже если выделить для кэша запросов очень много памяти, он никогда не заполнится больше, чем на потенциальный размер кэша.

Необходимо следить за тем, какая доля выделенной для кэша памяти реально используется. Если используется не вся память, уменьшите размер кэша, если из-за нехватки памяти часто происходит вытеснение, – увеличьте его. Впрочем, избыточный размер кэша – это не так страшно; из-за того, что выделено чуть больше или чуть меньше памяти, чем реально требуется, производительность сильно не пострадает. Проблема возникает, когда не используется очень много памяти или запросы вытесняются из кэша настолько часто, что все кэширование приносит только убытки.

Необходимо также соразмерять размер кэша запросов с другими серверными кэшами, например пулом буферов InnoDB или кэшем ключей MyISAM. Невозможно предложить коэффициент на все случаи жизни или какую-нибудь простую формулу, поскольку подходящий баланс зависит от конкретного приложения.

Как настраивать и обслуживать кэш запросов

Если вы понимаете, как работает кэш запросов, то настроить его достаточно легко. В нем всего несколько «движущихся деталей».

query_cache_type

Включен ли режим кэширования запросов. Допустимые значения: OFF, ON, DEMAND. Последнее означает, что кэшируются только запросы с модификатором SQL_CACHE. Эта переменная может быть установлена глобально или на уровне сеанса. Подробнее о глобальных и сеансовых переменных см. в главе 6.

`query_cache_size`

Общий объем памяти, отведенной под кэш запросов, в байтах. Значение должно быть кратно 1024, поэтому MySQL, возможно, слегка изменит заданное вами число.

`query_cache_min_res_unit`

Минимальный размер выделяемого блока. О том, как этот параметр используется, было рассказано в разделе «Как кэш использует память» на стр. 264; дополнительная информация будет представлена в следующем разделе.

`query_cache_limit`

Размер максимального результирующего набора, который разрешено кэшировать. Если при выполнении запроса результирующий набор оказывается больше, он не кэшируется. Напомним, что сервер кэширует результаты по мере их генерации, поэтому заранее не известно, поместится ли результат в кэш. Если размер результата превышает заданный порог, то MySQL увеличивает переменную состояния `Qcache_not_cached` и отбрасывает закэшированные к этому моменту строки. Если это происходит часто, то можно включить указание `SQL_NO_CACHE` в запросы, которые могут приводить к такой ситуации.

`query_cache_wlock_invalidate`

Следует ли обслуживать из кэша результаты, которые относятся к таблицам, заблокированным другими соединениями. По умолчанию этот параметр равен `OFF`, что изменяет семантику запроса, поскольку позволяет серверу читать кэшированные данные из заблокированной таблицы, хотя обычно сделать это невозможно. Если установить параметр равным `ON`, то чтение данных будет запрещено, но может увеличиться время ожидания блокировки. Для большинства приложений это несущественно, поэтому значение, принимаемое по умолчанию, как правило, приемлемо.

В принципе, настройка кэша – довольно простое дело, сложнее разобраться в эффекте внесенных изменений. В следующих разделах мы покажем, как анализировать поведение кэша запросов и принимать правильные решения.

Уменьшение фрагментации

Полностью избежать фрагментации невозможно, но тщательный выбор параметра `query_cache_min_res_unit` может помочь не расходовать понапрасну память, отведенную под кэш запросов. Хитрость в том, чтобы найти оптимальное соотношение размера каждого нового блока с количеством операций выделения памяти, которые сервер должен выполнить во время сохранения результатов. Если задать слишком маленькое значение, то сервер будет терять меньше памяти, но блоки ему придется выделять чаще, а, значит, объем работы увеличится. Если же размер слишком велик, то возрастет фрагментация. Придется искать

компромисс между напрасным расходованием памяти и затратами процессорного времени на выделение блоков.

Оптимальное значение зависит от размера результирующего набора типичного запроса. Среднюю величину кэшированных запросов можно найти, разделив объем используемой памяти (приблизительно query_cache_size – Qcache_free_memory) на значение переменной состояния Qcache_queries_in_cache. Если имеется смесь больших и маленьких результирующих наборов, то, возможно, не удастся подобрать размер так, чтобы одновременно избежать фрагментации и свести к минимуму количество выделений. Однако у вас могут быть основания полагать, что кэширование больших результатов не дает заметного выигрыша (часто так оно и есть). Подавить кэширование больших результирующих наборов можно, уменьшив значение переменной query_cache_limit. Иногда это позволяет достичь лучшего баланса между фрагментацией и накладными расходами на сохранение результатов в кэше.

Обнаружить, что кэш фрагментирован, можно, проанализировав переменную состояния Qcache_free_blocks, которая показывает количество блоков типа FREE (свободный). В конечной конфигурации на рис. 5.2 есть два свободных блока. Наихудший случай фрагментации имеет место, когда между каждой парой блоков, в которых хранятся данные, находится чуть меньший минимального размера блок, то есть каждый второй блок свободен. Поэтому если Qcache_free_blocks приблизительно равно Qcache_total_blocks / 2, то кэш сильно фрагментирован. Если переменная состояния Qcache_lowmem_prunes увеличивается и имеется много свободных блоков, значит, из-за фрагментации запросы преждевременно вытесняются из кэша.

Дефрагментировать кэш запросов можно с помощью команды FLUSH QUERY CACHE. Она уплотняет кэш, перемещая все блоки «вверх» и избавляясь от свободного пространства между ними, так что в итоге остается единственный свободный блок «внизу». На время выполнения команды доступ к кэшу запросов блокирован, то есть, по существу, приостановлена работа всего сервера, но обычно, если кэш не слишком велик, дефрагментация быстро завершается. Несмотря на название, команда FLUSH QUERY CACHE не удаляет запросы из кэша. Для этой цели предназначена команда RESET QUERY CACHE.

Улучшение коэффициента загрузки кэша

Если кэш не фрагментирован, но коэффициент попадания все равно низкий, то, возможно, вы отвели под него слишком мало памяти. Когда сервер не может найти свободный блок, достаточно большой для сохранения нового результата, он должен «вытеснить» (prune) из кэша какие-то запросы.

При вытеснении запроса сервер увеличивает переменную состояния Qcache_lowmem_prunes. Если ее значение быстро увеличивается, то возможны две причины:

- Если свободных блоков много, то пенять, скорее всего, следует на фрагментацию (см. предыдущий раздел).
- Если свободных блоков мало, то, вероятно, нагрузка на сервер такова, что при наличии такой возможности он мог бы использовать кэш большего размера. Узнать, сколько в кэше неиспользуемой памяти, поможет переменная `Qcache_free_memory`.

Если свободных блоков много, фрагментация низкая, вытеснений из-за нехватки памяти мало, а коэффициент попадания *все равно* невысок, то, вероятно, кэш запросов вообще мало помогает при данной загрузке сервера. Что-то мешает его использовать. Возможно, причина в большом количестве обновлений, а, возможно, ваши запросы не допускают кэширования.

Если вы измерили коэффициент попадания в кэш, но все же не уверены, выигрывает ли сервер от наличия кэша, то можно отключить его, последить за производительностью, затем снова включить и посмотреть, как производительность изменится. Чтобы отключить кэш запросов, установите параметр `query_cache_size` в 0 (изменение параметра `query_cache_size` глобально не отражается на уже открытых соединениях и не приводит к возврату памяти серверу). Можно также заняться эталонным тестированием, но иногда бывает сложно получить реалистичную комбинацию кэшированных запросов, некэшированных запросов и обновлений.

На рис. 5.3 изображена примерная блок-схема простой процедуры анализа и настройки кэша запросов.

InnoDB и кэш запросов

InnoDB возаимодействует с кэшем более сложным образом, чем другие подсистемы хранения, поскольку она реализует MVCC. В MySQL 4.0 кэш запросов внутри транзакций вообще отключен, но в версии 4.1 и более поздних InnoDB сообщает серверу, для каждой таблицы в отдельности, может ли транзакция обращаться к кэшу запросов. Она управляет доступом к кэшу как для операций чтения (выборки результатов из кэша), так и для операций записи (сохранения результатов в кэше).

Возможность доступа определяется идентификатором транзакции и наличием блокировок на таблицу. В словаре данных InnoDB, который хранится в памяти, с каждой таблицей ассоциирован счетчик идентификаторов транзакций. Тем транзакциям, идентификатор которых меньше этого счетчика, запрещено выполнять операции чтения и записи в кэш для запросов, в которых участвует данная таблица. Наличие любых блокировок на таблицу также препятствует кэшированию обращающихся к ней запросов. Например, если в транзакции выполняется запрос `SELECT FOR UPDATE` для некоторой таблицы, то никакая другая транзакция не сможет ни читать из кэша, ни писать в кэш запросы к этой же таблице до тех пор, пока все блокировки не будут сняты.

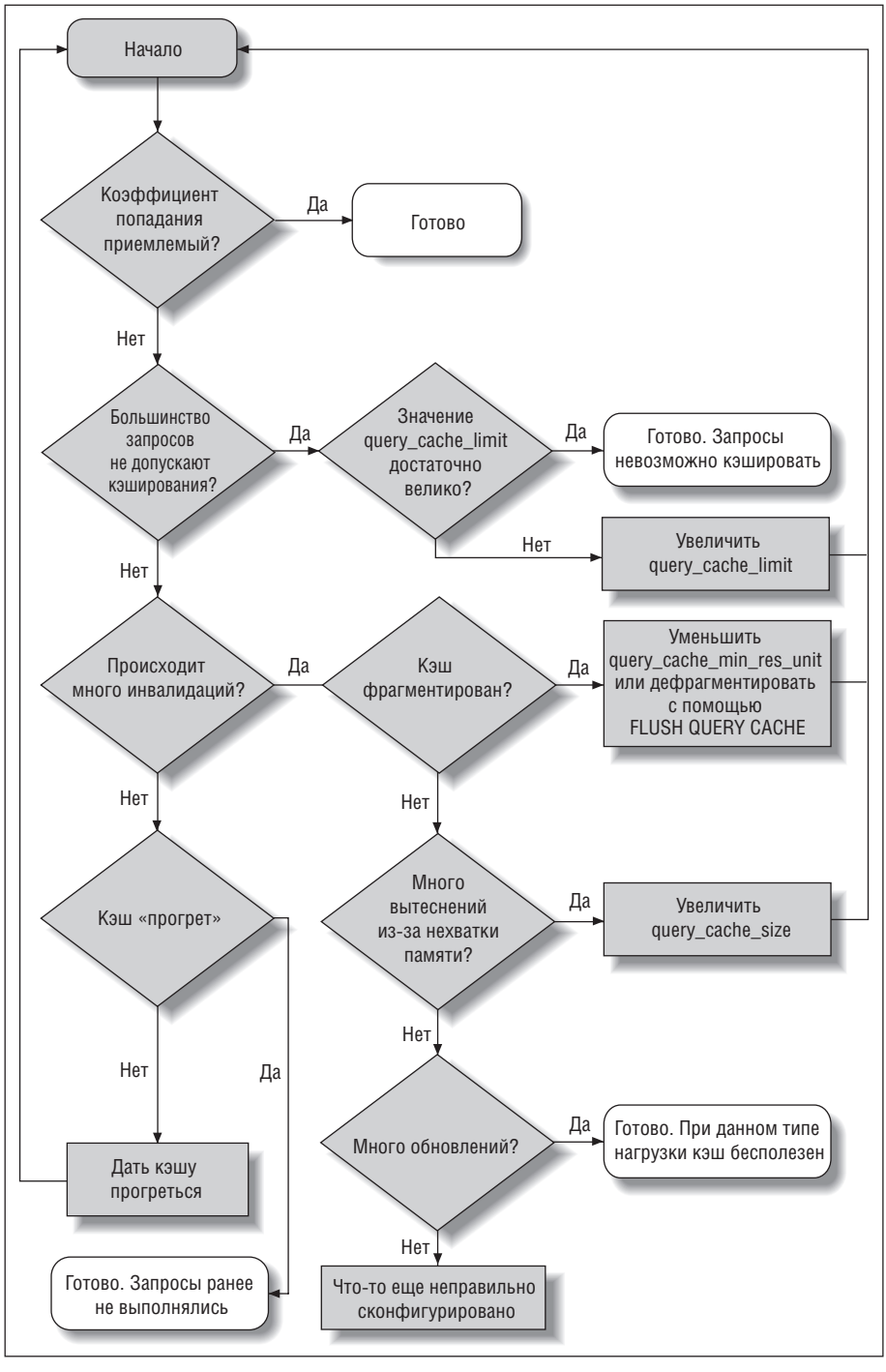


Рис. 5.3. *Порядок анализа и настройки кэша запросов*

После фиксации транзакции InnoDB обновляет счетчики для таблиц, на которые во время транзакции были поставлены блокировки. Наличие блокировки можно рассматривать как грубую эвристику для определения того, была ли таблица модифицирована внутри транзакции. Вполне возможно, что транзакция поставила блокировки на строки таблицы, но не обновила их. Однако не может быть так, чтобы данные в таблице были изменены без установки блокировок. InnoDB записывает в счетчик каждой таблицы максимальный системный идентификатор транзакции из существующих на текущий момент.

Отсюда следует, что:

- Счетчик таблицы – это абсолютная нижняя граница транзакций, которым разрешено использовать кэш запросов. Если системный идентификатор транзакции равен 5 и транзакция установила блокировки на строки в таблице, а затем была зафиксирована, то транзакции с 1 по 4 никогда не смогут обратиться к кэшу запросов для чтения или записи, если в запросе участвует эта таблица.
- В счетчик таблицы записывается не идентификатор транзакции, которая заблокировала строки в ней, а системный идентификатор транзакции. В результате транзакции, которые блокируют строки в некоторой таблице, могут оказаться не в состоянии читать из кэша или писать в него для будущих запросов, в которых участвует эта таблица.

Запись в кэш, выборка из него и объявление сохраненных запросов недействительными – все это выполняется на уровне сервера, поэтому InnoDB не в состоянии ни обойти эти механизмы, ни отложить операции. Однако подсистема хранения может явно попросить сервер объявить недействительными запросы, в которых участвуют определенные таблицы. Это необходимо, когда ограничение внешнего ключа, например `ON DELETE CASCADE`, изменяет содержимое таблицы, не упомянутой в запросе.

В принципе, архитектура MVCC в InnoDB допускает обслуживание запросов из кэша в случае, когда изменения в таблице не затрагивают согласованное представление, видимое другим транзакциям. Однако реализовать это было бы сложно. Для простоты алгоритмы InnoDB срезают некоторые углы ценой запрета доступа к кэшу запросов из транзакций даже в тех случаях, когда без этого можно было бы обойтись.

Общие оптимизации кэша запросов

Многие решения, касающиеся проектирования схемы, запросов и приложения, так или иначе затрагивают кэш запросов. В дополнение к тому, о чем было рассказано в предыдущих разделах, мы хотели бы отметить еще несколько моментов.

- Наличие нескольких маленьких таблиц вместо одной большой часто повышает эффективность использования кэша запросов. При этом стратегия объявления записей недействительными работает на более мелком уровне. Но не придавайте этому соображению решающе-

го значения при проектировании схемы, поскольку другие факторы могут легко перевесить все достигаемые таким образом выгоды.

- Более эффективно собирать операции записи в пакет, чем выполнять их по одиночке, поскольку при таком подходе инвалидация (invalidation) запросов в кэше выполняется всего один раз.
- Мы обратили внимание, что сервер может на длительное время «зависать», будучи занят инвалидацией записей или вытеснением запросов из большого кэша. Это имеет место по крайней мере в версиях MySQL до 5.1. Самое простое решение – не задавать слишком большое значение параметра `query_cache_size`; 256 Мбайт должно быть более, чем достаточно.
- Невозможно контролировать кэш запросов на уровне отдельной базы данных или таблицы, но можно включать в некоторые команды `SELECT` модификаторы `SQL_CACHE` или `SQL_NO_CACHE`. Можно также включать или выключать кэширование для отдельного соединения, установив подходящее значение сеансовой переменной `query_cache_type`.
- Для приложений, выполняющих много операций записи, иногда можно повысить производительность, полностью отключив кэширование. При этом исключаются накладные расходы на кэширование запросов, которые в скором времени все равно были бы объявлены недействительными. Напомним, что кэширование отключается установкой параметра `query_cache_size` в 0, так что кэш при этом не потребляет памяти.

Если вы хотите избежать кэширования большинства запросов, но знаете, что некоторые из них смогли бы сильно выиграть от кэширования, то присвойте глобальному параметру `query_cache_type` значение `DEMAND`, а затем включите в нужные запросы подсказку `SQL_CACHE`. Это более трудоемко, зато дает более точный контроль над кэшем. Наоборот, если требуется кэшировать большинство запросов, а исключить лишь некоторые, то можно включить в них указание `SQL_NO_CACHE`.

Альтернативы кэшу запросов

В основу работы кэша запросов MySQL положен простой принцип: быстрее всего обрабатывает запрос, который вообще не нужно выполнять. Однако все равно приходится отправлять запрос, а серверу – что-то с ним делать, пусть и не очень много. Но так ли уж нужно обращаться к серверу с каждым запросом? Кэширование на стороне клиента может еще больше снизить нагрузку на сервер. Мы вернемся к этому вопросу в главе 10.

Хранение кода внутри MySQL

MySQL позволяет хранить код на стороне сервера в форме триггеров, хранимых процедур и хранимых функций. В версии MySQL 5.1 можно

также сохранять код в периодически выполняемых заданиях, которые называются *событиями*. Хранимые процедуры и функции вместе называются «хранимыми подпрограммами».

Для всех четырех видов хранимого кода применяется специальный расширенный язык SQL, содержащий такие процедурные конструкции, как циклы и условные предложения[1]. Самое существенное различие между разными видами хранимого кода – контекст, в котором этот код выполняется, то есть то, откуда поступают входные данные и куда направляются выходные. Хранимые процедуры и функции могут получать параметры и возвращать результаты, триггеры и события – не могут.

В принципе, хранимый код – это неплохой способ совместного и повторного использования кода. Джузеппе Максиа (Giuseppe Maxia) с коллегами написал библиотеку полезных хранимых подпрограмм общего назначения, которая находится по адресу *http://mysql-sr-lib.sourceforge.net*. Однако использовать хранимые подпрограммы из других СУБД затруднительно, так как в большинстве из них применяется собственный язык (исключение составляет DB2, в которой используется очень похожий синтаксис, основанный на том же стандарте).[2]

Нас будет больше интересовать, как хранимый код отражается на производительности, нежели способы его написания. Если вы планируете писать хранимые процедуры для MySQL, то имеет смысл познакомиться с книгой Guy Harrison, Steven Feuerstein «MySQL Stored Procedure Programming» (издательство O'Reilly).

У хранимого кода есть как сторонники, так и противники. Не принимая ничью сторону, мы просто перечислим плюсы и минусы этого подхода в MySQL. Начнем с плюсов.

- Код исполняется там, где находятся данные, поэтому можно сэкономить на сетевом трафике и уменьшить время задержки.
- Это одна из форм повторного использования кода. Она помогает содержать бизнес-правила в одном месте, что обеспечивает согласованное поведение и позволяет избежать лишних треволнений.
- Это упрощает политику выпуска версий и сопровождение.
- Это положительно сказывается на безопасности и позволяет более точно управлять привилегиями. Типичный пример – хранимая процедура для перевода средств с одного банковского счета на другой; она выполняет операцию в контексте транзакции и протоколиру-

---

[1] Этот язык является подмножеством языка SQL/PSM (Persistent Stored Modules – постоянно хранимые модули), являющегося частью стандарта SQL. Он определен в документе ISO/IEC 9075-4:2003 (E).

[2] Существуют также утилиты, предназначенные для переноса, например проект *tsql2mysql* (*http://sourceforge.net/projects/tsql2mysql*) для переноса из СУБД Microsoft SQL Server.

ет ее для последующего аудита. Можно дать приложению право вызывать хранимую процедуру, не открывая доступ к используемым в ней таблицам.

- Сервер кэширует планы выполнения хранимых процедур, что снижает накладные расходы на повторные вызовы.
- Поскольку код содержится на сервере, его можно развертывать, включать в резервную копию и сопровождать средствами сервера. Поэтому хранимый код хорошо приспособлен для задач обслуживания базы данных. У него нет никаких внешних зависимостей, например от библиотек на языке Perl или другого программного обеспечения, которое по тем или иным причинам нежелательно устанавливать на сервере.
- Это способствует разделению труда между программистами приложений и баз данных. Лучше, когда хранимые процедуры пишет специалист по базам данных, поскольку не всякий прикладной программист умеет создавать эффективные SQL-запросы.

Теперь перейдем к минусам.

- Вместе с MySQL не поставляются хорошие инструменты разработки и отладки, поэтому писать хранимый код для MySQL труднее, чем для других СУБД.
- Сам язык медленный и примитивный по сравнению с прикладными языками. Количество доступных функций ограничено, программировать сложные манипуляции со строками и нетривиальную логику затруднительно.
- Наличие хранимого кода может даже усложнить развертывание приложения. Приходится не только вносить изменения в саму программу и схему базы данных, но и развертывать код, хранящийся внутри сервера.
- Поскольку хранимые подпрограммы содержатся в базе данных, могут возникнуть дополнительные уязвимости. Например, реализация нестандартных криптографических функций в хранимой подпрограмме не защитит ваши данные в случае компрометации базы. Если бы криптографическая функция была реализована в коде приложения, то хакеру пришлось бы взломать как этот код, так и базу данных.
- Хранимые подпрограммы увеличивают нагрузку на сервер баз данных, а его обычно труднее масштабировать, чем веб-серверы или серверы приложений.
- MySQL предлагает весьма скромные средства контроля над ресурсами, выделяемыми для исполнения хранимого кода, поэтому ошибка может привести к отказу всего сервера.
- Реализация хранимого кода в MySQL довольно ограничена – планы выполнения кэшируются на уровне соединения, курсоры материа-

лизуются в виде временных таблиц и так далее. Мы еще расскажем об ограничениях, присущих отдельным средствам, когда будем их описывать.

- Код хранимых процедур в MySQL трудно профилировать. Сложно проанализировать журнал медленных запросов, когда вы видите в нем лишь запись `CALL XYZ('A')`, поскольку придется найти соответствующую процедуру и изучить все команды в ней.
- Хранимый код скрывает сложность; это упрощает разработку, но зачастую пагубно отражается на производительности.

Размышляя о том, стоит ли использовать хранимый код, спросите себя, где должна находиться бизнес-логика: в коде приложения или в базе данных? Распространены оба подхода. Нужно лишь понимать, что, прибегая к хранимому коду, вы помещаете логику на сервер базы данных.

Хранимые процедуры и функции

Архитектура MySQL и оптимизатора запросов налагает определенные ограничения на способы использования хранимых подпрограмм и степень их эффективности. В момент работы над этой книгой действовали следующие ограничения:

- Оптимизатор не обращает внимания на модификатор `DETERMINISTIC` в хранимых функциях для оптимизации нескольких вызовов в одном запросе.
- Оптимизатор не умеет оценивать стоимость выполнения хранимой функции.
- Для каждого соединения (connection) ведется отдельный кэш планов выполнения хранимых процедур. Если одна и та же процедура вызывается в нескольких соединениях, то ресурсы расходуются впустую на кэширование одного и того же плана. При использовании пула соединений или постоянных соединений (persistent connection) полезная жизнь плана выполнения может продлиться дольше.
- Хранимые процедуры плохо сочетаются с репликацией. Возможно, вы хотите реплицировать не вызов процедуры, а лишь сами изменения в наборе данных. Построчная репликация, появившаяся в версии MySQL 5.1, несколько смягчает эту проблему. Если в MySQL 5.0 включен режим ведения двоичного журнала, то сервер «настаивает» на том, чтобы вы либо задали для всех хранимых процедур модификатор `DETERMINISTIC`, либо включили параметр с хитрым названием `log_bin_trust_function_creators` (доверять авторам функций).

Мы обычно предпочитаем писать небольшие, простые хранимые процедуры. На наш взгляд, сложную логику лучше оставить вовне базы данных и реализовывать ее с помощью более выразительного и гибкого процедурного языка. К тому же он может открыть доступ к больше-

му объему вычислительных ресурсов и потенциально позволяет реализовать различные формы кэширования.

Однако для некоторых операций хранимые процедуры могут оказаться гораздо быстрее, особенно если речь идет о мелких запросах. Если запрос достаточно мал, то накладные расходы на его разбор и передачу по сети занимают заметную долю всего времени обработки. Чтобы доказать это, мы написали простенькую хранимую процедуру, которая вставляет в таблицу заданное количество строк. Вот ее код:

```
1  DROP PROCEDURE IF EXISTS insert_many_rows;
2
3  delimiter //
4
5  CREATE PROCEDURE insert_many_rows (IN loops INT)
6  BEGIN
7    DECLARE v1 INT;
8    SET v1=loops;
9    WHILE v1 > 0 DO
10     INSERT INTO test_table values(NULL,0,
11       'qqqqqqqqqqqwwwwwwwwwweeeeeeeeeerrrrrrrrrrttttttttttt',
12        'qqqqqqqqqqqwwwwwwwwwweeeeeeeeeerrrrrrrrrrttttttttttt');
13     SET v1 = v1 - 1;
14   END WHILE;
15 END;
16 //
17
18 delimiter ;
```

Затем мы сравнили время вставки миллиона строк с помощью этой процедуры и последовательной вставки из клиентского приложения. Структура таблицы и состав оборудования значения не имеют, важно лишь относительное быстродействие при разных подходах. Просто ради интереса мы померили, сколько времени уходит на те же запросы при соединении с помощью MySQL Proxy. Чтобы не усложнять ситуацию, мы прогоняли эталонные тесты на одном сервере, где находились также клиентское приложение и экземпляр MySQL Proxy. Результаты представлены в табл. 5.1.

Таблица 5.1. Общее время вставки миллиона строк по одной

Метод	Общее время, сек
Хранимая процедура	101
Клиентское приложение	279
Клиентское приложение с соединением через MySQL Proxy	307

Хранимая процедура работает гораздо быстрее, главным образом из-за отсутствия накладных расходов на передачу по сети, разбор, оптимизацию и т. д.

В разделе «SQL-интерфейс к подготовленным командам» (стр. 288) мы приведем пример типичной хранимой процедуры для задач обслуживания базы данных.

Триггеры

Триггеры дают возможность выполнить код, когда встречаются команды `INSERT`, `UPDATE` или `DELETE`. Вы можете заставить MySQL обрабатывать триггеры до и/или после выполнения самой команды. Триггер не возвращает значения, но в состоянии читать и/или изменять данные, которые были модифицированы командой, вызвавшей его срабатывание. Следовательно, с помощью триггеров можно реализовывать ограничения целостности или бизнес-логику, которую иначе пришлось бы программировать на стороне клиента. В качестве примера упомянем эмуляцию внешних ключей для подсистем хранения, которые сами по себе их не поддерживают, например MyISAM.

Триггеры позволяют упростить логику приложения и повысить производительность, поскольку избавляют от необходимости обмениваться данными по сети. Они также бывают полезны для автоматического обновления денормализованных и сводных таблиц. Так, в демонстрационной базе Sakila триггеры применяются для поддержания таблицы `film_text` в актуальном состоянии.

В настоящее время реализация триггеров в MySQL неполна. Если у вас есть обширный опыт работы с триггерами в других СУБД, то не стоит предполагать, что в MySQL они будут работать точно так же. В частности, отметим следующие моменты.

- В каждой таблице для каждого события можно определить только один триггер (иными словами, не может быть двух триггеров, срабатывающих по событию `AFTER INSERT`).
- MySQL поддерживает только триггеры на уровне строки, т. е. триггер всегда работает в режиме `FOR EACH ROW`, а не для команды в целом. При обработке больших наборов данных это гораздо менее эффективно[1].

К MySQL относятся также следующие соображения общего характера о триггерах:

- Триггер может затруднить понимание того, что в действительности делает сервер, поскольку простая, на первый взгляд, команда нередко инициирует большой объем «невидимой» работы. Так, если триггер обновляет связанную таблицу, то количество затрагиваемых командой строк может удвоиться.
- Триггеры трудно отлаживать. При использовании триггеров зачастую очень сложно найти причину низкой производительности.

[1] Более того, классы задач, решаемые триггерами для команды в целом и для каждой строки в отдельности, различны. – *Прим. науч. ред.*

- Триггеры могут стать причиной неочевидных взаимоблокировок и ожиданий блокировок. Если в триггере возникает ошибка, то с ошибкой завершается и исходный запрос, а если вы не знаете о существовании триггера, то разобраться, о чем говорит код ошибки, будет затруднительно.

С точки зрения производительности, самое серьезное ограничение – это реализация в MySQL только триггеров FOR EACH ROW. Иногда из-за этого триггер работает настолько медленно, что оказывается непригоден для поддержания сводных и кэширующих таблиц. Основной причиной для использования триггеров вместо периодического массового обновления состоит в том, что данные в любой момент времени согласованы.

Кроме того, триггеры не всегда гарантируют атомарность. Например, действия, выполненные триггером по обновлению таблицы типа MyISAM, невозможно откатить, если в команде, ставшей причиной его срабатывания, обнаружится ошибка. Да и сам триггер может приводить к ошибкам. Предположим, что вы присоединили триггер типа AFTER UPDATE к MyISAM-таблице и обновляете в нем другую MyISAM-таблицу. Если из-за ошибки в триггере вторую таблицу обновить не удастся, то откат обновления первой не будет выполнен.

Триггеры над таблицами типа InnoDB выполняются в контексте текущей транзакции, поэтому произведенные ими действия будут атомарны, то есть фиксируются или откатываются вместе с вызвавшей их командой. Однако если триггер над InnoDB-таблицей проверяет данные в другой таблице, чтобы проконтролировать ограничение целостности, то следует помнить о механизме MVCC, так как иначе можно получить неправильные результаты. Пусть, например, требуется эмулировать внешние ключи, но использовать механизм внешних ключей, встроенный в InnoDB, вы по каким-то причинам не хотите. Можно написать триггер типа BEFORE INSERT, который проверит наличие соответствующей записи в другой таблице, но, если читать в триггере данные из нее не с помощью команды SELECT FOR UPDATE, то в случае параллельного обновления этой таблицы можно получить неверные результаты.

Мы вовсе не хотим отговаривать вас от использования триггеров. Напротив, они бывают очень полезны, особенно для проверки ограничений, задач обслуживания системы и поддержания денормализованных данных в согласованном состоянии.

Триггеры можно применять и с целью протоколирования обновлений строк. Это полезно для «самописных» схем репликации, когда требуется разорвать соединение между двумя системами, изменить данные, а затем восстановить синхронизацию. Простой пример – группа пользователей, которые берут с собой ноутбуки в другой офис. Произведенные ими изменения нужно затем синхронизировать с главной базой данных, после чего скопировать главную базу обратно на отдельные ноутбуки. Эта задача требует двусторонней синхронизации. Триггеры неплохо подходят для построения подобных систем. На каждом ноутбуке

с помощью триггеров все операции модификации данных протоколируются в специальные таблицы с указанием того, какие строки изменились. Затем специально написанная программа синхронизации переносит изменения в главную базу данных. А уже потом с помощью стандартной репликации MySQL можно синхронизировать ноутбуки с главной базой, в результате чего на каждом переносном компьютере окажутся изменения, произведенные на всех остальных ноутбуках.

Иногда удается даже обойти ограничение `FOR EACH ROW`. Роланд Боумен (Roland Bouman) обнаружил, что функция `ROW_COUNT()` внутри триггера возвращает 1 всегда, кроме первой строки в триггере типа `BEFORE`. Этим фактом можно воспользоваться для того, чтобы подавить выполнение триггера для каждой измененной строки, активировав его лишь один раз на всю команду. Это, конечно, не триггер уровня команды, но все-таки полезная техника, которая позволяет в некоторых случаях эмулировать триггер `BEFORE` уровня команды. Возможно, описанная особенность на самом деле является ошибкой, которая рано или поздно будет исправлена, поэтому относитесь к этому приему с осторожностью и проверяйте, работает ли он после перехода на новую версию. Вот как можно использовать эту недокументированную возможность:

```
CREATE TRIGGER fake_statement_trigger
BEFORE INSERT ON sometable
FOR EACH ROW
BEGIN
   DECLARE v_row_count INT DEFAULT ROW_COUNT( );
   IF v_row_count <> 1 THEN
      -- Здесь должен быть ваш код
   END IF;
END;
```

События

События – это новый вид хранимого кода, появившийся в MySQL начиная с версии 5.1. Они похожи на задания *cron*, но выполняются целиком внутри сервера MySQL. Можно создать событие, которое будет обрабатывать SQL-код в требуемый момент времени или с заданным интервалом. Обычно поступают так: оформляют сложный SQL-код в виде хранимой процедуры, а событие просто вызывает ее с помощью команды `CALL`.

События выполняются специальным потоком планировщика событий, поскольку никак не связаны с соединениями. Они не принимают параметров и не возвращают значений – просто потому, что их неоткуда получить и некому вернуть. Выполненные событием команды протоколируются в журнале сервера, если он активирован, но понять, что они были вызваны именно из события, затруднительно. Можно также заглянуть в таблицу `INFORMATION_SCHEMA.EVENTS` и посмотреть на состояние события, например узнать, когда оно в последний раз вызывалось.

К событиям относятся все те же соображения, что и к хранимым процедурам: это дополнительная нагрузка на сервер. Накладные расходы на сам запуск событий минимальны, но команды, которые выполняются внутри них, могут оказать заметное воздействие на производительность. Разумно использовать события для периодического запуска задач обслуживания, в том числе перестроения кэша и сводных таблиц, эмулирующих материализованные представления, а также для сохранения переменных состояния с целью мониторинга и диагностики.

В следующем примере создается событие, которое один раз в неделю запускает хранимую процедуру в указанной базе данных[1]:

```
CREATE EVENT optimize_somedb ON SCHEDULE EVERY 1 WEEK
DO
CALL optimize_tables('somedb');
```

Можно указать, следует ли реплицировать события на подчиненный сервер. Иногда это необходимо, иногда – нет. Возьмем только что рассмотренный пример: возможно, вы хотите выполнять операцию `OPTIMIZE TABLE` на всех подчиненных серверах, но имейте в виду, что производительность сервера в целом может пострадать (в частности, из-за установки табличных блокировок), если все подчиненные серверы начнут оптимизировать таблицы в одно и то же время.

Наконец, если для выполнения периодического события требуется длительное время, то может случиться так, что оно в очередной раз активируется в момент, когда предыдущее еще не завершилось. MySQL не защищает вас от этого, так что придется написать собственную логику взаимного исключения. Чтобы гарантировать, что работает только один экземпляр события, можно воспользоваться функцией `GET_LOCK()`:

```
CREATE EVENT optimize_somedb ON SCHEDULE EVERY 1 WEEK
DO
BEGIN
   DECLARE CONTINUE HANLDER FOR SQLEXCEPTION
      BEGIN END;
   IF GET_LOCK('somedb', 0) THEN
      DO CALL optimize_tables('somedb');
   END IF;
   DO RELEASE_LOCK('somedb');
END
```

«Пустой» обработчик продолжения гарантирует, что событие освободит блокировку, даже если хранимая процедура возбудит исключение.

Хотя события никак не связаны с соединениями, ассоциация между ними и потоками все же имеется. Существует главный поток планировщика событий, который необходимо активировать в конфигурационном файле сервера или командой `SET`:

[1] Ниже мы покажем, как создать эту хранимую процедуру.

```
mysql> SET GLOBAL event_scheduler := 1;
```

Будучи активирован, этот поток создает новый поток для выполнения каждого события. Внутри кода события функция CONNECTION_ID() возвращает, как обычно, уникальное значение, хотя «соединения» как такового и не существует (функция CONNECTION_ID() на самом деле возвращает просто идентификатор потока). Сведения о выполнении событий можно найти в журнале ошибок сервера.

Сохранение комментариев в хранимом коде

Код хранимых процедур и функций, триггеров и событий может быть довольно длинным, поэтому неплохо бы снабдить его комментариями. Но комментарии иногда не сохраняются на сервере, поскольку стандартный клиент командной строки может вырезать их (эта «особенность» командного клиента, возможно, вызывает у вас раздражение, но *c'est la vie*.)

Чтобы оставить комментарии в хранимом коде, существует полезный прием: воспользоваться зависящими от версии комментариями, которые сервер воспринимает как потенциально исполняемый код (т. е. код, который сервер будет выполнять, если номер его версии не меньше указанного). И сервер, и клиентские программы знают, что это не обычные комментарии, поэтому не удаляют их. Чтобы такой «код» гарантированно не выполнялся, можно просто задать очень большой номер версии, скажем 99999. Вот, например, как можно документировать триггер из предыдущего примера, чтобы сделать его более понятным:

```
CREATE TRIGGER fake_statement_trigger
BEFORE INSERT ON sometable
FOR EACH ROW
BEGIN
   DECLARE v_row_count INT DEFAULT ROW_COUNT( );
   /*!99999
     ROW_COUNT( ) равно 1 для всех строк, кроме первой,
     поэтому этот триггер выполняется один раз на всю команду.
   */
```

Курсоры

В настоящее время MySQL предлагает однонаправленные (с прокруткой вперед) серверные курсоры только для чтения, и использовать их можно лишь в хранимых процедурах. Курсор позволяет построчно обойти результат запроса, извлекая строки в переменные для последующей обработки. Хранимая процедура позволяет открывать сразу несколько курсоров, причем они могут быть «вложены» друг в друга (во вложенных циклах).

Возможно, в будущих версиях MySQL появятся и обновляемые курсоры, но ни в одной из имеющихся сейчас их нет. Курсоры допускают

только чтение, потому что обходят временные таблицы, а не таблицы, в которых хранятся реальные данные.

Для непосвященного устройство курсоров в MySQL таит немало сюрпризов. Поскольку курсоры реализованы с помощью временных таблиц, у разработчика может возникнуть ложное ощущение эффективности. Самое важное, что нужно помнить, – это то, что *курсор выполняет весь запрос в момент открытия*. Рассмотрим следующую процедуру:

```
1 CREATE PROCEDURE bad_cursor( )
2 BEGIN
3    DECLARE film_id INT;
4    DECLARE f CURSOR FOR SELECT film_id FROM sakila.film;
5    OPEN f;
6    FETCH f INTO film_id;
7    CLOSE f;
8 END
```

В этом примере видно, что курсор можно закрыть до завершения обхода результатов. Разработчик, привыкший к Oracle или Microsoft SQL Server, возможно, и не заметит в этой процедуре ничего плохого, но в MySQL она приводит к массе лишней работы. Профилирование с помощью команды `SHOW STATUS` показывает, что эта процедура выполняет 1000 операций чтения индекса и 1000 операций вставки. А все потому, что в таблице `sakila.film` ровно 1000 строк. Все 1000 операций чтения и записи производятся при обработке строки 5, еще до начала выполнения строки 6.

Мораль заключается в том, что раннее закрытие курсора, выбирающего данные из большой таблицы, не дает никакой экономии. Если нужно всего несколько строк, воспользуйтесь фразой `LIMIT`.

Из-за курсоров MySQL может выполнять и дополнительные операции ввода/вывода, которые иногда оказываются весьма медленными. Поскольку временные таблицы в памяти не поддерживают типы `BLOB` и `TEXT`, то MySQL вынуждена создавать временные таблицы на диске для курсоров, извлекающих данные таких типов. Но даже если это не так, временные таблицы, размер которых превышает значение параметра `tmp_table_size`, все равно создаются на диске.

MySQL не поддерживает курсоры на стороне клиента, но в клиентском API есть функции, которые эмулируют курсор путем загрузки всего результирующего набора в память. На самом деле, это ничем не отличается от копирования результата в созданный приложением массив с последующей манипуляцией этим массивом. Дополнительную информацию о последствиях загрузки всех результатов в память клиента см. в разделе «Клиент-серверный протокол MySQL» на стр. 211.

Подготовленные команды

Начиная с версии MySQL 4.1 сервер поддерживает *подготовленные (prepared) команды*; которые используют расширенный двоичный клиент-

серверный протокол, обеспечивающий эффективную передачу данных между клиентом и сервером. Получить доступ к функциональности подготовленных команд позволяют библиотеки, поддерживающие новый протокол, например MySQL C API. Библиотеки MySQL Connector/J и MySQL Connector/NET предлагают те же возможности для Java и .NET соответственно. Существует также SQL-интерфейс к подготовленным командам, который мы рассмотрим ниже.

В момент создания подготовленной команды клиентская библиотека посылает серверу прототип будущего запроса. Сервер разбирает и обрабатывает эту «заготовку» запроса, сохраняет структуру, представляющую частично оптимизированный запрос, и возвращает клиенту *дескриптор команды* (*statement handle*).

В подготовленных командах могут присутствовать параметры, обозначаемые вопросительными знаками, вместо которых в момент выполнения подставляются фактические значения. Например, можно подготовить такой запрос:

```
mysql> INSERT INTO tbl(col1, col2, col3) VALUES (?, ?, ?) ;
```

Чтобы впоследствии выполнить этот запрос, серверу необходимо отправить дескриптор команды и значения всех параметров, представленных вопросительными знаками. Это действие можно повторять сколько угодно раз. Способ отправки дескриптора команды серверу зависит от языка программирования. Один из вариантов – воспользоваться MySQL-коннекторами для Java и .NET. Многие клиентские библиотеки, компонуемые с библиотеками MySQL на языке C, тоже предоставляют некоторый интерфейс к двоичному протоколу; вам следует ознакомиться с документацией по конкретному MySQL API.

Использование подготовленных команд может оказаться эффективнее повторного выполнения запросов по нескольким причинам.

- Серверу нужно разобрать запрос только один раз, так что на разборе и еще кое-каких операциях можно сэкономить.
- Сервер должен проделать некоторые шаги оптимизации однократно, так как частичный план выполнения запроса уже закэширован.
- Отправка параметров в двоичном виде эффективнее передачи в виде ASCII-текста. Например, для отправки значения типа `DATE` нужно всего 3 байта вместо 10 при передаче в ASCII-виде. Но наибольшая экономия достигается для значений типа `BLOB` и `TEXT`, которые можно отправлять серверу блоками, а не одним гигантским куском. Таким образом, двоичный протокол позволяет экономить память клиента, а заодно уменьшает сетевой трафик и устраняет накладные расходы на преобразование данных из естественного формата хранения в кодировку ASCII.
- Для каждого выполнения запроса нужно посылать только параметры, а не весь текст запроса, что еще больше снижает сетевой трафик.

- MySQL сохраняет параметры прямо в буферах сервера, так что серверу не нужно копировать значения из одного места памяти в другое.

Подготовленные команды также повышают безопасность. Нет необходимости экранировать специальные символы на уровне приложения, а это удобнее и делает программу менее уязвимой к внедрению SQL-кода (SQL injection) или другим видам атак. Никогда нельзя доверять данным, полученным от пользователей, даже в случае задействования подготовленных команд.

Двоичный протокол применим *только* к подготовленным командам. При отправке запросов с помощью обычной функции API `mysql_query()` двоичный протокол *не* используется. Многие клиентские библиотеки позволяют «подготовить» команду, включив в ее текст вопросительные знаки, а затем задавать фактические значения параметров при каждом выполнении, но это лишь эмуляция цикла «подготовка-отправка» в клиентском коде, а реально каждый запрос отправляется серверу с помощью `mysql_query()`.

Оптимизация подготовленных команд

MySQL кэширует частичные планы выполнения для подготовленных команд, но некоторые оптимизации зависят от фактических значений параметров, поэтому не могут быть заранее вычислены и закэшированы. Все оптимизации можно разделить на три группы в зависимости от того, когда они производятся. На момент написания этой книги действовала следующая классификация, которая в будущем может измениться.

На этапе подготовки

Сервер разбирает текст запроса, устраняет отрицания и переписывает подзапросы.

При первом выполнении

Сервер упрощает вложенные соединения и преобразует `OUTER JOIN` в `INNER JOIN` там, где это возможно.

При каждом выполнении

Сервер выполняет следующие действия:

- Исключает из рассмотрения секции (prunes partitions)
- Всюду, где это возможно, заменяет `COUNT()`, `MIN()` и `MAX()` константами
- Устраняет константные подвыражения
- Обнаруживает константные таблицы
- Распространяет равенство
- Анализирует и оптимизирует методы доступа `ref`, `range` и `index_merge`
- Оптимизирует порядок соединения таблиц

За дополнительной информацией обо всех этих оптимизациях обратитесь к главе 4.

SQL-интерфейс к подготовленным командам

SQL-интерфейс к подготовленным командам существует, начиная с версии 4.1. Вот пример использования подготовленной команды из SQL:

```
mysql> SET @sql := 'SELECT actor_id, first_name, last_name
    -> FROM sakila.actor WHERE first_name = ?';
mysql> PREPARE stmt_fetch_actor FROM @sql;
mysql> SET @actor_name := 'Penelope';
mysql> EXECUTE stmt_fetch_actor USING @actor_name;
+----------+------------+-----------+
| actor_id | first_name | last_name |
+----------+------------+-----------+
|        1 | PENELOPE   | GUINESS   |
|       54 | PENELOPE   | PINKETT   |
|      104 | PENELOPE   | CRONYN    |
|      120 | PENELOPE   | MONROE    |
+----------+------------+-----------+
mysql> DEALLOCATE PREPARE stmt_fetch_actor;
```

Получив такие команды, сервер транслирует их в те же самые операции, которые получились бы при использовани клиентской библиотеки. Поэтому никакого особого двоичного протокола для создания и выполнения подготовленных команд вам применять не придется.

Как легко видеть, подобный синтаксис несколько тяжеловеснее, чем при вводе обычных команд `SELECT`. Тогда в чем же преимущество такого использования подготовленных команд?

Основное применение они находят в хранимых процедурах. В MySQL 5.0 подготовленные команды разрешено использовать внутри хранимых процедур, а их синтаксис аналогичен SQL-интерфейсу. Это означает, что в хранимых процедурах можно строить и выполнять «динамические SQL-команды» путем конкатенации строк, что еще больше повышает гибкость подобных конструкций. Ниже приведен пример хранимой процедуры, в которой для каждой таблицы в указанной базе данных выполняется команда `OPTIMIZE TABLE`:

```
DROP PROCEDURE IF EXISTS optimize_tables;
DELIMITER //
CREATE PROCEDURE optimize_tables(db_name VARCHAR(64))
BEGIN
   DECLARE t VARCHAR(64);
   DECLARE done INT DEFAULT 0;
   DECLARE c CURSOR FOR
      SELECT table_name FROM INFORMATION_SCHEMA.TABLES
      WHERE TABLE_SCHEMA = db_name AND TABLE_TYPE = 'BASE TABLE';
   DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
```

```
    OPEN c;
    tables_loop: LOOP
       FETCH c INTO t;
       IF done THEN
          CLOSE c;
          LEAVE tables_loop;
       END IF;
       SET @stmt_text := CONCAT("OPTIMIZE TABLE ", db_name, ".", t);
       PREPARE stmt FROM @stmt_text;
       EXECUTE stmt;
       DEALLOCATE PREPARE stmt;
    END LOOP;
    CLOSE c;
END//
DELIMITER ;
```

Использовать эту процедуру можно следующим образом:

```
mysql> CALL optimize_tables('sakila');
```

Цикл в данной процедуре можно записать и по-другому:

```
REPEAT
   FETCH c INTO t;
   IF NOT done THEN
      SET @stmt_text := CONCAT("OPTIMIZE TABLE ", db_name, ".", t);
      PREPARE stmt FROM @stmt_text;
      EXECUTE stmt;
      DEALLOCATE PREPARE stmt;
   END IF;
UNTIL done END REPEAT;
```

Между этими двумя конструкциями имеется важное различие: в случае `REPEAT` условие цикла проверяется дважды на каждой итерации. В данной ситуации это, пожалуй, не очень существенно, так как мы просто сравниваем целочисленные значения, но более сложная проверка может обойтись и дороже.

Порождение имен таблиц и баз данных с помощью конкатенации строк – это отличный пример применения SQL-интерфейса к подготовленным командам, поскольку так можно строить директивы, не принимающие параметров. Невозможно подставить параметры вместо имен таблиц и баз данных, так как они являются идентификаторами. Еще одно возможное применение – конструирование фразы `LIMIT`, которая тоже не допускает параметризации.

SQL-интерфейс полезен для ручного тестирования подготовленных команд, но других применений вне хранимых процедур, пожалуй, и не назовешь. Поскольку интерфейс основан на SQL, двоичный протокол не используется, а сетевой трафик не сокращается, так как при наличии параметров приходится отправлять дополнительные запросы для задания их значений. В некоторых частных случаях удается получить

выигрыш, например при подготовке очень длинной строки SQL-запроса, который будет выполняться многократно без параметров. Однако всякий раз, когда вы думаете, что применение SQL-интерфейса к подготовленным командам позволит что-то сэкономить, проводите эталонное тестирование.

Ограничения подготовленных команд

У подготовленных команд есть несколько ограничений и подводных камней.

- Подготовленные команды локальны по отношению к соединению, поэтому в другом соединении тот же самый дескриптор использовать нельзя. По той же причине клиент, который разрывает и вновь устанавливает соединение, теряет все подготовленные команды (смягчить эту проблему позволяет пул соединений и устойчивые соединения).
- До версии MySQL 5.1 подготовленные команды не попадали в кэш запросов.
- Использование подготовленных команд не всегда эффективно. Если подготовленная команда выполняется всего один раз, вы можете потратить на подготовку больше времени, чем ушло бы на выполнение обычной SQL-команды. Кроме того, для подготовки команды необходимо дополнительное обращение к серверу.
- В настоящее время подготовленные команды нельзя использовать в хранимых функциях (но можно в хранимых процедурах).
- Если вы забудете освободить дескриптор подготовленной команды, то возникнет «утечка». Это может приводить к потерям большого количества ресурсов сервера. Кроме того, поскольку существует глобальное ограничение на количество подготовленных команд, такая ошибка может отразиться на других соединениях – они не смогут подготовить команду.

Определяемые пользователем функции

MySQL поддерживает *определяемые пользователем функции* (*user-defined functions – UDF*) уже давно. В отличие от хранимых функций, которые пишутся на языке SQL, определяемые пользователем функции можно писать на любом языке программирования, который поддерживает соглашения о вызове, принятые в языке C.

UDF-функции необходимо сначала скомпилировать, а затем динамически скомпоновать с сервером. Поэтому они платформенно-зависимы, зато наделяют разработчика огромной мощью. Такие функции могут работать очень быстро и пользоваться необозримой функциональностью, которую предлагает операционная система и имеющиеся библиотеки. Хранимые функции, написанные на SQL, хороши для про-

стых операций, например для вычисления расстояния по дуге большого круга между двумя точками на глобусе, но если требуется отправить какие-то пакеты по сети, то без UDF-функций не обойтись. Кроме того, в текущей версии на SQL нельзя написать агрегатные функции, а с помощью UDF это делается легко.

Но большая власть налагает и большую ответственность. Ошибка в UDF-функции может привести к аварийному останову сервера, запортить память или данные пользователя и вообще внести хаос, как любой плохо написанный код на C.

В отличие от хранимых функций, написанных на SQL, UDF-функции пока не умеют читать из таблиц и писать в них – по крайней мере, не в том же транзакционном контексте, в котором выполняется вызвавшая их команда. Это означает, что они полезны скорее для чистых вычислений или для взаимодействия с внешним миром. MySQL постоянно расширяет возможности обращения к ресурсам, внешним по отношению к серверу. Отличным примером того, что можно сделать с помощью UDF-функций, являются написанные Брайаном Эйкером (Brian Aker) и Патриком Гэлбрайтом (Patrick Galbraith) функции для взаимодействия с сервером *memcached* (*http://tangent.org/586/Memcached_Functions_for_MySQL.html*).

Применяя UDF-функции, внимательно следите за изменениями в новых версиях MySQL, поскольку не исключено, что код придется перекомпилировать или даже модифицировать, чтобы он работал с новым сервером. Кроме того, UDF-функции должны быть полностью потоковобезопасны (thread-safe), так как исполняются внутри сервера MySQL, то есть в многопоточном окружении.

Для MySQL есть много хороших библиотек с готовыми UDF-функциями и немало примеров, показывающих, как их писать. Самый большой репозиторий UDF-функций находится по адресу *http://www.mysqludf.org*.

Ниже приведен код UDF-функции `NOW_USEC()`, которую мы будем использовать для измерения скорости репликации (см. раздел «Насколько быстро работает репликация?» главы 8 на стр. 501).

```
#include <my_global.h>
#include <my_sys.h>
#include <mysql.h>

#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>

extern "C" {
   my_bool now_usec_init(UDF_INIT *initid, UDF_ARGS *args, char *message);
   char *now_usec(
```

```
                  UDF_INIT *initid,
                  UDF_ARGS *args,
                  char *result,
                  unsigned long *length,
                  char *is_null,
                  char *error);
}

my_bool now_usec_init(UDF_INIT *initid, UDF_ARGS *args, char *message) {
   return 0;
}

char *now_usec(UDF_INIT *initid, UDF_ARGS *args, char *result,
               unsigned long *length, char *is_null, char *error) {
   struct timeval tv;
   struct tm* ptm;
   char time_string[20]; /* e.g. "2006-04-27 17:10:52" */
   char *usec_time_string = result;
   time_t t;

   /* Получить текущее время и преобразовать его в структуру tm struct. */
   gettimeofday (&tv, NULL);
   t = (time_t)tv.tv_sec;
   ptm = localtime (&t);

   /* Отформатировать дату и время с точностью до секунд. */
   strftime (time_string, sizeof (time_string), "%Y-%m-%d %H:%M:%S", ptm);

   /* Напечатать отформатированное время в секундах,
    * затем десятичную точку и количество микросекунд. */
   sprintf(usec_time_string, "%s.%06ld\n", time_string, tv.tv_usec);

   *length = 26;

   return(usec_time_string);
}
```

Представления

Представления – это широко распространенный в СУБД механизм, который был добавлен в MySQL, начиная с версии 5.0. В MySQL *представление* – это таблица, в которой не хранятся данные. Вместо этого информация, «находящаяся» в таблице, берется из результатов обработки SQL-запроса.

В этой книге мы не собираемся объяснять, как создавать и использовать представления; об этом можно прочитать в соответствующем разделе руководства по MySQL и в другой документации. Во многих отношениях MySQL трактует представления точно так же, как таблицы, – они даже разделяют общее пространство имен. Тем не менее, в MySQL

таблицы и представления – не одно и то же. Например, для представления нельзя создать триггер, и невозможно удалить представление командой DROP TABLE.

Чтобы представления не стали причиной падения производительности, важно понимать, как они реализованы и как взаимодействуют с оптимизатором запросов. Для иллюстрации работы представлений мы воспользуемся демонстрационной базой данных world:

```
mysql> CREATE VIEW Oceania AS
    ->    SELECT * FROM Country WHERE Continent = 'Oceania'
    ->    WITH CHECK OPTION;
```

Чтобы реализовать этот запрос, серверу проще всего выполнить команду SELECT и поместить результаты во временную таблицу. В дальнейшем этой временной таблицей можно подменять все ссылки на представление. Разберемся, как это могло бы сработать на примере следующего запроса:

```
mysql> SELECT Code, Name FROM Oceania WHERE Name = 'Australia';
```

Вот как сервер мог бы подойти к его выполнению (имя временной таблицы выбрано исключительно для примера):

```
mysql> CREATE TEMPORARY TABLE TMP_Oceania_123 AS
    ->    SELECT * FROM Country WHERE Continent = 'Oceania';
mysql> SELECT Code, Name FROM TMP_Oceania_123 WHERE Name = 'Australia';
```

Очевидно, при таком подходе не избежать проблем с производительностью и оптимизацией. Более правильный способ реализации представлений – переписать запрос, в котором встречается представление, объединив SQL-код самого запроса с SQL-кодом представления. Ниже показано, как мог бы выглядеть исходный запрос после подобного объединения:

```
mysql> SELECT Code, Name FROM Country
    -> WHERE Continent = 'Oceania' AND Name = 'Australia';
```

MySQL может применять оба метода. Для этой цели имеются два алгоритма: MERGE и TEMPTABLE, причем по возможности MySQL старается использовать алгоритм MERGE (объединение). MySQL умеет даже объединять вложенные определения представлений, когда в определении одного представления имеется ссылка на другое. Посмотреть, что получилось в результате переписывания запроса, позволяет команда EXPLAIN EXTENDED, сопровождаемая командой SHOW WARNINGS.

Если при реализации представления был использован алгоритм TEMPTABLE, то EXPLAIN обычно показывает его производную (DERIVED) таблицу. На рис. 5.4 показаны обе реализации.

MySQL применяет метод TEMPTABLE, когда в определении представления встречаются фразы GROUP BY, DISTINCT, агрегатные функции, фраза UNION, подзапросы и вообще любые другие конструкции, которые не сохраня-

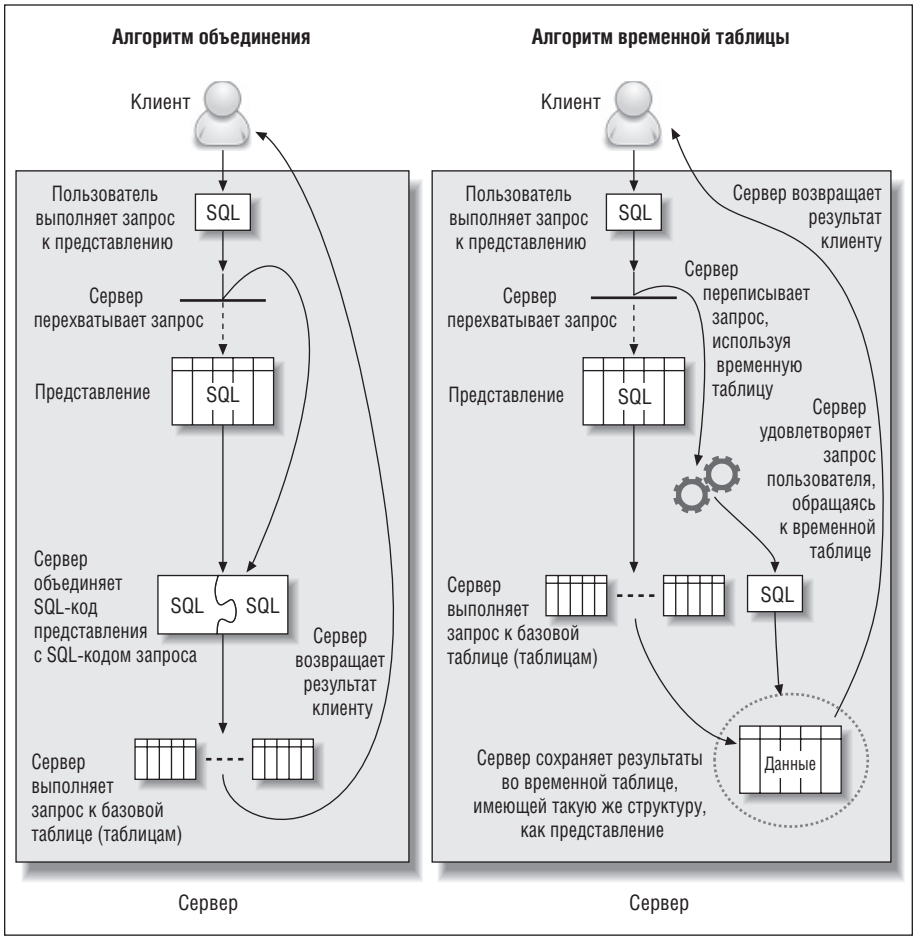


Рис. 5.4. *Две реализации представления*

ют взаимно однозначное соответствие между строками в базовых таблицах и строками в представлении. Приведенный выше перечень неполон и в будущем может измениться. Чтобы узнать, какой из двух методов будет применен для конкретного представления, можно выполнить команду `EXPLAIN` для тривиального запроса `SELECT` к этому представлению:

```
mysql> EXPLAIN SELECT * FROM <view_name>;
+----+-------------+
| id | select_type |
+----+-------------+
|  1 |     PRIMARY |
|  2 |     DERIVED |
+----+-------------+
```

Значение DERIVED в колонке select_type говорит, что для данного представления будет использован метод TEMPTABLE.

Обновляемые представления

Обновляемое представление позволяет изменять данные в базовой таблице через ее представление. При соблюдении определенных условий к представлению можно применять команды UPDATE, DELETE и даже INSERT, как к обычной таблице. Например, следующая операция допустима:

```
mysql> UPDATE Oceania SET Population = Population * 1.1 WHERE Name = 'Australia';
```

Представление не является обновляемым, если оно содержит фразы GROUP BY, UNION, агрегатную функцию, а также еще в нескольких случаях. Запрос, изменяющий данные, может содержать операцию соединения, но все изменяемые колонки должны находиться в одной таблице[1] . Представление, для реализации которого применен метод TEMPTABLE, не является обновляемым.

Фраза CHECK OPTION, которая была включена в определение представления из предыдущего раздела, гарантирует, что все строки, измененные через представление, будут соответствовать условию WHERE в определении представления и после изменения. Поэтому мы не можем изменять столбец Continent или вставлять строку с другим значением Continent. В обоих случаях сервер выдал бы ошибку[2]:

```
mysql> UPDATE Oceania SET Continent = 'Atlantis';
ERROR 1369 (HY000): CHECK OPTION failed 'world.Oceania'
```

В некоторых СУБД для представлений разрешено определять триггеры типа INSTEAD OF; это позволяет уточнить, что должно происходить при попытке модифицировать данные представления, но MySQL триггеры над представлениями не поддерживает. Не исключено, что в будущих версиях некоторые ограничения на обновляемые представления будут сняты, что откроет дорогу для ряда интересных и полезных применений. Одна из потенциальных возможностей – построение объединяющих таблиц над таблицами, реализованными с помощью разных подсистем хранения. С точки зрения производительности, это стало бы чрезвычайно полезным применением представлений.

Представления и производительность

Большинство разработчиков не считают, что представления могут както повысить производительность, однако в MySQL это не так. Кроме того, представления могут быть подспорьем для других методов повы-

---

[1] Имеется в виду, что запрос, по которому построено представление, может содержать соединения, но при обновлении можно изменять только столбцы одной таблицы. – *Прим. науч. ред.*

[2] Atlantis – Атлантида (легендарный континент). – *Прим. ред.*

шения производительности. Например, если рефакторинг схемы происходит поэтапно, то иногда с помощью представлений можно сохранить работоспособность кода, который обращается к таблице с изменившейся структурой.

В некоторых приложениях для каждого пользователя заводится отдельная таблица, главным образом для того, чтобы реализовать механизм обеспечения безопасности на уровне строки. Представление, аналогичное рассмотренному выше, могло бы дать тот же результат с помощью всего одной таблицы, а чем меньше открытых таблиц, тем выше производительность. Во многих проектах с открытым исходным кодом, которые применяются в системах массового хостинга, количество таблиц исчисляется миллионами, поэтому такой подход приносит несомненную выгоду. Вот пример из гипотетической базы данных для обслуживания блогов:

```
CREATE VIEW blog_posts_for_user_1234 AS
   SELECT * FROM blog_posts WHERE user_id = 1234
   WITH CHECK OPTION;
```

С помощью представлений можно реализовать и привилегии на уровне столбцов без накладных расходов на физическое создание таких привилегий, которые могут оказаться весьма ощутимыми. К тому же наличие привилегий на уровне столбцов препятствует кэшированию запроса. Представление позволяет ограничить доступ к интересующим вас столбцам, избежав такого рода проблем:

```
CREATE VIEW public.employeeinfo AS
   SELECT firstname, lastname – но не socialsecuritynumber
   FROM private.employeeinfo;
GRANT SELECT ON public.* TO public_user;
```

Иногда хороший эффект дают псевдовременные представления. Невозможно создать по-настоящему временное представление, которое существует только в текущем соединении, но можно выбрать для таких представлений специальные имена, быть может, создавать их в особой базе данных, чтобы знать, что впоследствии их можно спокойно удалять. Затем такое представление используется во фразе `FROM` точно так же, как подзапрос. Теоретически оба подхода эквивалентны, но в MySQL для работы с представлениями имеется специальный код, так что временное представление может оказаться более эффективным. Рассмотрим пример[1]:

```
-- Пусть 1234 – результат, возвращенный CONNECTION_ID()
CREATE VIEW temp.cost_per_day_1234 AS
   SELECT DATE(ts) AS day, sum(cost) AS cost
   FROM logs.cost
   GROUP BY day;
```

[1] Cost per day – расходы за день; sales per day – продажи за день. – *Прим. ред.*

```
SELECT c.day, c.cost, s.sales
FROM temp.cost_per_day_1234 AS c
   INNER JOIN sales.sales_per_day AS s USING(day);

DROP VIEW temp.cost_per_day_1234;
```

Отметим, что мы воспользовались идентификатором соединения (1234) как уникальным суффиксом, который позволяет избежать конфликта имен. При таком соглашении проще производить очистку в случае, когда приложение завершается аварийно и не удаляет временное представление. Дополнительную информацию об этой технике см. в разделе «Отсутствующие временные таблицы» на стр. 488.

Производительность представлений, которые реализуются методом TEMPTABLE, может оказаться очень низкой (хотя все равно лучше, чем у эквивалентного запроса без использования представлений). MySQL выполняет их с помощью рекурсивного шага на стадии оптимизации внешнего запроса, еще до того, как он полностью оптимизирован, поэтому ряд оптимизаций, к которым вы могли привыкнуть по опыту работы с другими СУБД, не применяется. Так, при обработке строящего временную таблицу запроса условия WHERE не «опускается вниз» с уровня внешнего запроса на уровень представления, а индексы над временными таблицами не строятся. Рассмотрим пример, в котором используется все то же представление temp.cost_per_day_1234:

```
mysql> SELECT c.day, c.cost, s.sales
    -> FROM temp.cost_per_day_1234 AS c
    ->    INNER JOIN sales.sales_per_day AS s USING(day)
    ->    WHERE day BETWEEN '2007-01-01' AND '2007-01-31';
```

В данном случае сервер выполняет запрос, по которому построено представление, и помещает результат во временную таблицу, а затем соединяет с ней таблицу sales_per_day. Встречающееся во фразе WHERE ограничение BETWEEN не «проталкивается» в представление, поэтому в результирующем наборе представления окажутся все присутствующие в таблице даты, а не только для указанного месяца. Ко всему прочему, над временной таблицей нет индексов. В данном случае это не составляет проблемы: сервер ставит временную таблицу первой, поэтому для выполнения соединения можно воспользоваться индексом таблицы sales_per_day. Но если бы мы группировали два таких представления, то никаких индексов, позволяющих оптимизировать соединение, не существовало бы.

Всякий раз, пытаясь использовать представления для повышения производительности, выполняйте эталонное тестирование или хотя бы детальное профилирование. Даже методу MERGE присущи накладные расходы, поэтому трудно заранее сказать, как представление отразится на производительности. Если производительность важна, не ограничивайтесь гипотезами – меряйте и еще раз меряйте.

С представлениями сопряжены некоторые проблемы, характерные не только для MySQL. Представление может создать у разработчика иллюзию простоты, скрыв истинную сложность. Разработчик, не понимающий, что таится за этой кажущейся простотой, возможно, не увидит ничего плохого в повторении запросов к объекту, выглядящему, как обычная таблица, а на самом деле являющемуся сложным представлением. Нам встречались такие обманчиво элементарные запросы, для которых команда `EXPLAIN` выводила сотни строк, потому что одна из «таблиц» на деле оказывалась представлением, ссылающимся на множество других таблиц и представлений.

Ограничения представлений

MySQL не поддерживает материализованные представления, с которыми вы, возможно, знакомы по работе с другими СУБД. В общем случае *материализованное представление* хранит результаты выполнения в невидимой таблице, которую периодически обновляет по исходным данным. MySQL также не поддерживает индексированные представления. Впрочем, материализованные и индексированные представления можно эмулировать, создав кэш или сводные таблицы, а в MySQL 5.1 для их периодического перестроения можно воспользоваться событиями.

Кроме того, в реализации представлений MySQL есть ряд неприятных особенностей. Самая серьезная из них состоит в том, что данная СУБД не сохраняет исходный SQL-код представления, поэтому, если вы попытаетесь изменить представление, надеясь выполнить команды `SHOW CREATE VIEW`, а затем отредактировать результат, то столкнетесь с неприятным сюрпризом. Запрос будет представлен во внутреннем каноническом виде со всеми кавычками, но без какого бы то ни было форматирования, отступов и комментариев.

Если потребуется отредактировать представление, а исходный красиво отформатированный код утрачен, то его можно найти в последней строке *frm*-файла, соответствующего данному представлению. Если у вас есть привилегия `FILE`, а *frm*-файл доступен для чтения всем пользователям, то можно даже загрузить его содержимое с помощью SQL-функции `LOAD_FILE()`. С помощью несложных манипуляций со строками (спасибо Роланду Боумену за изобретательность) можно полностью восстановить исходный код:

```
mysql> SELECT
    ->    REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(
    ->    REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(
    ->      SUBSTRING_INDEX(LOAD_FILE('/var/lib/mysql/world/Oceania.frm'),
    ->      '\nsource=', -1),
    ->    '\\_','\_'), '\\%','\%'), '\\\\','\\'), '\\Z','\Z'), '\\t','\t'),
    ->    '\\r','\r'), '\\n','\n'), '\\b','\b'), '\\"','\"'), '\\'','\''),
    ->    '\\0','\0')
    -> AS source;
```

```
+-------------------------------------------------------------------------+
| source                                                                  |
+-------------------------------------------------------------------------+
| SELECT * FROM Country WHERE continent = 'Oceania'
  WITH CHECK OPTION
|
+-------------------------------------------------------------------------+
```

Кодировки и схемы упорядочения

Кодировкой (*character set*) называется отображение множества двоичных кодов на некоторое множество символов; можно считать, что это способ представления конкретного алфавита в виде комбинаций битов. *Схема упорядочения* (*collation*) – это набор правил сортировки для конкретной кодировки. Начиная с версии MySQL 4.1, с каждым символьным значением могут быть связаны кодировка и схема упорядочения[1]. Поддержка кодировок и схем упорядочения в MySQL реализована на уровне лучших мировых стандартов, но за нее приходится расплачиваться дополнительной сложностью, а иногда и падением производительности.

В этом разделе мы рассмотрим те параметры и функциональность, которых в большинстве случаев достаточно. Желающие познакомиться с подробностями могут обратиться к руководству по MySQL.

Использование кодировок в MySQL

С каждой кодировкой может быть связано несколько схем упорядочения, и ровно одна из них подразумевается по умолчанию. Схема упорядочения принадлежит конкретной кодировке, и ни с какой другой ее использовать нельзя. Кодировка и схема упорядочения применяются совместно, поэтому начиная с текущего момента мы будем называть эту пару просто «кодировкой».

В MySQL есть множество параметров для управления кодировками. Эти параметры легко спутать с самими кодировками, поэтому всегда помните: только символьные значения могут «иметь» кодировку. Во всех остальных случаях речь идет всего лишь о параметре, который говорит, какую кодировку использовать для сравнения и других операций. Символьным значением может быть значение, хранящееся в столбце, литерал в запросе, результат выражения, пользовательская переменная и т. д.

Параметры MySQL можно разбить на две категории: умолчания при создании объектов и параметры, управляющие взаимодействием между клиентом и сервером.

---

[1] В MySQL 4.0 и более ранних версиях использовалась глобальная настройка для всего сервера и можно было выбрать одну из нескольких 8-разрядных кодировок.

Умолчания при создании объектов

В MySQL сервер, каждая база данных и каждая таблица имеют свою кодировку и схему упорядочения по умолчанию. Они образуют иерархию умолчаний, на основе которой выбирается кодировка вновь создаваемого столбца. Это, в свою очередь, служит указанием серверу на то, в какой кодировке хранить значения в этом столбце.

На каждом уровне иерархии кодировку можно либо определить явно, либо позволить серверу использовать подходящие умолчания.

- При создании базы данных кодировка наследуется от определенного на уровне сервера параметра `character_set_server`.
- При создании таблицы кодировка наследуется от базы данных.
- При создании столбца кодировка наследуется от таблицы.

Не забывайте, что значения хранятся только в таблицах, поэтому на более высоких уровнях иерархии определены всего лишь умолчания. Кодировка по умолчанию для таблицы никак не отражается на том, как хранятся значения в этой таблице; это лишь способ сообщить MySQL, какую кодировку следует использовать при создании нового столбца, если она не указана явно.

Параметры взаимодействия между клиентом и сервером

Клиент и сервер могут посылать друг другу данные в разных кодировках. Сервер выполняет преобразование по мере необходимости.

- Сервер предполагает, что клиент посылает команды в кодировке, заданной параметром `character_set_client`.
- Получив команду от клиента, сервер преобразует ее в кодировку, заданную параметром `character_set_connection`. Этот же параметр управляет преобразованием чисел в строки.
- Результаты и сообщения об ошибках сервер преобразует в кодировку, заданную параметром `character_set_result`.

Весь этот процесс представлен на рис. 5.5.

Изменить эти параметры позволяют команды `SET NAMES` и `SET CHARACTER SET`. Отметим, однако, что и та, и другая влияют *только на параметры сервера*. Клиентское приложение и API клиента также нужно правильно настроить, чтобы не возникало проблем при взаимодействии с сервером.

Предположим, что клиент открывает соединение с кодировкой latin1 (принимается по умолчанию, если не была изменена с помощью функции `mysql_options()`), а затем выполняет команду `SET NAMES utf8`, сообщая серверу о том, что он будет посылать данные в кодировке UTF-8. В результате получилось несоответствие кодировок, которое может привести к ошибкам и даже поставить под угрозу безопасность. Необходимо установить кодировку клиента и вызывать функцию `mysql_real_escape_`

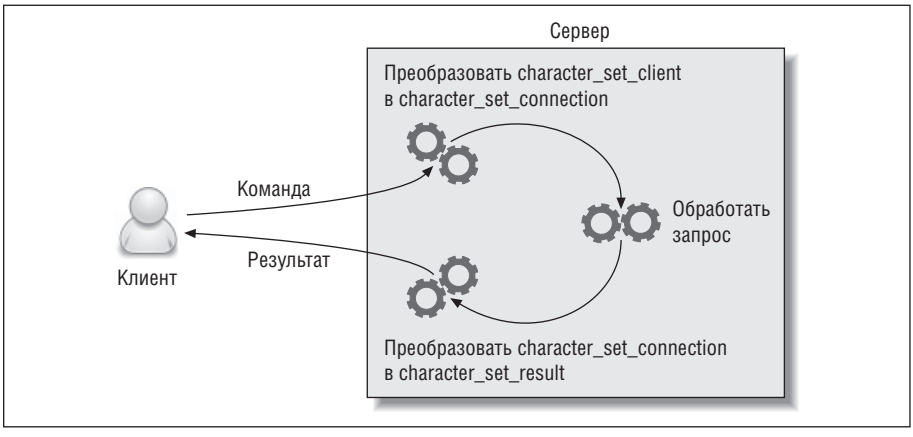


Рис. 5.5. *Кодировки клиента и сервера*

`string()` для экранирования значений. В PHP для изменения кодировки клиента можно воспользоваться функцией `mysql_set_charset()`.

Как MySQL сравнивает значения

При сравнении двух значений в разных кодировках MySQL должен сначала привести их к общей кодировке. Если кодировки несовместимы, то возникнет ошибка «ERROR 1267 (HY000): Illegal mix of collations». В таком случае следует воспользоваться функцией `CONVERT()` и явно преобразовать одно значение в кодировку, совместимую с кодировкой другого значения. Начиная с версии MySQL 5.0, такое преобразование часто выполняется неявно, поэтому вышеупомянутая ошибка более характерна для MySQL 4.1.

MySQL также назначает значениям характеристику, которая называется *приводимостью* (*coercibility*). Она определяет приоритет кодировки значения и влияет на то, к какому значению MySQL будет применять неявное преобразование. Для отладки ошибок, связанных с кодировками и схемами упорядочения, можно использовать функции `CHARSET()`, `COLLATION()` и `COERCIBILITY()`.

Чтобы задать кодировку и/или схему упорядочения литеральных значений в SQL-командах, можно использовать *вступительные элементы* (*introducers*) и *фразы COLLATE*, например:

```
mysql> SELECT _utf8 'hello world' COLLATE utf8_bin;
+--------------------------------------+
| _utf8 'hello world' COLLATE utf8_bin |
+--------------------------------------+
| hello world                          |
+--------------------------------------+
```

Поведение в особых случаях

Поведение кодировок таит в себе несколько сюрпризов. Ниже описаны ситуации, о которых стоит помнить.

Магический параметр `character_set_database`

По умолчанию параметр `character_set_database` совпадает с кодировкой базы данных, принимаемой по умолчанию. Когда изменяется база данных по умолчанию, изменяется и этот параметр. При подключении к серверу, для которого не определена база данных по умолчанию, он принимает значение `character_set_server`.

`LOAD DATA INFILE`

Команда `LOAD DATA INFILE` интерпретирует входные данные в соответствии с текущим значением параметра `character_set_database`. Некоторые версии MySQL принимают необязательную фразу `CHARACTER SET` в данной команде, но полагаться на это не стоит. Мы полагаем, что надежнее всего выбрать нужную базу данных с помощью команды `USE`, затем задать кодировку с помощью команды `SET NAMES`, а уже потом загружать данные. MySQL считает, что все загружаемые данные записаны в одной и той же кодировке, вне зависимости от того, какая кодировка задана для столбцов, в которые данные загружаются.

`SELECT INTO OUTFILE`

Команда `SELECT INTO OUTFILE` выводит данные без перекодировки. В настоящее время единственный способ задать кодировку выводимых данных – обернуть каждый столбец функцией `CONVERT()`.

Внутренние escape-последовательности

MySQL интерпретирует escape-последовательности в командах в соответствии с параметром `character_set_client`, даже когда имеется вступительный элемент или фраза `COLLATE`. Это объясняется тем, что escape-последовательности в литералах интерпретирует синтаксический анализатор, а он ничего не знает о схемах упорядочения. С точки зрения анализатора, вступительный элемент – не команда, а просто лексема.

Выбор кодировки и схемы упорядочения

Начиная с версии 4.1, MySQL поддерживает довольно много кодировок и схем упорядочения, в том числе и Unicode с многобайтовыми символами UTF-8 (MySQL поддерживает трехбайтовое подмножество UTF-8, достаточное для представления практически всех символов в большинстве языков). Узнать, какие кодировки поддерживаются, позволяют команды `SHOW CHARACTER SET` и `SHOW COLLATION`.

При выборе схемы упорядочения обычно исходят из того, как сортировать буквы: с учетом регистра, без учета регистра или в соответствии с двоичным кодом. Соответственно, имена схем упорядочения, как пра-

вило, заканчиваются на _cs, _ci или _bin, чтобы проще было определить, к какой их трех групп они относятся.

Когда кодировка задается явно, необязательно указывать и имя кодировки, и имя схемы упорядочения. Если одно или оба имени опущены, MySQL подставит недостающее по умолчанию. В табл. 5.2 показано, как MySQL принимает решение о выборе кодировки и схемы упорядочения.

Таблица 5.2. Как MySQL определяет кодировку и схему упорядочения по умолчанию

Если заданы	Выбирается кодировка	Выбирается схема упорядочения
Кодировка и схема упорядочения	Заданная	Заданная
Только кодировка	Заданная	Принимаемая по умолчанию для заданной кодировки
Только схема упорядочения	Та, к которой относится схема упорядочения	Заданная
Ни то, ни другое	Применимое умолчание	Применимое умолчание

Ниже показаны команды для создания базы данных, таблицы и столбца с явно заданными кодировкой и схемой упорядочения:

```
CREATE DATABASE d CHARSET latin1;
CREATE TABLE d.t(
   col1 CHAR(1),
```

Будьте проще

Смесь разных кодировок в одной базе данных может привести к хаосу. Несовместимые кодировки служат источником разнообразных ошибок. Все может быть хорошо, пока в данных не встретится какой-то конкретный символ, а потом начинаются сложности при выполнении тех или иных операций (например, при соединении таблиц). Для разрешения проблемы приходится либо выполнять команду ALTER TABLE, чтобы преобразовать столбцы в совместимые кодировки, либо выполнять приведение к нужной кодировке и схеме упорядочения в каждой SQL-команде.

Во избежание всех этих трудностей рекомендуем задать разумные умолчания на уровне сервера и, быть может, на уровне базы данных. А затем в исключительных случаях задавать кодировку на уровне столбца.

```
   col2 CHAR(1) CHARSET utf8,
   col3 CHAR(1) COLLATE latin1_bin
) DEFAULT CHARSET=cp1251;
```

В получившейся таблице для столбцов будут действовать следующие схемы упорядочения:

```
mysql> SHOW FULL COLUMNS FROM d.t;
+-------+---------+-------------------+
| Field | Type    | Collation         |
+-------+---------+-------------------+
| col1  | char(1) | cp1251_general_ci |
| col2  | char(1) | utf8_general_ci   |
| col3  | char(1) | latin1_bin        |
+-------+---------+-------------------+
```

Как кодировка и схема упорядочения отражаются на запросах

При использовании некоторых кодировок может возрасти потребление ресурсов процессора, памяти и места на диске. Возможно, даже не удастся воспользоваться индексами. Поэтому к выбору кодировки и схемы упорядочения следует подходить очень тщательно.

Преобразование из одной кодировки или схемы упорядочения в другую может повлечь за собой издержки при выполнении некоторых операций. Так, по столбцу `title` таблицы `sakila.film` построен индекс, способный ускорить выполнение запросов с фразой `ORDER BY`:

```
mysql> EXPLAIN SELECT title, release_year FROM sakila.film ORDER BY title\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: index
possible_keys: NULL
          key: idx_title
      key_len: 767
          ref: NULL
         rows: 953
        Extra:
```

Однако сервер может использовать этот индекс для сортировки, только если он обработан в соответствии с той же схемой упорядочения, что указана в запросе. При построении индекса используется схема упорядочения столбца, в данном случае `utf8_general_ci`. Если требуется отсортировать результаты, исходя из другой схемы упорядочения, то сервер прибегнет к обычной сортировке (filesort):

```
mysql> EXPLAIN SELECT title, release_year
    -> FROM sakila.film ORDER BY title COLLATE utf8_bin\G
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 953
        Extra: Using filesort
```

MySQL должна не только адаптироваться к кодировке, заданной по умолчанию для соединения, а также ко всем параметрам, явно указанным в запросах, но и выполнять перекодирование, чтобы можно было сравнивать значения, записанные в разных кодировках. Например, при соединении двух таблиц по столбцам, имеющим разные кодировки, MySQL вынуждена перекодировать один из них. В результате может получиться так, что не удастся воспользоваться индексом, поскольку перекодировку можно уподобить функции, обертывающей столбец.

В случае многобайтовой кодировки UTF-8 для хранения символов отводится различное число байтов (от одного до трех). Для выполнения многих операций со строками MySQL применяет буферы фиксированного размера, поэтому приходится выделять память, исходя из максимально возможной длины строки. Например, для хранения значения типа `CHAR(10)` в кодировке UTF-8 потребуется 30 байтов, даже если фактическая строка не содержит так называемых широких символов. При хранении полей переменной длины (`VARCHAR`, `TEXT`) на диске эта проблема не возникает, но во временных таблицах в памяти, которые используются для обработки запросов и сортировки результатов, место всегда выделяется по максимальной длине.

В многобайтовых кодировках символ и байт уже не одно и то же. Поэтому в MySQL имеется две функции: `LENGTH()` и `CHAR_LENGTH()`, которые в случае многобайтовой кодировки возвращают разные результаты. При работе с многобайтовой кодировкой для подсчета символов пользуйтесь функцией `CHAR_LENGTH()` (например, при вычислении параметров функции `SUBSTRING()`). То же предостережение действует и для многобайтовых кодировок в языках программирования приложений.

Еще один возможный сюрприз – ограничения на индексы. Если индексируется столбец в кодировке UTF-8, то MySQL вынужден предполагать, что каждый символ может занимать до трех байтов, поэтому обычные ограничения на длину уменьшаются в три раза.

```
mysql> CREATE TABLE big_string(str VARCHAR(500), KEY(str)) DEFAULT
CHARSET=utf8;
Query OK, 0 rows affected, 1 warning (0.06 sec)
mysql> SHOW WARNINGS;
```

```
+---------+------+-------------------------------------------------------+
| Level   | Code | Message                                               |
+---------+------+-------------------------------------------------------+
| Warning | 1071 | Specified key was too long; max key length is 999 bytes |
+---------+------+-------------------------------------------------------+
```

Отметим, что MySQL автоматически укорачивает ключ индекса до 333 символов:

```
mysql> SHOW CREATE TABLE big_string\G
*************************** 1. row ***************************
       Table: big_string
Create Table: CREATE TABLE `big_string` (
   `str` varchar(500) default NULL,
   KEY `str` (`str`(333))
) ENGINE=MyISAM DEFAULT CHARSET=utf8
```

Если вы не обратите внимания на предупреждение и посмотрите на определение таблицы, то увидите, что индекс создан лишь по части столбца. При этом могут возникнуть нежелательные побочные эффекты, в частности, невозможность использования покрывающих индексов.

Иногда рекомендуют глобально использовать кодировку UTF-8, чтобы «упростить себе жизнь». Однако с точки зрения производительности это далеко не всегда хорошо. Многим приложениям кодировка UTF-8 совсем ни к чему, а при ее использовании места на диске может потребоваться гораздо больше (в зависимости от характера данных).

Принимая решение о выборе кодировки, нужно подумать о том, какие данные вы собираетесь хранить. Например, если речь идет преимущественно об англоязычном тексте, то кодировка UTF-8 практически не повлияет на используемое дисковое пространство, так как большинство символов английского языка кодируются одним байтом. Но для языков с алфавитом, отличным от латиницы, например русского или арабского, разница может оказаться очень заметной. Если приложение должно хранить *только* арабские символы, то можно воспользоваться кодировкой `cp1256`, в которой все знаки арабского алфавита кодируются одним байтом. Однако, если требуется представлять тексты на разных языках и вы возьмете кодировку UTF-8, то для тех же самых арабских символов потребуется больше места. Аналогично преобразование столбца из национальной кодировки в UTF-8 можно ощутимо увеличить объем потребляемого места на диске. В случае InnoDB размер данных может увеличиться настолько, что значение перестанет помещаться на одной странице и придется задействовать внешнюю память, а это приводит к непроизводительной растрате места на диске и к фрагментации. Дополнительную информацию по этому вопросу см. в разделе «Оптимизация доступа к полям типа BLOB и TEXT» на стр. 372.

Иногда вообще не нужно использовать кодировку. Кодировки полезны прежде всего для сравнения с учетом регистра, сортировки и тех операций со строками, в которых нужно распознавать границы между сим-

волами, например SUBSTRING(). Если не требуется, чтобы сервер баз данных мог обрабатывать символьные значения, то можно хранить все, включая данные в кодировке UTF-8, в столбцах типа BINARY. В этом случае рекомендуется добавить еще специальный столбец, в котором будет содержаться информация о том, в какой кодировке представлена информация. Хотя подобный подход используется уже давно, он требует большей внимательности. Если забыть о том, что байт и символ – не одно и то же, то могут возникнуть трудные для отладки ошибки, например при использовании функций SUBSTRING() и LENGTH(). Мы рекомендуем по возможности избегать такой практики.

Полнотекстовый поиск

В большинстве типичных запросов присутствует фраза WHERE, в которой значения сравниваются на равенство, выделяются диапазоны строк и т. д. Но иногда нужно искать по ключевому слову, и в этом случае поиск должен быть основан на релевантности, а не простом сравнении строк. Для этой цели и предназначены системы полнотекстового поиска.

Для полнотекстового поиска требуется специальный синтаксис запроса. Индекс необязателен, но при его наличии поиск выполняется быстрее. Полнотекстовые индексы имеют специальную структуру, ускоряющую поиск документов, содержащих заданные ключевые слова.

По крайней мере, с одним типом систем полнотекстового поиска вы точно знакомы, даже если не осознавали этого. Речь идет о поисковых системах в Интернете. Хотя работают они с гигантскими объемами данных и обычно не используют реляционные СУБД для их хранения, тем не менее, основные принципы схожи.

В MySQL поддержка полнотекстового поиска реализована только в подсистеме MyISAM. Она позволяет искать в символьных столбцах (типа CHAR, VARCHAR и TEXT) и обеспечивает как булевский поиск, так и поиск на естественном языке. В реализации полнотекстового поиска есть целый ряд ограничений[1], кроме того она достаточно сложна, но все же находит широкое применение, так как является частью сервера и пригодна для многих приложений. В настоящем разделе мы познакомимся с тем, как эту систему использовать и как проектировать полнотекстовый поиск с учетом производительности.

Полнотекстовый индекс MyISAM оперирует *полнотекстовым набором*, который составлен из одного или нескольких столбцов какой-либо таблицы. По сути дела MySQL конкатенирует все столбцы, входящие в набор, и индексирует результат как одну длинную текстовую строку.

[1] Возможно, вы придете к выводу, что ограничения на полнотекстовый поиск в MySQL делают его практическое использование в вашей программе бессмысленным. В приложении C мы обсудим внешнюю поисковую систему Sphinx.

Полнотекстовый индекс MyISAM представляет собой специальный вид B-дерева с двумя уровнями. На первом уровне находятся ключевые слова. А на втором уровне для каждого ключевого слова располагается список ассоциированных с ним *указателей на документы*, ведущих на полнотекстовые наборы, в которых встречается данное ключевое слово. В индекс не добавляется каждое слово в наборе. Часть слов отбрасывается, а именно:

- Слова, входящие в список *стоп-слов*, то есть «шум». По умолчанию перечень стоп-слов построен исходя из традиционного словоупотребления в английском языке, но параметр `ft_stopword_file` позволяет заменить его списком, взятым из внешнего файла.
- Слово игнорируется, если оно короче `ft_min_word_len` символов или длиннее `ft_max_word_len` символов.

В полнотекстовом индексе не хранится информация о том, в каком столбце набора находится ключевое слово, поэтому если необходимо искать по различным комбинациям столбцов, то придется создать несколько индексов.

Это также означает, что во фразе `MATCH AGAINST` нельзя сказать, будто слова, встречающиеся в одном столбце, важнее слов, встречающихся во всех остальных. А это типичное требование при построении систем поиска по веб-сайтам. Например, иногда желательно, чтобы документы, в которых ключевое слово встречается в заголовке, оказывались в списке результатов раньше. Если для вас это существенно, то придется писать более сложные запросы (ниже мы приведем пример).

Полнотекстовые запросы на естественном языке

Поисковый запрос на естественном языке определяет релевантность каждого документа исходному запросу. Релевантность вычисляется исходя из количества совпавших слов и частоты их вхождения в документ. Чем реже слово встречается в индексе, тем более релевантным делается совпадение. И наоборот, слова, употребляемые очень часто, вообще не стоит искать. При полнотекстовом поиске на естественном языке исключаются вхождения, которые встречаются более чем в 50% строк таблицы, даже если их нет в списке стоп-слов[1].

Синтаксис полнотекстового поиска несколько отличается от запросов других типов. Чтобы MySQL выполнил полнотекстовый поиск, во фразе `WHERE` должен присутствовать предикат `MATCH AGAINST`. Рассмотрим пример. В демонстрационной базе Sakila по столбцам `title` и `description` таблицы `film_text` построен полнотекстовый индекс.

[1] При тестировании часто допускают типичную ошибку: помещают несколько строк с тестовыми данными в полнотекстовый индекс, а потом обнаруживают, что поиск ничего не находит. Проблема в том, что каждое слово встречается более чем в половине строк таблицы.

```
mysql> SHOW INDEX FROM sakila.film_text;
+-----------+-----------------------+-------------+------------+
| Table     |              Key_name | Column_name | Index_type |
+-----------+-----------------------+-------------+------------+
| ...
| film_text | idx_title_description | title       | FULLTEXT   |
| film_text | idx_title_description | description | FULLTEXT   |
+-----------+-----------------------+-------------+------------+
```

Вот пример полнотекстового запроса на естественном языке:

```
mysql> SELECT film_id, title, RIGHT(description, 25),
    ->    MATCH(title, description) AGAINST('factory casualties') AS
relevance
    -> FROM sakila.film_text
    -> WHERE MATCH(title, description) AGAINST('factory casualties');
+---------+-----------------------+--------------------------+-----------------+
| film_id | title                 | RIGHT(description, 25)   | relevance       |
+---------+-----------------------+--------------------------+-----------------+
| 831     | SPIRITED CASUALTIES   | a Car in A Baloon Factory | 8.4692449569702 |
| 126     | CASUALTIES ENCINO     | Face a Boy in A Monastery | 5.2615661621094 |
| 193     | CROSSROADS CASUALTIES | a Composer in The Outback | 5.2072987556458 |
| 369     | GOODFELLAS SALUTE     | a Cow in A Baloon Factory | 3.1522686481476 |
| 451     | IGBY MAKER            | a Dog in A Baloon Factory | 3.1522686481476 |
```

Чтобы выполнить полнотекстовый поиск, MySQL разбил поисковую строку на слова и сопоставил их с содержимым полей `title` и `description`, которые были объединены в один полнотекстовый набор на этапе построения индекса. Отметим, что только одна строка содержит оба этих слова и что три результата, содержащие слово «casualties» (таких всего три во всей таблице) перечислены в начале. Это объясняется тем, что в индексе результаты отсортированы по убыванию релевантности.

В отличие от обычных запросов, выдача полнотекстового поиска автоматически сортируется по релевантности. MySQL не умеет использовать для сортировки индекс, если запрос полнотекстовый. Поэтому, если вы хотите избежать сортировки (filesort), не включайте в запрос фразу `ORDER BY`.

Как легко видеть из примера, функция `MATCH()` возвращает релевантность в виде числа с плавающей точкой. Этим можно воспользоваться для фильтрации результатов по релевантности или для показа релевантности в пользовательском интерфейсе. Функцию `MATCH()` можно употреблять более одного раза, не опасаясь дополнительных издержек; MySQL понимает, что речь идет об одном и том же, и выполняет операцию только один раз. Однако если поместить вызов `MATCH()` во фразу `ORDER BY`, то MySQL для упорядочения результатов прибегнет к сортировке (filesort).

Столбцы во фразе `MATCH()` следует перечислять точно в том порядке, в котором они были заданы при построении полнотекстового индекса. В противном случае MySQL не сможет воспользоваться индексом. Про-

блема в том, что индекс не содержит сведений, в каком столбце встретилось ключевое слово.

Как мы уже отмечали, это также означает, что при полнотекстовом поиске нельзя сказать, что ключевое слово должно встречаться в конкретном столбце. Однако существует обходной путь: можно провести нестандартную сортировку с помощью нескольких полнотекстовых индексов по различным комбинациям столбцов, чтобы добиться необходимого ранжирования. Предположим, что столбец `title` считается более приоритетным. Тогда можно добавить еще один индекс по этому столбцу:

```
mysql> ALTER TABLE film_text ADD FULLTEXT KEY(title) ;
```

Теперь для целей ранжирования можно удвоить приоритет `title`:

```
mysql> SELECT film_id, RIGHT(description, 25),
    -> ROUND(MATCH(title, description) AGAINST('factory casualties'), 3)
    ->    AS full_rel,
    -> ROUND(MATCH(title) AGAINST('factory casualties'), 3) AS title_rel
    -> FROM sakila.film_text
    -> WHERE MATCH(title, description) AGAINST('factory casualties')
    -> ORDER BY (2 * MATCH(title) AGAINST('factory casualties'))
    ->    + MATCH(title, description) AGAINST('factory casualties') DESC;
+---------+--------------------------+----------+-----------+
| film_id | RIGHT(description, 25)   | full_rel | title_rel |
+---------+-------------- ------------+----------+-----------+
|     831 | a Car in A Baloon Factory |    8.469 |     5.676 |
|     126 | Face a Boy in A Monastery |    5.262 |     5.676 |
|     299 | jack in The Sahara Desert |    3.056 |     6.751 |
|     193 | a Composer in The Outback |    5.207 |     5.676 |
|     369 | d Cow in A Baloon Factory |    3.152 |     0.000 |
|     451 | a Dog in A Baloon Factory |    3.152 |     0.000 |
|     595 | a Cat in A Baloon Factory |    3.152 |     0.000 |
|     649 | nizer in A Baloon Factory |    3.152 |     0.000 |
```

Однако такой подход обычно оказывается неэффективным, так как приводит к сортировке (filesort).

Булевский полнотекстовый поиск

При булевском поиске в самом запросе задается относительная релевантность каждого слова. Как и раньше, для фильтрации шума используется список стоп-слов, однако требование о том, чтобы слова были не короче, чем `ft_min_word_len` символов, и не длиннее, чем `ft_max_word_len`, не предъявляется. Результаты никак не сортируются.

При конструировании булевского запроса можно использовать префиксы для относительного ранжирования каждого слова в поисковой строке. Наиболее употребительные модификаторы приведены в табл. 5.3.

Можно использовать и другие операторы, например скобки для группировки. Таким образом конструируются сложные запросы.

Таблица 5.3. Наиболее употребительные модификаторы для булевского полнотекстового поиска

Пример	Назначение
`dinosaur`	Строки, содержащие слово «dinosaur», имеют больший ранг
`~dinosaur`	Строки, содержащие слово «dinosaur», имеют меньший ранг
`+dinosaur`	Строка *должна содержать* слово «dinosaur»
`-dinosaur`	В строке *должно отуствовать* слово «dinosaur»
`dino*`	Строки, содержащие слова, которые начинаются с «dino», имеют больший ранг

В качестве примера еще раз рассмотрим таблицу `sakila.film_text` и найдем в ней фильмы, в описании которых содержатся слова «factory» и «casualties». Как мы уже видели, поиск на естественном языке возвращает результаты, где присутствуют одно или оба слова. Но если воспользоваться булевским поиском, то можно потребовать, чтобы обязательно встречались оба слова:

```
mysql> SELECT film_id, title, RIGHT(description, 25)
    -> FROM sakila.film_text
    -> WHERE MATCH(title, description)
    ->    AGAINST('+factory +casualties' IN BOOLEAN MODE);
+---------+--------------------+--------------------------+
| film_id | title              | RIGHT(description, 25)   |
+---------+--------------------+--------------------------+
|     831 | SPIRITED CASUALTIES | a Car in A Baloon Factory |
+---------+--------------------+--------------------------+
```

Можно также произвести *поиск фразы*, заключив несколько слов в кавычки: это означает, что должна встречаться в точности указанная фраза:

```
mysql> SELECT film_id, title, RIGHT(description, 25)
    -> FROM sakila.film_text
    -> WHERE MATCH(title, description)
    ->    AGAINST('"spirited casualties"' IN BOOLEAN MODE);
+---------+--------------------+--------------------------+
| film_id |              title |    RIGHT(description, 25) |
+---------+--------------------+--------------------------+
|     831 | SPIRITED CASUALTIES | a Car in A Baloon Factory |
+---------+--------------------+--------------------------+
```

Поиск фразы может выполняться довольно медленно. Один лишь поиск по полнотекстовому индексу не в состоянии дать ответ на такой запрос, поскольку индекс не содержит информации о том, как слова расположены друг относительно друга в исходном полнотекстовом наборе. Поэтому сервер должен анализировать сами строки.

Чтобы выполнить такой запрос, сервер сначала находит все документы, содержащие оба слова: «spirited» и «casualties». Затем он выбирает строки из найденных документов и проверяет вхождение фразы целиком. Поскольку на первом этапе используется полнотекстовый индекс, может сложиться впечатление, что поиск производится очень быстро, гораздо быстрее, чем с помощью эквивалетной операции `LIKE`. Это действительно так, если входящие во фразу слова не являются общеупотребительными, так что поиск по полнотекстовому индексу вернул не слишком много документов. Если же эти слова встречаются очень часто, то `LIKE` может оказаться намного быстрее, так как в этом случае строки читаются последовательно, а не в квазислучайном порядке индекса, а обращаться к полнотекстовому индексу вовсе не требуется.

Для булевского поиска полнотекстовый индекс фактически не нужен. Если такой индекс есть, он просматривается, в противном случае сканируется вся таблица. Можно даже применить булевский полнотекстовый поиск к столбцам из нескольких таблиц, например к результату соединения. Впрочем, во всех таких случаях процедура поиска будет выполняться медленно.

Изменения в полнотекстовом поиске в версии MySQL 5.1 и более поздних

В версии MySQL 5.1 полнотекстовый поиск претерпел несколько изменений. Это и улучшенная производительность, и возможность подключать внешние анализаторы, расширяющие встроенные возможности. Так, подключаемый анализатор может изменить способ индексирования. С его помощью можно разбивать текст на слова более гибко, чем по умолчанию (скажем, стало возможным определить, что “C++” – это одно слово), выполнять препроцессирование, индексировать документы различных форматов (например, PDF) или реализовать специальный алгоритм стемминга (выделения основы слова). Подключаемые модули могут также изменить способ выполнения поиска, например подвергнуть поисковые слова стеммингу.

Разработчики InnoDB в настоящее время работают над поддержкой полнотекстового поиска, но когда он будет готов, неизвестно.

Компромиссы полнотекстового поиска и обходные пути

Реализация полнотекстового поиска в MySQL содержит несколько архитектурных ограничений. Они могут препятствовать решению конкретных задач, но существует ряд способов эти ограничения обойти.

Например, полнотекстовое индексирование в MySQL поддерживает единственную форму ранжирования по релевантности: частоту. В индексе не хранится смещение слова от начала строки, поэтому учитывать близость при вычислении релевантности нельзя. Для многих применений (особенно, если объем данных невелик) это не страшно, но вас

такой подход может не устраивать, а MySQL не позволяет выбрать другой алгоритм ранжирования (она даже не хранит данных, необходимых для ранжирования по близости).

Вторая проблема – размер. Полнотекстовое индексирование в MySQL работает хорошо, если весь индекс помещается в памяти, но если это не так, то поиск может производиться очень медленно, особенно когда поля велики. Для поиска по фразе приемлемая производительность достигается, когда и данные, и индексы находятся в ОЗУ. Вставка, обновление и удаление строк из полнотекстового индекса обходятся очень дорого по сравнению с индексами других типов.

- Модификация фрагмента текста, содержащего 100 слов, требует не одной, а 100 операций с индексом.
- Скорость работы с другими типами индексов мало зависит от длины поля, а в случае полнотекстового индекса время индексирования текста из 3 слов и из 10000 слов отличается на несколько порядков.
- Полнотекстовые индексы в гораздо большей степени подвержены фрагментации, поэтому выполнять команду `OPTIMIZE TABLE` придется более часто.

Полнотекстовые индексы также влияют на оптимизацию запросов сервером. И выбор индекса, и обработка фраз `WHERE` и `ORDER BY` работают не так, как можно было бы ожидать.

- Если имеется полнотекстовый индекс, и в запросе присутствует фраза `MATCH AGAINST`, которая может его использовать, то MySQL задействует его при обработке запроса. Она не станет сравнивать полнотекстовый индекс с другими индексами, которые можно было бы применить при выполнении запроса. Некоторые из таких индексов могли бы ускорить выполнение, но MySQL даже рассматривать их не будет.
- Полнотекстовый индекс пригоден только для выполнения полнотекстового поиска. Все остальные указанные в запросе (в частности, во фразе `WHERE`) условия нужно будет применять после считывания строки из таблицы. Это отличается от поведения для индексов других типов, которые позволяют проверять сразу несколько условий во фразе `WHERE`.
- В полнотекстовом индексе не хранится сам индексируемый текст. Поэтому воспользоваться таким индексом как покрывающим не удастся.
- Полнотекстовые индексы можно использовать только для одного вида сортировки: по релевантности при поиске на естественном языке. Если нужно сортировать как-то иначе, MySQL прибегает к обычной сортировке (filesort).

Теперь посмотрим, как эти ограничения сказываются на запросах. Предположим, что имеется миллион документов, по которым построен обычный индекс по полю «автор документа» и полнотекстовый индекс

по содержимому. Требуется выполнить полнотекстовый поиск по содержимому, но только для документов, составленных автором 123. Такой запрос можно было бы записать следующим образом:

```
... WHERE MATCH(content) AGAINST ('High Performance MySQL')
    AND author = 123;
```

Однако он будет крайне неэффективен. Сначала MySQL осуществит полнотектовый поиск по всему миллиону документов, так как отдает предпочтение полнотекстовому индексу. Затем СУБД применит фразу `WHERE`, чтобы оставить только документы указанного автора, но при этом не сможет воспользоваться индексом по автору.

Одно из решений описанной проблемы – включить идентификатор автора в полнотекстовый индекс. Можно выбрать какой-нибудь префикс, появление которого в тексте маловероятно, дописать к нему сзади идентификатор автора и включить это «слово» в столбец `filters`, который обновляется независимо (возможно, с помощью триггера).

Затем можно расширить полнотекстовый индекс, включив в него столбец `filters`, и переписать запрос так:

```
... WHERE MATCH(content, filters)
    AGAINST ('High Performance MySQL +author_id_123' IN BOOLEAN MODE);
```

Это может оказаться эффективным, если идентификатор автора очень селективен, так как MySQL очень быстро сузит список документов, произведя в полнотекстовом индексе поиск по слову «author_id_123». Если же селективность невелика, то производительность может еще и ухудшиться. Поэтому не применяйте такой подход безоглядно.

Иногда полнотекстовые индексы можно использовать для поиска по ограничивающему прямоугольнику. Например, если вы хотите локализовать поиск некоторой прямоугольной областью на координатной плоскости (в случае географических приложений), то можно закодировать координаты в виде полнотекстового набора. Предположим, что в данной строке хранятся координаты X=123 и Y=456. Можно построить из них чередующуюся строку цифр, XY142536, и поместить ее в столбец, по которому построить полнотекстовый индекс. Теперь, если потребуется ограничить поиск, например, прямоугольником, для которого X изменяется от 10 до 199, а Y – от 400 до 499, то в запрос можно будет включить условие «+XY14*». Это может оказаться гораздо быстрее, чем фильтрация с помощью фразы `WHERE`.

Еще один прием, в котором иногда удается с пользой применить полнотекстовые индексы, особенно для разбиения списка результатов на страницы, состоит в том, чтобы выбрать перечень первичных ключей полнотекстовым запросом и кэшировать результаты. Когда приложение будет готово к выводу итогов поиска, оно может отправить еще один запрос, отбирающий нужные строки по их идентификаторам. В этот запрос можно включить более сложные критерии или соединения, при обработке которых допустимо использовать другие индексы.

Полнотекстовые индексы поддерживаются только подсистемой хранения MyISAM, но не впадайте в отчаяние, если вы работаете с InnoDB или какой-то другой подсистемой: существует способ сесть на ежа и при этом не уколоться. Самый распространенный метод – реплицировать таблицы на подчиненный сервер, где они будет храниться в формате MyISAM, и обслуживать полнотекстовые запросы там. Если вы не хотите обрабатывать часть запросов на другом сервере, то можете секционировать таблицу по вертикали, отделив текстовые столбцы от прочих данных.

Можно также продублировать некоторые столбцы в таблицу, над которой построен полнотекстовый индекс. Эта стратегия применена к таблице `sakila.film_text`, которая поддерживается в актуальном состоянии с помощью триггеров. Еще одна альтернатива – воспользоваться внешней системой полнотекстового поиска, например Lucene или Sphinx. Подробнее о системе Sphinx можно прочитать в приложении C.

Полнотекстовые запросы с фразой `GROUP BY` могут безнадежно «посадить» производительность, если полнотекстовый поиск находит много результатов; вслед за ним следует случайный ввод/вывод, а потом построение временной таблицы или сортировка (filesort) для последующей группировки. Поскольку такие запросы часто применяются с целью поиска максимальных элементов в каждой группе, то неплохой оптимизацией может стать поиск по выборке из таблицы вместо стремления к абсолютной точности. Например, поместите первые 1000 строк во временную таблицу, а затем ищите максимальные элементы в каждой группе по этой выборке.

Настройка и оптимизация полнотекстового поиска

Одно из самых важных условий повышения производительности – регулярное обслуживание полнотекстовых индексов. Из-за большого количества слов в типичных документах и структуры двухуровневого B-дерева полнотекстовые индексы страдают от фрагментации больше, чем обычные. Для устранения фрагментации старайтесь как можно чаще выполнять команду `OPTIMIZE TABLE`. Если сервер выполняет много операций ввода/вывода, то гораздо быстрее будет периодически удалять и заново создавать полнотекстовые индексы.

Чтобы сервер эффективно производил полнотекстовый поиск, нужно выделить достаточно памяти под буферы ключей, чтобы полнотекстовые индексы целиком помещались в ОЗУ, – в этом случае они функционируют значительно лучше. Можно использовать отдельные буферы ключей, чтобы другие индексы не вытесняли полнотекстовый из памяти. Дополнительную информацию о буферах ключей MyISAM см. в разделе «Кэш ключей MyISAM» на стр. 343.

Важно также создать качественный список стоп-слов. Список, предлагаемый по умолчанию, «заточен» под англоязычную прозу, но для других языков или узкоспециализированных, например технических, текстов не годится. Например, при индексировании документа, относя-

щегося к MySQL, имеет смысл сделать «mysql» стоп-словом, поскольку оно встречается слишком часто и потому бесполезно.

Зачастую удается повысить производительность за счет игнорирования коротких слов. Минимальная длина задается с помощью параметра `ft_min_word_len`. Если увеличить значение по умолчанию, то будет пропускаться больше слов, поэтому индекс станет короче и быстрее, правда, ценой снижения точности поиска. Не забывайте, впрочем, что для каких-то специальных целей могут быть значимы и очень короткие слова. Например, если искать по запросу «cd player» в документах о бытовой электронике, то при запрете на короткие слова может быть выдано много нерелевантных результатов. Пользователь вряд ли хочет видеть документы о MP3 и DVD-плеерах, но, если минимальная длина слова составляет 4 символа, то будет произведен поиск только по слову «player» и, стало быть, возвращены документы, касающиеся вообще всех плееров.

Задание списка стоп-слов и минимальной длины запроса может ускорить поиск за счет исключения некоторых слов из индекса, но качество поиска при этом пострадает. Подходящий баланс зависит от приложения. Если требуется хорошая производительность и высокое качество поиска, то придется настроить оба параметра. Было бы разумно интегрировать в приложение какой-нибудь механизм протоколирования, выполнить типичный поиск, нетипичный поиск, поиск, не возвращающий никаких результатов, и поиск, возвращающий очень много результатов, и посмотреть, что получается. Таким образом, можно получить полезную информацию о пользователях приложения и характере содержимого своих документов, а затем воспользоваться ей для повышения скорости и качества поиска.

Имейте в виду, что после изменения минимальной длины слова необходимо перестроить индекс командой `OPTIMIZE TABLE`, чтобы новое значение вступило в силу. Существует также параметр `ft_max_word_len`, страхующий от индексирования слишком длинных слов.

Если вы импортируете много данных в таблицы, содержащие проиндексированные текстовые столбцы, то перед началом импорта отключите полнотекстовые индексы командой `DISABLE KEYS`, а по завершении включите командой `ENABLE KEYS`. Обычно это дает существенный выигрыш во времени загрузки из-за высокой стоимости обновления индекса для каждой строки. А в качестве премии вы получаете еще и дефрагментированный индекс.

Если набор данных очень велик, то имеет смысл вручную распределить информацию по нескольким узлам и производить поиск на них параллельно. Это непростая задача, поэтому, возможно, лучше воспользоваться внешней системой полнотекстового поиска, например Lucene или Sphinx. Наш опыт показывает, что они могут обеспечить производительность на несколько порядков выше.

Ограничения внешнего ключа

В настоящее время InnoDB – основная подсистема хранения для MySQL, поддерживающая внешние ключи, так что у тех, кому они необходимы, выбор ограничен[1]. Компания MySQL AB обещала, что когда-нибудь сервер самостоятельно будет поддерживать внешние ключи способом, не зависимым от подсистемы хранения, но в обозримом будущем InnoDB остается единственной из основных подсистем, в которых эта функция реализована. Поэтому ее мы и будем рассматривать.

Внешние ключи обходятся не даром. Как правило, их наличие означает, что сервер должен заглядывать в другую таблицу при каждом изменении определенных данных. Хотя для ускорения операции InnoDB принудительно строит индекс, это вовсе не устраняет все негативные последствия подобных проверок. При этом может даже получиться очень большой индекс, имеющий крайне низкую селективность. Предположим, например, что в огромной таблице имеется столбец `status`, и требуется, чтобы он мог содержать только корректные значения, а таковых всего три. Необходимый для этого дополнительный индекс заметно увеличит общий размер таблицы – даже если размер самого столбца мал и, в особенности, если велика длина первичного ключа; при этом сам индекс не нужен ни для чего, кроме проверки внешнего ключа.

И все же в некоторых случаях внешние ключи могут повысить производительность. Если жизненно необходимо гарантировать, что данные в двух взаимосвязанных таблицах непротиворечивы, то эффективнее поручить проверку серверу, а не заниматься этим на уровне приложения. Внешние ключи полезны также для каскадного удаления и обновления, хотя эти операции выполняются построчно, то есть медленнее, чем запрос с обновлением нескольких таблиц или пакетная операция.

Из-за внешних ключей запрос может «распространяться» на другие таблицы, а это означает захват блокировок. Например, при попытке вставить строку в подчиненную таблицу, ограничение внешнего ключа заставит InnoDB проверить наличие соответствующего значения в главной таблице. При этом необходимо установить блокировку на строку главной таблицы, чтобы ее никто не удалил до завершения транзакции. Это может привести к неожиданному ожиданию блокировки и даже к взаимоблокировкам на таблицах, к которым вы напрямую не обращаетесь. Такого рода проблемы далеки от интуитивно очевидных и решать их очень трудно.

Иногда вместо внешних ключей можно использовать триггеры. Для таких операций, как каскадное обновление, внешние ключи работают быстрее триггеров, но если единственное назначение ключа – проверить ограничение, как в примере со столбцом `status`, то, возможно, эф-

[1] Их поддерживает также подсистема PBXT.

фективнее написать триггер, включив в него явный список допустимых значений (а можно просто воспользоваться типом данных ENUM).

Зачастую имеет смысл проверять ограничения в приложении, а не использовать для этой цели внешние ключи.

Объединенные таблицы и секционирование

Объединенные таблицы и секционирование – взаимосвязанные понятия, которые легко спутать. В MySQL *объединенные таблицы* (*merge tables*) – это способ объединить несколько таблиц типа MyISAM в одну «виртуальную таблицу». Это очень похоже на представление, в котором таблицы объединяются посредством фразы UNION. Объединенная таблица создается с помощью подсистемы хранения Merge, и, строго говоря, не является таблицей. Она больше напоминает контейнер для таблиц с похожими определениями.

Напротив, *секционированные таблицы* (*partitioned tables*) выглядят как обычные таблицы со специальным набором указаний, сообщающих MySQL, где нужно физически хранить строки. Откроем секрет: код, реализующий хранение секционированных таблиц, очень похож на код для объединенных таблиц! Фактически, на нижнем уровне каждая секция представляет собой отдельную таблицу со своими индексами, а вся секционированная таблица – это просто обертка вокруг набора объектов Handler. Секционированная таблица выглядит и ведет себя, как единая таблица, хотя на самом деле является совокупностью отдельных таблиц. Однако обратиться к таблицам-секциям напрямую невозможно, тогда как объединенные таблицы это позволяют.

Секционирование появилось начиная с версии MySQL 5.1, а объединенные таблицы существуют уже давно. У обоих механизмов одни и те же достоинства. Они позволяют:

- Отделить статические данные от изменяющихся
- Воспользоваться физической близостью взаимосвязанных данных для оптимизации запросов
- Проектировать таблицы так, чтобы запрос обращался к возможно меньшему объему данных
- Упростить обслуживание очень больших наборов данных (в этом вопросе объединенные таблицы обладают некоторыми преимуществами по сравнению с секционированными)

Поскольку реализации объединенных и секционированных таблиц в MySQL имеют много общего, то и некоторые ограничения у них сходны. Например, существует практический предел количества объединяемых таблиц и секций. В большинстве случаев он составляет несколько сотен, после чего эффективность заметно падает. Ограничения, относящиеся к каждому их двух механизмов, подробнее мы рассмотрим ниже.

Объединенные таблицы

Если хотите, можете считать объединенные таблицы устаревшим и более ограниченным вариантом секционирования, но у них по-прежнему есть право на существование. Более того, они обладают рядом возможностей, которые у секций отсутствуют.

Объединенная таблица – это просто контейнер для реальных таблиц. Имена объединяемых таблиц задаются с помощью ключевого слова `UNION` в команде `CREATE TABLE`. Следующий пример демонстрирует многие аспекты объединенных таблиц:

```
mysql> CREATE TABLE t1(a INT NOT NULL PRIMARY KEY) ENGINE=MyISAM;
mysql> CREATE TABLE t2(a INT NOT NULL PRIMARY KEY) ENGINE=MyISAM;
mysql> INSERT INTO t1(a) VALUES(1),(2);
mysql> INSERT INTO t2(a) VALUES(1),(2);
mysql> CREATE TABLE mrg(a INT NOT NULL PRIMARY KEY)
    -> ENGINE=MERGE UNION=(t1, t2) INSERT_METHOD=LAST;
mysql> SELECT a FROM mrg;
+------+
| a    |
+------+
|    1 |
|    1 |
|    2 |
|    2 |
+------+
```

Отметим, что все объединяемые таблицы должны иметь одинаковое количество и типы столбцов, и что все индексы, построенные над объединенной таблицей, существуют и над ее составляющими. Это обязательное требование к созданию объединенной таблицы. Отметим также, что в каждой из объединяемых таблиц имеется первичный ключ, состоящий из одного столбца, тем не менее, в объединенной таблице присутствуют строки-дубликаты. Это одно из ограничений объединенных таблиц: каждая таблица внутри объединения ведет себя нормально, однако объединенная таблица не проверяет ограничений по всему набору своих составляющих.

Инструкция `INSERT_METHOD=LAST` в определении таблицы говорит серверу MySQL о том, что все вставки командой `INSERT` следует производить в последнюю таблицу. Управлять тем, в какое место объединенной таблицы вставляются новые строки, можно только с помощью этого параметра, который принимает значения `LAST` или `FIRST` (вставлять в первую из объединяемых таблиц). Впрочем, никто не запрещает осуществлять вставку в составляющие таблицы напрямую. Секционированные таблицы обеспечивают лучший контроль над тем, где сохраняются данные.

Результат выполнения команды `INSERT` виден и в объединенной таблице, и в одной из составляющих ее:

```
mysql> INSERT INTO mrg(a) VALUES(3);
mysql> SELECT a FROM t2;
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
+---+
```

У объединенных таблиц есть ряд других интересных особенностей и ограничений. Например, что происходит, когда сама объединенная таблица или одна из ее составляющих удаляется? При удалении объединенной таблицы все ее составляющие остаются на месте, однако уничтожение любой из составляющих приводит к последствиям, зависящим от операционной системы. Например, на платформе GNU/Linux дескриптор файла удаленной таблицы остается открытым, и таблица все еще доступна, но только через объединенную таблицу:

```
mysql> DROP TABLE t1, t2;
mysql> SELECT a FROM mrg;
+------+
| a    |
+------+
|    1 |
|    1 |
|    2 |
|    2 |
|    3 |
+------+
```

Есть еще немало ограничений и частных случаев. Мы предлагаем вам познакомиться с деталями в руководстве, но все-таки отметим, что команда `REPLACE` для объединенных таблиц не работает вовсе, а механизм автоинкремента (ключевое слово `AUTO_INCREMENT`) работает не так, как вы думаете.

Объединенные таблицы и производительность

Способ, которым объединенные таблицы реализованы в MySQL, имеет некоторые существенные последствия для производительности. Как и любое другое средство MySQL, объединенные таблицы для каких-то задач приспособлены лучше, а для каких-то – хуже. Ниже перечислены некоторые аспекты объединенных таблиц, о которых следует помнить:

- Для объединенной таблицы необходимо больше открытых файловых дескрипторов, чем для обычной таблицы, содержащей те же данные. Хотя выглядит объединенная таблица как одна таблица, на самом деле каждая составляющая таблица открывается независимо от остальных. Поэтому единственная запись в кэше таблиц может быть связана с большим количеством файловых дескрипторов. Сле-

довательно, даже если вы сконфигурировали кэш таблиц так, чтобы сервер не превышал лимитов операционной системы на количество файловых дескрипторов, открытых в одном процессе, объединенные таблицы все же могут стать причиной выхода за указанные пределы.

- Команда CREATE, которая создает объединенную таблицу, не проверяет, совместимы ли ее составляющие. Если определения объединяемых таблиц слегка различаются, то MySQL может создать объединенную таблицу, которой впоследствии не сумеет воспользоваться. Кроме того, если изменить определение одной из составляющих таблиц уже после создания объединенной таблицы, последняя перестанет работать, и сервер выдаст сообщение об ошибке «ERROR 1168 (HY000): Unable to open underlying table which is differently defined or of non-MyISAM type or doesn't exist» (Не могу открыть составляющую таблицу, поскольку она определена по-другому, имеет тип, отличный от MyISAM, или не существует).
- Запросы, обращенные к объединенной таблице, переадресуются к каждой из составляющих таблиц. В результате поиск единственной строки может оказаться медленнее по сравнению с поиском в одной таблице. Поэтому рекомендуется ограничивать количество объединяемых таблиц, особенно если объединенная таблица стоит на втором или последующих местах в операции соединения. Чем меньше количество данных, к которым вы обращаетесь в одной операции, тем выше стоимость доступа к каждой таблице в расчете на операцию в целом. Вот несколько соображений, которые стоит иметь в виду при планировании использования объединенных таблиц:
  - Для запросов по диапазону накладные расходы на доступ к составляющим таблицам не так существенны, как для запросов на поиск одной строки
  - Сканирование выполняется для объединенной таблицы так же быстро, как для обычной
  - Поиск по уникальному и первичному ключу прекращается, как только искомая строка найдена. В данном случае сервер обращается к составляющим таблицам поочередно, пока не найден нужное значение, после чего оставшиеся таблицы не просматриваются
  - Составляющие таблицы читаются в порядке, указанном в команде CREATE TABLE. Если вам часто нужно извлекать данные в определенном порядке, то этой особенностью можно воспользоваться, чтобы ускорить операцию сортировки слиянием.

Плюсы объединенных таблиц

Достоинства объединенных таблиц проявляются особенно ярко, когда можно естественным образом разделить данные на активные и неактивные. Классический пример – журналы. Допускается запись только в конец журнала, поэтому можно, например, создавать по одной таблице на

каждый день. В этом случае в начале каждого дня создается новая составляющая таблица, после чего определение объединенной таблицы изменяется, чтобы присоединить ее. При желании можно заодно исключить таблицу за предыдущий день из объединенной таблицы, преобразовать ее в формат MyISAM со сжатием, а затем вернуть обратно.

Но это не единственное применение объединенных таблиц. Они часто используются в хранилищах данных, поскольку упрощают управление очень большими наборами данных. Практически невозможно обслуживать одну терабайтную таблицу, но задача существенно упрощается, если она представляет собой объединение таблиц размером 50 Гбайт.

При работе с очень большими базами приходится думать не только об обычных операциях, но и о том, как восстановить данные после сбоя. Поддержание небольшого размера таблиц – очень неплохая мысль, если, конечно, она реализуема в ваших условиях. Гораздо быстрее проверить и исправить набор небольших таблиц, чем одну гигантскую, особенно если последняя не помещается в память. Кроме того, при наличии нескольких таблиц процедуру проверки и исправления можно распараллелить.

При организации хранилищ информации встает также вопрос об уничтожении старых данных. Использование команды `DELETE` для удаления строк из очень большой таблицы в лучшем случае неэффективно, а в худшем может привести к катастрофе, но что может быть проще, чем изменить определение объединенной таблицы и с помощью команды `DROP TABLE` избавиться от старых данных. Эта процедура легко поддается автоматизации.

Объединенные таблицы полезны не только для протоколирования и обслуживания больших наборов данных. Они прекрасно подходят при генерации таблиц «на лету». Создание и удаление объединенных таблиц обходится дешево, поэтому их можно использовать так же, как представления с фразой `UNION ALL`; однако накладные расходы при этом ниже, так как сервер не копирует результаты во временную таблицу перед отправкой клиенту. Поэтому объединенные таблицы отлично приспособлены для нужд генерации отчетов и хранилищ данных. К примеру, можно создать периодически выполняемое задание, которое будет запускаться каждую ночь, и объединять вчерашние данные с данными за последние 8 дней, последние 15 дней и так далее. Потом на основе этих данных можно будет составлять недельные отчеты. Это позволит запускать запросы для подготовки регулярных отчетов без каких бы то ни было модификаций, при этом каждый раз автоматически будет производиться доступ к нужным данным. Можно даже создавать временные объединенные таблицы. с представлениями такой пример не пройдет.

Поскольку объединенные таблицы не скрывают составляющих MyISAM-таблиц, то можно делать некоторые вещи, невозможные при использовании секций.

- Одна MyISAM-таблица может входить в несколько объединенных таблиц.
- Составляющие таблицы можно переносить с одного сервера на другой; для этого достаточно скопировать файлы с расширениями *.frm*, *.MYI* и *.MYD*.
- В объединенную таблицу легко добавлять составляющие таблицы; достаточно создать новую таблицу и изменить определение объединения.
- Можно создать временную объединенную таблицу, которая включает только необходимые данные, например за конкретный период времени. Секции этого не позволяют.
- Можно исключить таблицу из объединения, если требуется поместить ее в резервную копию, восстановить с копии, изменить определение, исправить или выполнить еще какую-то операцию. Впоследствии таблицу можно вернуть обратно в объединение.
- Команда `myisampack` позволяет сжимать некоторые или все составляющие таблицы.

Напротив, части секционированных таблиц скрыты сервером MySQL и доступны только через саму секционированную таблицу.

Секционированные таблицы

На нижнем уровне реализация секционирования в MySQL очень похожа на организацию объединенных таблиц. Однако она более тесно интегрирована с сервером и отличается от объединенных таблиц в одном существенном отношении: конкретная строка данных может храниться в одной и только одной секции. В определении таблицы указывается способ распределения строк по секциям, основанный на *функции секционирования*, о которой мы поговорим чуть ниже. Это означает, что первичные и уникальные ключи работают, как и ожидается, то есть они уникальны для всей таблицы. Так что оптимизатор может обрабатывать запросы к секционированным таблицам более интеллектуально, чем к объединенным.

Ниже перечислены некоторые важные достоинства секционированных таблиц.

- Можно указать, что некоторые строки должны храниться вместе в одной секции; что позволяет уменьшить объем данных, просматриваемых сервером, и, следовательно, ускорить выполнение запроса. Например, если секционирование производится по диапазону дат, а в запросе указан диапазон, попадающих в одну секцию, то к остальным секциям сервер может не обращаться.
- Секционированные данные проще обслуживать, в частности нетрудно вычистить устаревшие значения путем удаления целой секции.

- Секционированные данные можно распределить по физически разным устройствам, так что сервер сможет более эффективно использовать жесткие диски.

Реализация секционирования в MySQL еще не окончательна и слишком сложна для того, чтобы описывать ее здесь во всех деталях. Мы хотим обратить внимание лишь на аспекты, относящиеся к производительности, а за базовой информацией отсылаем к руководству по MySQL, в котором имеется очень много материалов по секционированию. Рекомендуем прочитать целиком главу, посвященную секционированию, и заглянуть в разделы, в которых описываются команды `CREATE TABLE`, `SHOW CREATE TABLE`, `ALTER TABLE`, `INFORMATION_SCHEMA.PARTITIONS table` и `EXPLAIN`. Из-за секционирования синтаксис команд `CREATE TABLE` и `ALTER TABLE` заметно усложнился.

Как и объединенная таблица, на уровне подсистемы хранения секционированная таблица состоит из набора отдельных таблиц (секций) со своими индексами. Это означает, что требования, предъявляемые секционированными таблицами к памяти и к файловым дескрипторам, аналогичны тем, которые относятся к объединенным таблицам. Однако к секциям нельзя обращаться независимо от секционированной таблицы, и каждая секция принадлежит одной и только одной таблице.

Выше уже отмечалось, что MySQL применяет функцию секционирования, чтобы решить, в какую секцию должна попасть конкретная строка. Эта функция должна возвращать непостоянное, детерминированное целое число. Существует несколько видов секционирования. В случае секционирования *по диапазону* для каждой секции задается диапазон значений, и строки распределяются по секциям в зависимости от того, в какой диапазон они попадают. MySQL поддерживает также секционирование *по ключам*, *хеш-секционирование* и *секционирование по списку*. У каждого способа есть свои сильные и слабые стороны, а также ограничения – особенно в части первичных ключей.

Почему секционирование работает?

Ключом к проектированию секционированных таблиц в MySQL является представление о секционировании, как о крупномодульном индексировании. Предположим, что имеется таблица, содержащая миллиард строк с данными об истории продаж – по одной строке на каждый день и на каждый товар, и пусть размер каждой строки достаточно велик, скажем, 500 байтов. В эту таблицу можно вставлять новые строки, но старые никогда не изменяются, а в выполняемых запросах, как правило, указывается диапазон дат. Основная проблема при выполнении запросов заключается в том, что таблица слишком велика, ее размер без сжатия и без индексов составляет примерно полтерабайта.

Один из способов ускорить выполнение запросов для получения данных за один день – добавить первичный ключ по столбцам (`day`, `itemno`) и использовать InnoDB. Тогда данные за один день физически будут распо-

лагаться рядом, поэтому при обработке запроса нужно будет просматривать меньше информации. Альтернатива – использовать MyISAM и вставлять строки в нужном порядке, чтобы при просмотре индекса данные читались последовательно, а не произвольно.

Еще один вариант – отказаться от первичного ключа и организовать по одной секции на каждый день. Тогда запрос, который обращается к некоторому диапазону дат, должен будет просматривать секции целиком, но это может оказаться гораздо эффективнее поиска по индексу в гигантской таблице. Можно считать секционирование грубым аналогом индекса: мы говорим серверу MySQL, где примерно искать строку, если известна дата. Но при этом не потребляется ни место на диске, ни память – именно потому, что знание секции не определяет местоположение строки точно (как в случае индекса).

Но не поддавайтесь искушению одновременно секционировать таблицу *и* задать первичный ключ – производительность при этом может даже упасть, особенно если запрос затрагивает все секции. При использовании секционирования необходимо производить тщательное тестирование, поскольку секционированные таблицы далеко не всегда способствуют росту производительности.

Примеры секционирования

Мы приведем два небольших примера ситуаций, в которых секционирование может быть полезно. Во-первых, посмотрим, как проектируется секционированная таблица для распределения значений по датам. Предположим, что имеется агрегированная статистическая информация по заказам и продажам для каждого товара. Поскольку часто выполняются запросы по диапазонам дат, то делаем дату заказа частью первичного ключа и применяем подсистему хранения InnoDB для кластеризации значений по датам. Теперь можно «кластеризовать» данные на более высоком уровне с помощью секционирования таблицы по диапазонам дат. Вот как выглядит определение базовой таблицы без задания секционирования:

```
CREATE TABLE sales_by_day (
   day DATE NOT NULL,
   product INT NOT NULL,
   sales DECIMAL(10, 2) NOT NULL,
   returns DECIMAL(10, 2) NOT NULL,
   PRIMARY KEY(day, product)
) ENGINE=InnoDB;
```

Для данных, содержащих даты, часто применяется секционирование по году или по дню. В этих случаях в качестве функции секционирования можно взять соответственно `YEAR()`или `TO_DAYS()`. В общем случае хорошая функция для секционирования по диапазону должна линейно зависеть от значения, определяющего секцию, а эти функции такому условию удовлетворяют. Давайте секционируем данные по годам:

```
mysql> ALTER TABLE sales_by_day
    -> PARTITION BY RANGE(YEAR(day)) (
    ->   PARTITION p_2006 VALUES LESS THAN (2007),
    ->   PARTITION p_2007 VALUES LESS THAN (2008),
    ->   PARTITION p_2008 VALUES LESS THAN (2009),
    ->   PARTITION p_catchall VALUES LESS THAN MAXVALUE );
```

Теперь вставляемые строки будут попадать в секцию, соответствующую значению в столбце `day`:

```
mysql> INSERT INTO sales_by_day(day, product, sales, returns) VALUES
    -> ('2007-01-15', 19, 50.00, 52.00),
    -> ('2008-09-23', 11, 41.00, 42.00);
```

Эти данные нам еще понадобятся чуть ниже. Однако прежде чем двигаться дальше, хотелось бы отметить важное ограничение: чтобы впоследствии добавить новые годы, придется изменить определение таблицы, что в случае большого размера обойдется дорого (а ведь мы предполагаем, что таблица велика, иначе, зачем вообще прибегать к секционированию). Имеет смысл подумать заранее и определить больше годов, чем необходимо сегодня. Даже если вы в течение длительного времени не будете их использовать, на производительности это никак не скажется.

Еще одно типичное применение секционированных таблиц состоит в том, чтобы просто равномерно распределить строки в большой таблице. Предположим например, что для некоторой гигантской таблицы выполняется достаточно много запросов. Если требуется, чтобы одновременные запросы обслуживались разными физическими дисками, то желательно, чтобы MySQL распределила строки по этим дискам. В данном случае нам неважно, будут логически связанные данные располагаться близко друг к другу или нет, нужно лишь, чтобы они распределялись равномерно без нашего участия. В следующем примере эта задача решается распределением по модулю первичного ключа:

```
mysql> ALTER TABLE mydb.very_big_table
    -> PARTITION BY KEY(<primary key columns>) (
    -> PARTITION p0 DATA DIRECTORY='/data/mydb/big_table_p0/',
    -> PARTITION p1 DATA DIRECTORY='/data/mydb/big_table_p1/');
```

Того же эффекта можно добиться по-другому, с помощью RAID-контроллера. Иногда это даже лучше: будучи реализован аппаратно, этот механизм скрывает детали своей работы, поэтому не приходится усложнять схему базы данных и запросы. Да и результат будет более равномерным, если ваша единственная цель – физически распределить данные.

Ограничения секционированных таблиц

Секционированные таблицы – не панацея. Вот перечень некоторых ограничений в текущей реализации.

- Все секции должны управляться одной и той же подсистемой хранения. Например, нельзя сжать только часть секций, хотя для составляющих объединенной таблицы это допустимо.
- Любой уникальный индекс над секционированной таблицей должен содержать столбцы, на которые ссылается функция секционирования. Поэтому во многих учебных примерах стараются не использовать первичный ключ. Для хранилищ данных таблицы без первичных ключей и уникальных индексов – дело обычное, но в OLTP-системах такое встречается гораздо реже. Следовательно, вы далеко не так свободны в выборе способа секционирования данных, как казалось поначалу.
- Хотя сервер MySQL может обойтись без доступа к каждой секции при обработке запроса к секционированной таблице, он тем менее ставит блокировки на все секции.
- Существует ряд ограничений на функции и выражения, которые можно использовать в механизме секционирования.
- Некоторые подсистемы хранения вообще не поддерживают секционирование.
- Внешние ключи не работают.
- Нельзя пользоваться командой `LOAD INDEX INTO CACHE`.

Существует много других ограничений (по крайней мере, на момент написания этой книги, когда версия MySQL 5.1 еще не была официально выпущена). В некоторых отношениях секционированные таблицы обладают меньшей гибкостью, чем объединенные. Например, для секционированной таблицы нельзя создавать индекс посекционно; команда `ALTER` заблокирует и перестроит всю таблицу. Объединенные таблицы предоставляют больше возможностей, например индексирование составляющих таблиц по одной. Аналогично, невозможно выполнять резервное копирование или восстановление только одной секции, тогда как для составляющих объединенной таблицы это допустимо.

Выиграет ли некая таблица от секционирования, зависит от многих факторов. Чтобы решить, устраивает ли вас такое решение, необходимо проводить эталонное тестирование приложения.

Оптимизация запросов к секционированным таблицам

Секционирование привносит новые способы оптимизации запросов (и вместе с ними новые сюрпризы). Оптимизатор может использовать функцию секционирования, чтобы *отсечь* некоторые секции, то есть исключить их из рассмотрения при обработке запроса. Для этого ему нужно лишь понять, что нужные строки могут находиться только в определенных секциях. Таким образом, отсечение позволяет просматривать гораздо меньше данных (в лучшем случае).

Очень важно задавать ключ, по которому производится секционирование, во фразе WHERE, даже если он больше ни для чего не нужен; только при этом условии оптимизатор сможет отсечь ненужные секции. В противном случае подсистема выполнения запроса будет обращаться ко всем секциям, как это происходит с объединенными таблицами, а для больших таблиц этот подход может оказаться очень медленным.

Чтобы понять, производит ли оптимизатор отсечение секций, можно воспользоваться командой EXPLAIN PARTITIONS. Вернемся к введенным ранее тестовым данным:

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales_by_day
   partitions: p_2006,p_2007,p_2008
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 3
        Extra:
```

Как видите, этот запрос обращается ко всем секциям. А теперь добавим во фразу WHERE условие и посмотрим, что получится:

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day WHERE day > '2007-01-01'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales_by_day
   partitions: p_2007,p_2008
```

Оптимизатор достаточно «умен» для того чтобы понять, как производить отсечение. Он даже может преобразовать диапазон в список дискретных значений и отсечь секции, соответствующие элементам этого списка. Однако он не всеведущ. Например, следующая фраза WHERE теоретически допускает отсечение, но MySQL этого не сделает:

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day WHERE YEAR(day) = 2007\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales_by_day
   partitions: p_2006,p_2007,p_2008
```

В настоящее время MySQL умеет производить отсечение только на основе сравнения со столбцами, указанными в функции секционирования. Он не может принять решение об отсечении по результату вычисления выражения, даже если выражение совпадает с функцией секциониро-

вания. Однако предыдущий запрос можно переписать в эквивалентном виде:

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day
    -> WHERE day BETWEEN '2007-01-01' AND '2007-12-31'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales_by_day
   partitions: p_2007
```

Поскольку теперь во фразе `WHERE` явно указан столбец, по которому производилось секционирование, а не выражение, то оптимизатор может спокойно отсечь все секции, кроме одной.

Оптимизатор умеет также отсекать секции во время обработки запроса. Например, если секционированная таблица – вторая в операции соединения, а само соединение производится по ключу секционирования, то MySQL будет искать соединяемые строки только в подходящих секциях. Это существенное отличие от объединенных таблиц, для которых такой запрос будет просматривать все составляющие таблицы.

Распределенные (XA) транзакции

Если транзакции подсистем хранения обеспечивают свойства ACID внутри самой подсистемы, то распределенная (XA) транзакция позволяет распространить некоторые из этих свойств вовне подсистемы хранения и даже вовне базы данных – посредством механизма двухфазной фиксации. Начиная с версии 5.0 в MySQL реализована частичная поддержка XA-транзакций.

Для XA-транзакции требуется координатор транзакции, который просит всех участников подготовиться к фиксации (фаза 1). Получив от всех участников ответ «готов», координатор предлагает им выполнить фиксацию (фаза 2). MySQL может выступать в роли участника XA-транзакции, но не в роли координатора.

На самом деле в MySQL существует два вида XA-транзакций. Сервер MySQL может участвовать в любой управляемой извне распределенной транзакции, но он использует тот же механизм и внутри себя для координации подсистем хранения и записи в двоичный журнал.

Внутренние XA-транзакции

Причиной для внутреннего использования XA-транзакций в MySQL является желание архитектурно отделить сервер от подсистем хранения. Подсистемы хранения абсолютно независимы и ничего не знают друг о друге, поэтому любая транзакция, пересекающая границы подсистем, по своей природе является распределенной и нуждается в третьей стороне для координации. В роли третьей стороны высту-

пает сервер MySQL. Не будь XA-транзакций, для фиксации транзакции, пересекающей границы подсистем, пришлось бы последовательно просить каждую подсистему выполнить свою часть фиксации. Но тогда аварийный останов в момент после того, как одна подсистема выполнила фиксацию, а другая еще не успела, привел бы к нарушению правил транзакционности (напомним, что транзакция по определению выполняет все или ничего).

Если взглянуть на двоичный журнал, как на «подсистему хранения» протоколируемых событий, то становится понятно, почему XA-транзакции нужны даже тогда, когда в обработке запроса участвует лишь одна подсистема. Синхронизация фиксации в смысле подсистемы хранения с «фиксацией» события в двоичном журнале – это распределенная транзакция, поскольку за двоичный журнал отвечает сервер.

В настоящее время протокол XA создает некую дилемму в плане производительности. В версии 5.0 он «поломал» поддержку *групповой фиксации* в InnoDB (метод, позволяющий зафиксировать несколько транзакций одной операцией ввода/вывода), поэтому требуется выполнять больше системных вызовов `fsync()`, чем реально необходимо. Кроме того, если двоичный журнал включен, то каждая транзакция должна синхронизироваться с двоичным журналом, и вместо одного сброса в журнал на операцию фиксации необходимо выполнять два. Другими словами, если требуется, чтобы двоичный журнал безопасно синхронизировался с транзакциями, то на каждую транзакцию будет приходиться, по меньшей мере, три вызова `fsync()`. Единственный способ предотвратить это – отключить двоичный журнал и установить параметр `innodb_support_xa` в 0.

Такие установки параметров несовместимы с репликацией. Для репликации необходимы и двоичный журнал, и поддержка XA, а кроме того – чтобы уж все было безопасно – параметр `need sync_binlog` должен быть равен 1, тогда подсистема хранения и двоичный журнал будут синхронизированы (в противном случае поддержка XA теряет смысл, так как двоичный журнал может не «зафиксироваться» на диске). Это одна из причин, по которым мы настоятельно рекомендуем использовать RAID-контроллер, оборудованный кэшем записи с аварийным электропитанием: кэш может ускорить выполнение дополнительных вызовов `fsync()` и вернуть производительность в норму.

В следующей главе мы подробно рассмотрим, как конфигурировать журнал транзакций и двоичный журнал.

Внешние XA-транзакции

Сервер MySQL может быть участником распределенной транзакции, но не может ей управлять. Он не поддерживает спецификацию XA в полном объеме. Например, спецификация XA позволяет соединять в одной транзакции таблицы из разных баз данных, но в настоящее время MySQL этого не умеет.

Внешние XA-транзакции еще дороже внутренних вследствие дополнительных задержек и большей вероятности ошибки одного из участников. Использовать XA поверх сети WAN, а уж тем более Интернета не рекомендуется из-за непредсказуемого поведения сети. Лучше избегать XA-транзакций, когда имеется хотя бы один непредсказуемый компонент, например медленная сеть или пользователь, который долго не нажимает кнопку «Сохранить». Все, что способно задержать транзакцию, обходится очень дорого, поскольку может вызвать задержку не в одной, а во многих системах.

Впрочем, есть другие способы организации высокопроизводительных распределенных транзакций. Например, можно вставить данные локально и поставить обновление в очередь, а затем атомарно распространить его в составе гораздо более маленькой, быстрой транзакции. Можно также воспользоваться репликацией MySQL для доставки данных из одного места в другое. Мы обнаружили, что некоторым приложениям, использующим распределенные транзакции, они вообще не нужны.

Несмотря на все вышесказанное, XA-транзакции могут оказаться полезным способом синхронизации данных между серверами. Этот метод хорошо работает, когда по какой-то причине применять репликацию невозможно или скорость выполнения обновлений не критична.

6

Оптимизация параметров сервера

Нам часто задают вопросы примерно такого плана: «Как оптимально задать конфигурационные параметры для моего сервера с 16 Гбайт памяти и объемом данных 100 Гбайт?» Но на этот вопрос нет однозначного ответа. Конфигурация сервера сильно зависит от оборудования, объема данных, типичных запросов и требований к системе (времени реакции, долговечности и согласованности транзакций и т. д.).

Предлагаемая по умолчанию конфигурация рассчитана на умеренное потребление ресурсов, поскольку предполагается, что MySQL – не единственная программа, работающая на сервере. В конфигурации по умолчанию используется ровно столько ресурсов, сколько необходимо для запуска MySQL и выполнения простых запросов к небольшому объему данных. Если объем хранимой в базах информации превышает несколько мегабайтов, то параметры, безусловно, необходимо настраивать. Можете для начала взять один из конфигурационных файлов, входящих в состав дистрибутива MySQL, и модифицировать его в соответствии с вашими потребностями.

Не следует ожидать, что любое изменение конфигурации даст заметный прирост производительности. В зависимости от рабочей нагрузки обычно удается добиться двух- или трехкратного повышения за счет выбора подходящих значений небольшой группы параметров (а вот что должно войти в эту «небольшую группу», зависит от самых разных факторов). Затем улучшения производятся постепенно. Возможно, вы заметите, что некоторый запрос выполняется медленно, и сумеете улучшить его, подправив один-два параметра, но заставить сервер работать на порядок быстрее удается крайне редко. Чтобы достичь такого результата, обычно приходится пересматривать схему, запросы и всю архитектуру приложения.

Мы начнем эту главу с демонстрации того, как работают конфигурационные параметры MySQL и как их можно изменить. Затем мы обсудим,

каким образом MySQL использует память и что тут можно оптимизировать. Далее мы рассмотрим те же вопросы в применении к вводу/выводу и дисковой памяти. В следующем разделе речь пойдет о настройке с учетом рабочей нагрузки, цель которой – оптимизировать MySQL под ваши конкретные условия. И, наконец, мы немного поговорим о динамической настройке переменных для отдельных запросов, нуждающихся в нестандартных параметрах.

Замечание по поводу терминологии: поскольку многие параметры командной строки MySQL соответствуют серверным переменным, мы будем употреблять слова *параметр* и *переменная* как синонимы.

Основы конфигурирования

В этом разделе мы дадим краткий обзор процедуры конфигурирования MySQL. Сначала объясним, как работает механизм конфигурирования, а затем приведем некоторые рекомендации, основанные на опыте. Вообще говоря, MySQL снисходительно относится к ошибкам конфигурации, но следующие замечания помогут сэкономить немало времени и сил.

Прежде всего, вы должны знать, откуда MySQL получает конфигурационную информацию: из аргументов командной строки и параметров в конфигурационном файле. В системах на базе UNIX таковым обычно является файл */etc/my.cnf* или */etc/mysql/my.cnf*. Если вы пользуетесь сценариями запуска, входящими в состав операционной системы, то обычно только в этом файле и надо определять конфигурацию. Если же вы запускаете MySQL вручную, например на тестовой системе, то можете задавать параметры и в командной строке.

Большинство конфигурационных переменных называются так же, как соответствующие параметры командной строки, но есть несколько исключений. Например, параметр *--memlock* устанавливает переменную `locked_in_memory`.

Любые параметры, которые должны действовать постоянно, следует помещать в глобальный конфигурационный файл, а не задавать в командной строке. В противном случае вы рискуете случайно запустить сервер без них. Кроме того, разумно хранить все конфигурационные файлы в одном месте, чтобы их было проще инспектировать.

Не забывайте, где находится конфигурационный файл вашего сервера! Нам приходилось встречать людей, которые безуспешно пытались настроить сервер, изменяя файл, который он и не собирался никогда читать, например */etc/my.cnf* в системе Debian GNU/Linux, где сервер ищет файл */etc/mysql/my.cnf*. Иногда файлы могут находиться в разных местах, быть может, потому что предыдущий системный админи-

стратор тоже запутался. Если вы не знаете, какой файл читает сервер, то можно у него же и спросить:

```
$ which mysqld
/usr/sbin/mysqld
$ /usr/sbin/mysqld --verbose --help | grep -A 1 'Default options'
Default options are read from the following files in the given order:
/etc/mysql/my.cnf ~/.my.cnf /usr/etc/my.cnf
```

Все вышесказанное относится к типичным инсталляциям, когда на машине установлен единственный сервер. Можно придумать и более сложные конфигурации, но все подобные способы нестандартны. В дистрибутив MySQL входит программа *mysqlmanager*, которая может запускать несколько экземпляров сервера, используя один конфигурационный файл с несколькими секциями (она заменила старый сценарий *mysqld_multi*). Однако во многих дистрибутивах операционных систем эта программа отсутствует или не используется в сценариях запуска. Честно говоря, операционные системы зачастую вообще игнорируют сценарии запуска, поставляемые вместе с MySQL.

Конфигурационный файл разделен на секции, каждая из которых начинается со строки с именем секции в квадратных скобках. Любая программа, входящая в состав дистрибутива MySQL, обычно читает секцию, имя которой совпадает с именем самой программы. Кроме того, многие клиентские приложения читают секцию `client`, в которую можно поместить общие для всех клиентов параметры. Сервер обычно читает секцию `mysqld`. Следите за тем, чтобы помещать параметры в нужную секцию, иначе они не возымеют эффекта.

Синтаксис, область видимости и динамичность

Конфигурационные параметры записываются строчными буквами, слова разделяются символами подчеркиваниями или дефисами. Следующие формы записи эквивалентны, любую из них можно встретить в командной строке или в конфигурационном файле:

```
/usr/sbin/mysqld --auto-increment-offset=5
/usr/sbin/mysqld --auto_increment_offset=5
```

Мы рекомендуем выбрать какой-нибудь один стиль и придерживаться его. Так будет проще искать в файле конкретный параметр.

У конфигурационной переменной может быть несколько областей видимости. Некоторые параметры действуют на уровне всего сервера (глобальная область видимости), другие могут задаваться по-разному для каждого соединения (сеансовая область видимости), третьи относятся к конкретным объектам. У многих сеансовых параметров есть глобальные эквиваленты, которые можно рассматривать как умолчания. Если изменить сеансовую переменную, то это отразится только на том соединении, где она была изменена; после закрытия соединения изменения будут потеряны. Вот несколько примеров разнообразного поведения.

- Переменная query_cache_size имеет глобальную область видимости.
- Переменная sort_buffer_size имеет глобальное значение по умолчанию, но может быть изменена на уровне сеанса.
- Переменная join_buffer_size имеет глобальное значение по умолчанию, может быть изменена на уровне сеанса, но, кроме того, для каждого запроса, в котором соединяется несколько таблиц, можно выделить по одному буферу *на операцию соединения*, то есть для одного запроса может существовать несколько буферов соединения.

Помимо задания переменных в конфигурационном файле многие из них (но не все) можно изменять во время работы сервера. Такие конфигурационные переменные в MySQL называются *динамическими*. Ниже показаны разные способы динамически изменить значение переменной sort_buffer_size на уровне сеанса и глобально:

```
SET          sort_buffer_size  = <value>;
SET GLOBAL   sort_buffer_size  = <value>;
SET        @@sort_buffer_size := <value>;
SET @@session.sort_buffer_size := <value>;
SET  @@global.sort_buffer_size := <value>;
```

Изменяя переменные динамически, не забывайте, что новое значение будет потеряно после останова MySQL. Если вы хотите сохранить измененные параметры, то запишите их также в конфигурационный файл.

Изменение глобальной переменной во время работы сервера не отражается на ее значении в текущем и во всех остальных уже открытых сеансах. Связано это с тем, что сеансовые переменные инициализируются в момент создания соединения. После любого изменения переменной выполняйте команду SHOW GLOBAL VARIABLES, чтобы удостовериться в том, что желаемый эффект достигнут.

Для разных переменных подразумеваются различные единицы измерения. Например переменная table_cache определяет количество кэшируемых таблиц, а не размер кэша таблиц в байтах. Переменная key_buffer_size задается в байтах, тогда как другие параметры могут измеряться в количестве страниц или в других единицах, например в процентах.

Для многих переменных можно указывать суффикс, например 1M означает один мегабайт. Однако такое соглашение применяется лишь при задании переменной в конфигурационном файле или в аргументе командной строки. При использовании SQL-команды SET следует указывать литеральное значение 1048576 или выражение, например 1024 * 1024. В конфигурационных файлах выражения не допускаются.

В команде SET можно задавать также специальное значение DEFAULT. Если присвоить такое значение сеансовой переменной, то она станет равна соответствующей глобальной переменной. Присваивание же значения DEFAULT глобальной переменной делает ее равной значению, заданному на этапе компиляции сервера (а не тому, что указано в конфигурацион-

ном файле). Это бывает полезно, когда нужно вернуть переменной то состояние, которое она имела в момент открытия соединения. Мы не рекомендуем применять этот прием к глобальным переменным, поскольку результат может отличаться от ожидаемого, – вы не получите значение, действовавшее на момент запуска сервера.

Побочные эффекты установки переменных

Динамическое задание переменных может иметь неожиданные побочные эффекты, например сброс на диск изменившихся блоков из буферов. Будьте осторожны при динамическом изменении переменных, так как можете, сами того не желая, заставить сервер выполнить большой объем работы.

Иногда назначение переменной можно вывести из ее имени. Например, переменная `max_heap_table_size` делает именно то, что подразумевает ее наименование: задает *максимальный* размер, до которого могут расти таблицы типа MEMORY. Однако имена не всегда отражают поведение, поэтому предположение о назначении переменной может оказаться ошибочным.

Рассмотрим некоторые важные переменные и последствия их динамического изменения.

`key_buffer_size`

При задании этой переменной сразу же резервируется указанный объем памяти для буфера ключей (он также называется кэшем ключей). Однако операционная система не выделяет физическую память до момента ее фактического использования. Так, требование выделить один гигабайт под буфер ключей вовсе не означает, что сервер немедленно получит весь гигабайт (о том, как можно следить за использованием памяти сервером, мы расскажем в следующей главе).

Как будет объяснено ниже в этой главе, MySQL позволяет создавать несколько кэшей ключей. Если присвоить этой переменной значение 0 для кэша ключей, отличного от подразумеваемого по умолчанию, то MySQL переместит все индексы из указанного кэша в кэш по умолчанию, а указанный кэш удалит, когда больше никто не будет его использовать. В результате задания этой переменной для несуществующего кэша указанный кэш будет создан.

Присваивание данной переменной ненулевого значения для существующего кэша приводит к тому, что память, отведенная под этот кэш, сбрасывается на диск[1]. Технически это оперативное действие, однако все операции, пытающиеся получить доступ к кэшу, блокируются до завершения сброса.

[1] На диск сбрасываются «грязные» блоки. – *Прим. науч. ред.*

`table_cache`

Задание этой переменной не дает немедленного эффекта – действие откладывается до момента, когда поток попытается открыть очередную таблицу. Вот тогда-то MySQL и проверяет значение данного параметра. Если оно больше текущего количества таблиц в кэше, то поток может поместить вновь открытую таблицу в кэш. В противном случае MySQL удалит из кэша неиспользуемые таблицы.

`thread_cache_size`

Задание этой переменной не дает немедленного эффекта – действие откладывается до момента следующего закрытия соединения. В этот момент MySQL проверит, есть ли в кэше место для хранения потока. Если да, то поток кэшируется для ассоциации с соединением, которое будет открыто в будущем. В противном случае поток уничтожается. При этом количество потоков в кэше, а, следовательно, и объем памяти, отведенной под кэш потоков, не сокращается сразу; уменьшение происходит только тогда, когда новое соединение позаимствует для себя поток из кэша (MySQL добавляет потоки в кэш только при закрытии соединения, а удаляет их из кэша лишь при создании новых соединений).

`query_cache_size`

MySQL выделяет и инициализирует указанное количество памяти для кэша запросов в момент запуска сервера. При изменении этой переменной (даже если новое значение совпадает с предыдущим) MySQL немедленно удаляет все кэшированные запросы, устанавливает новый размер кэша и повторно инициализирует отведенную под него память.

`read_buffer_size`

MySQL не выделяет память для этого буфера, пока она не потребуется запросу. Когда же необходимость возникает, MySQL выделяет весь блок запрошенного размера.

`read_rnd_buffer_size`

MySQL не выделяет память для этого буфера, пока она не потребуется запросу. Когда же необходимость возникает, MySQL выделяет лишь столько памяти, сколько необходимо (имя `max_read_rnd_buffer_size` описывало бы назначение этой переменной более точно).

`sort_buffer_size`

MySQL не выделяет память для этого буфера, пока она не потребуется запросу с целью выполнения сортировки. Когда же необходимость возникнет, запрошенный блок выделяется целиком, даже если столько памяти и не нужно.

Ниже назначение этих переменных будет объяснено более подробно. Пока мы хотим лишь показать, какого поведения ожидать при изменении этих важных параметров.

Приступая к работе

Будьте осторожны при задании переменных. Больше – не всегда лучше; установив слишком большое значение, легко накликать беду на свою голову: памяти может не хватить, и тогда сервер начнет выгружать ее в файл подкачки или вообще выйдет за пределы адресного пространства.

Мы рекомендуем подготовить комплект эталонных тестов перед тем, как приступать к настройке сервера (эталонное тестирование обсуждалось в главе 2). Чтобы оптимизировать конфигурацию, вам потребуется комплект тестов, имитирующий типичную рабочую нагрузку и граничные случаи, например очень большие и сложные запросы. Обнаружив конкретную проблему, скажем, одиночный запрос, который выполняется медленно, вы можете попробовать оптимизировать данный случай, но при этом есть риск непреднамеренного негативного воздействия на другие запросы.

Обязательно нужно иметь систему мониторинга, которая определяет, улучшилась или ухудшилась общая производительность сервера после изменения. Если пренебречь этим, то можно сделать хуже, даже не осознавая того. Сколь раз мы видели, как кто-то менял конфигурацию сервера, полагая, что повысил быстродействие, тогда как на деле общая производительность снижалась, поскольку в разное время дня или в разные дни недели нагрузка варьировалась. В главе 14 мы рассмотрим некоторые системы мониторинга.

Самый лучший подход к делу – изменять не более одной-двух переменных за раз, и после каждой такой модификации прогонять тесты. Иногда результаты бывают удивительными; вот вы немного изменили переменную, и результаты улучшились, а потом еще чуть-чуть изменили – и раз! – резкое падение производительности. Если после такого изменения быстродействие падает, то задайте себе вопрос, не слишком ли большой объем ресурса вы запросили. Например, речь может идти о чрезмерном размере памяти для буфера, который часто выделяется и освобождается. Не исключено также, что возникло несоответствие между MySQL и операционной системой или оборудованием. Так, мы обнаружили, что оптимальное значение переменной `sort_buffer_size` может зависеть от способа работы кэша процессора, а при настройке переменной `read_buffer_size` нужно учитывать конфигурацию упреждающего чтения сервера и общие настройки подсистемы ввода/вывода. Больше – не всегда лучше. Кроме того, некоторые переменные зависят от других, и в полной мере освоить все это можно только на опыте и при условии понимания архитектуры системы. Например, оптимальное значение `innodb_log_file_size` зависит от значения `innodb_buffer_pool_size`.

Вы можете сэкономить массу своего (и того, кто придет на ваше место) времени, если будете делать заметки, например в виде комментариев в конфигурационном файле. Еще лучше поместить конфигурационный файл в систему управления версиями. Это в любом случае похвальная практика, поскольку позволяет откатывать изменения. Чтобы упростить обслуживание большого количества конфигурационных файлов, создайте символическую ссылку с конфигурационного файла на центральный репозиторий системы управления версиями. Прочитать о таких приемах можно в любой приличной книге по системному администрированию.

Прежде чем приступать к изменению конфигурационных параметров, следует настроить запросы и схему, обращая внимание, по крайней мере, на такие очевидные оптимизации, как добавление индексов. Если вы слишком далеко продвинетесь по пути изменения конфигурации, а потом займетесь модификацией запросов или схемы, то настройку, возможно, придется начинать заново. Не забывайте, что настройка – это непрерывный, итерационный процесс. Если только оборудование, рабочая нагрузка и данные не являются абсолютно неизменными, то шансы на то, что к конфигурированию придется вернуться, очень велики. Это означает, что не нужно стремиться выжать из сервера все до последней капли, ведь отдача от затраченного времени, скорее всего, будет крайне мала. Мы рекомендуем продолжать настройку до тех пор, пока не получится «достаточно хороший» результат, а потом ничего не трогать до появления причин полагать, что можно добиться существенного повышения производительности. Возможно, к этому занятию имеет смысл вернуться после изменения запросов или схемы.

Обычно мы разрабатываем образцы конфигурационных файлов для разных целей и используем их в качестве наших собственных умолчаний, особенно если в одном центре приходится администрировать несколько схожих серверов. Но, как мы уже предупреждали в самом начале главы, у нас нет одного «самого лучшего конфигурационного файла», скажем, для сервера с четырьмя процессорами, 16 Гбайт оперативной памяти и 12 жесткими дисками, который годился бы на все случаи жизни. Вам все-таки придется выработать собственные конфигурации, поскольку даже при наличии неплохой отправной точки все сильно зависит от того, как используется конкретное оборудование.

Общие принципы настройки

Конфигурирование можно представлять себе как процедуру из двух шагов: использование основных знаний о своей системе для создания осмысленной отправной точки и последующая модификация с учетом деталей рабочей нагрузки.

Проще всего взять за основу один из образцов конфигурационного файла, поставляемых вместе с MySQL. Делая выбор, примите во внимание аппаратное оснащение сервера. Сколько у него жестких дисков, сколько

процессоров, сколько памяти? Образцы конфигурационных файлов имеют говорящие имена, например: *my-huge.cnf*, *my-large.cnf*, *my-small.cnf*, поэтому разобраться, с чего начинать, совсем нетрудно. Однако образцы применимы лишь в том случае, когда вы работаете только с таблицами типа MyISAM. Если же используются другие подсистемы хранения, то придется разрабатывать конфигурацию с нуля.

Настройка использования памяти

Правильное конфигурирование работы MySQL с памятью жизненно важно для достижения хорошей производительности. Распределение памяти вам придется настраивать почти наверняка. Потребляемая память в MySQL распадается на две группы: подконтрольную и неподконтрольную вам. Вы не можете управлять тем, сколько памяти MySQL использует для запуска сервера, разбора запросов и внутренних целей, но тем, сколько ее потребляется для различных конкретных целей, управлять вполне можно. Правильно распорядиться подконтрольной вам памятью не так уж сложно, если понимать, что именно конфигурируется.

Подходить к настройке использования памяти можно поэтапно.

1. Определить абсолютный верхний предел объема памяти, которую MySQL может использовать.
2. Определить, сколько памяти MySQL будет использовать на каждое соединение, например для буферов сортировки и временных таблиц.
3. Определить, сколько памяти нужно операционной системе для нормальной работы. Сюда следует включить и память для других программ, работающих на той же машине, например периодически выполняемых заданий.
4. Если это имеет смысл, отдайте всю оставшуюся память под кэши MySQL, например, под пул буферов InnoDB.

Мы рассмотрим каждый из этих шагов в последующих разделах, а затем более подробно обсудим требования к различным кэшам MySQL.

Сколько памяти может использовать MySQL?

В любой конкретной системе существует верхний предел объема памяти, в принципе доступной MySQL. За точку отсчета следует взять объем физически установленной памяти. Если у сервера нет памяти, то и MySQL ее не получит.

Примите также во внимание ограничения, характерные для операционной системы и аппаратной архитектуры, например лимиты размера адресного пространства одного процесса в 32-разрядной ОС. Поскольку MySQL работает как один процесс с несколькими потоками, объем доступной ему памяти может быть серьезно ограничен. Скажем, в 32-разрядных ядрах Linux любой процесс может адресовать от 2,5 до 2,7 Гбайт памяти. Выход за пределы адресного пространства очень опасен и может привести к аварийному останову MySQL.

Существует много зависящих от операционной системы параметров и странностей, которые следует учитывать. К ним относятся не только лимиты на ресурсы, выделяемые одному процессу, но также размер стека и другие настройки. Ограничения на размер одного выделяемого блока памяти может налагать и системная библиотека *glibc*. Например, невозможно присвоить параметру `innodb_buffer_pool` значение, большее 2 Гбайт, если это максимальный размер блока, который может быть выделен *glibc* за один подход.

Некоторые ограничения относятся даже к 64-разрядным серверам. Например, на 64-разрядном сервере размеры многих буферов, в частности, буфера ключей, ограничены 4 Гбайт. Некоторые ограничения сняты в MySQL 5.1, и, вероятно, в будущем положение дел еще изменится, поскольку компания MySQL AB активно работает над тем, чтобы эта СУБД могла задействовать преимущества современного мощного оборудования. Максимальные значения каждой конфигурационной переменной документированы в руководстве по MySQL.

Сколько памяти нужно соединению?

MySQL нуждается в небольшом объеме памяти просто для того, чтобы поддерживать соединение (поток) открытым. Кроме того, сколько-то памяти необходимо, чтобы хотя бы начать выполнение запроса. Вы должны отвести достаточно ресурсов для выполнения запросов даже в периоды пиковой нагрузки. В противном случае запросы будут «сидеть на голодном пайке», вследствие чего станут выполняться очень медленно или вообще завершаться с ошибкой.

Вообще полезно знать, сколько памяти MySQL потребляет в пиковые периоды, но в некоторых ситуациях потребление памяти неожиданно и резко возрастает, что делает любые прогнозы ненадежными. Один из примеров – подготовленные команды, поскольку их может быть открыто сразу много. Другой пример – кэш таблиц InnoDB (подробнее об этом рассказано ниже).

Пытаясь спрогнозировать пиковое потребление памяти, необязательно предполагать наихудший сценарий. Например, если MySQL сконфигурирован из расчета не более 100 одновременных соединений, то теоретически возможно, что на всех 100 соединениях одновременно будут выполняться очень тяжелые запросы, но практически это, скорее всего, не случится. Если вы установите параметр `myisam_sort_buffer_size` равным `256M`, то при худшем варианте развития событий потребуется, по меньшей мере, 25 Гбайт памяти, но такой уровень потребления крайне маловероятен.

Чем вести подсчеты для худшего случая, лучше понаблюдать за сервером в условиях реальной нагрузки и посмотреть, сколько же памяти он потребляет. Для этого взгляните на размер виртуальной памяти процесса. Во многих UNIX-системах этот показатель отображается в столбце `VIRT` таблицы, выдаваемой командой *top*, или в столбце `VSZ`, если вы

пользуетесь командой *ps*. В следующей главе мы подробнее расскажем о том, как вести мониторинг потребления памяти.

Резервирование памяти для операционной системы

Для работы операционной системы, как и для выполнения запросов, необходимо отвести достаточно памяти. Лучшим свидетельством того, что в распоряжении ОС достаточно памяти, является отсутствие выгрузки страниц в файл подкачки на диске (см. ниже раздел «Свопинг» на стр. 418, где эта тема обсуждается более подробно.)

Вряд ли стоит резервировать под нужды операционной системы больше 1–2 Гбайт, даже если компьютер располагает очень большим объемом памяти. Добавьте небольшой запас для страховки, а если периодически запускаются задачи, потребляющие необычно много памяти (к примеру, резервное копирование), можно сделать этот запас побольше. Не добавляйте память для кэшей операционной системы, поскольку они могут быть очень велики. Обычно ОС использует в этих целях всю свободную память; далее мы рассмотрим кэши отдельно от нужд операционной системы.

Выделение памяти для кэшей

Если сервер выделен исключительно для MySQL, то вся память, не отведенная под нужды операционной системы или для обработки запросов, может быть использована в целях кэширования.

Для кэшей MySQL нужно больше памяти, чем для чего-либо другого. Кэши используются, чтобы избежать доступа к диску, который на несколько порядков медленнее, чем обращение к памяти. Операционная система может кэшировать некоторые данные в интересах MySQL (особенно в случае MyISAM), но и самому MySQL требуется много памяти.

Ниже перечислены наиболее важные кэши, которые нужно принимать во внимание в большинстве случаев:

- Кэши операционной системы для данных MyISAM
- Кэши ключей MyISAM
- Пул буферов InnoDB
- Кэш запросов

Существуют и другие кэши, но они, как правило, потребляют не так много памяти. Кэш запросов мы подробно рассматривали в предыдущей главе, поэтому далее займемся вопросом о кэшах, которые необходимы подсистемам хранения MyISAM и InnoDB для нормальной работы.

Сервер гораздо проще настроить, если используется только одна подсистема хранения. Если вы работаете исключительно с таблицами MyISAM, то InnoDB можно вообще отключить, а если только с таблицами InnoDB, то под MyISAM можно отвести минимум ресурсов (MySQL применяет таблицы типа MyISAM для своих внутренних нужд). Однако в ситуа-

ции, когда используются сразу несколько подсистем хранения, бывает очень трудно подобрать правильный баланс между ними. Лучший подход, который мы можем предложить, – высказать некую обоснованную гипотезу, а затем провести эталонное тестирование.

Кэш ключей MyISAM

Кэш ключей MySQL часто называют также *буфером ключей*. По умолчанию существует только один такой буфер, но можно создать дополнительные. В отличие от InnoDB и некоторых других подсистем хранения, MyISAM самостоятельно кэширует только индексы, но не данные (оставляя кэширование данных операционной системе). Если вы используете преимущественно MyISAM, то должны выделить под кэши ключей как можно больше памяти.

Во многих рекомендациях из этого раздела предполагается, что вы работаете только с таблицами MyISAM. При использовании MyISAM в сочетании с другой подсистемой хранения следует принимать во внимание нужды обеих подсистем.

Наиболее важен параметр `key_buffer_size`, желательно, чтобы он был на уровне между 25% и 50% от общего объема памяти, зарезервированного для кэшей. Остаток будет отведен под кэши операционной системы, в которых обычно хранятся данные, считанные из *MYD*-файлов MyISAM. В версии MySQL 5.0 для этого параметра «зашито» ограничение в 4 Гбайт вне зависимости от архитектуры. В MySQL 5.1 допустимы большие значения. Уточните этот показатель в документации к имеющейся у вас версии сервера.

По умолчанию MyISAM кэширует все индексы в буфере ключей, подразумеваемом по умолчанию, но разрешается создавать дополнительные именованные буферы. Это дает возможность хранить в памяти более 4 Гбайт индексных ключей. Чтобы создать буферы ключей с именами `key_buffer_1` и `key_buffer_2` по 1 Гбайт каждый, добавьте в конфигурационный файл такие строчки:

```
key_buffer_1.key_buffer_size = 1G
key_buffer_2.key_buffer_size = 1G
```

Теперь имеется три буфера ключей: два созданы явно, а один – по умолчанию. Команда `CACHE INDEX` позволяет установить соответствие между таблицами и кэшами. Вот как можно указать, что MySQL должна использовать кэш `key_buffer_1` для индексов по таблицам `t1` и `t2`,

```
mysql> CACHE INDEX t1, t2 IN key_buffer_1;
```

Теперь все блоки, прочитанные из индексов по этим таблицам, MySQL будет помещать в указанный буфер. Можно также заранее загрузить индексы в кэш командой `LOAD INDEX`:

```
mysql> LOAD INDEX INTO CACHE t1, t2;
```

Эту SQL-команду можно поместить в файл, выполняемый MySQL на этапе запуска. Имя файла задается с помощью параметра `init_file`; в нем может быть несколько SQL-команд, каждая в отдельной строке (комментарии не допускаются). Все индексы, которым явно не сопоставлен буфер ключей, ассоциируются с буфером по умолчанию при первом доступе к соответствующему *MYI*-файлу.

Следить за производительностью и использованием буферов ключей можно с помощью информации, которую выводят команды `SHOW STATUS` и `SHOW VARIABLES`. Эффективность использования и процент заполнения буфера вычисляются по следующим формулам:

Коэффициент попаданий в кэш

$$100 - \frac{\texttt{key\_reads} \times 100}{\texttt{key\_reads\_requests}}$$

Коэффициент заполненности буфера

$$100 - \frac{\texttt{key\_blocks\_unused} \times \texttt{key\_cache\_block\_size} \times 100}{\texttt{key\_buffer\_size}}$$

В главе 14 мы рассмотрим некоторые инструменты, в частности *innotop*, предлагающие более удобный мониторинг производительности.

Знать коэффициент попаданий в кэш полезно, но его значение может сбить с толку. Например, разница между 99% и 99,9% кажется совсем небольшой, но в действительности это десятикратное увеличение. Коэффициент попаданий зависит также от приложения: некоторые из них прекрасно работают при 95%, тогда как другие могут постоянно обращаться к диску даже при 99.9%. Если размер кэша выбран верно, то можно даже достичь уровня 99.99%.

Эмпирически обычно полезнее другой показатель: количество *непопаданий* в кэш за секунду. Предположим, что имеется один жесткий диск, способный выполнять 100 операций произвольного чтения в секунду. Пять непопаданий за этот промежуток времени еще не сделают систему ввода/вывода узким местом, но при 80 возникнут проблемы. Данное значение вычисляется по формуле:

$$\frac{\texttt{Key\_reads}}{\texttt{Uptime}}$$

Чтобы составить представление о текущей производительности, вычисляйте количество непопаданий несколько раз с интервалом от 10 до 100 секунд. Следующая команда снимает показания каждые 10 секунд:

```
$ mysqladmin extended-status -r -i 10 | grep Key_reads
```

Принимая решение об отводимом под кэши ключей объеме памяти, полезно знать, сколько места занимают MyISAM-индексы на диске. Не имеет смысла делать буферы ключей больше, чем объем данных, кото-

рый теоретически может в них находиться. В UNIX-системе узнать размер хранящих индексы файлов позволяет следующая команда:

```
$ du -sch `find /path/to/mysql/data/directory/ -name "*.MYI"`
```

Напомним, что для файлов данных, которые зачастую больше индексов, MyISAM пользуется кэшем операционной системы. Поэтому разумно оставлять для этого кэша больше памяти, чем для кэшей ключей. И, наконец, даже если у вас вообще нет таблиц типа MyISAM, все равно следует выделить с помощью параметра `key_buffer_size` хотя бы небольшой объем памяти, скажем, `32M`. Иногда MySQL использует MyISAM-таблицы для внутренних целей, например так устроены временные таблицы, создаваемые при обработке запросов с фразой `GROUP BY`.

Размер блока ключей MyISAM

Размер блока ключей важен (особенно в случае рабочей нагрузки, в которой превалируют операции записи) из-за способа взаимодействия между MyISAM, кэшем ОС и файловой системой. Если размер блока ключей слишком мал, то можно столкнуться с феноменом *записи после чтения* (*read-around writes*), то есть с такими операциями записи, которые операционная система не может выполнить, не прочитав предварительно какие-то данные с диска. Покажем, как может возникать этот феномен в предположении, что размер страницы ОС равен 4 Кбайт (типично для архитектуры x86), а размер блока ключей равен 1 Кбайт.

1. MyISAM запрашивает блок ключей размером 1 Кбайт с диска.
2. ОС считывает страницу данных размером 4 Кбайт с диска, кэширует ее, а затем передает MyISAM затребованный 1 Кбайт.
3. ОС отбрасывает закэшированные данные, замещая их какими-то другими.
4. MyISAM модифицирует блок ключей размером 1 Кбайт и просит операционную систему записать его обратно на диск.
5. ОС считывает ту же самую страницу размером 4 Кбайт с диска в свой кэш, модифицирует в ней тот килобайт, который изменил MyISAM, и записывает все 4 Кбайт обратно на диск.

Феномен записи после чтения имеет место на шаге 5, когда MyISAM просит операционную систему записать только часть страницы размером 4 Кбайт. Если бы размер блока MyISAM был равен размеру страницы ОС, то чтения на шаге 5 можно было бы избежать[1].

[1] Теоретически, если бы можно было гарантировать, что исходные 4 Кбайт данных все еще находятся в кэше операционной системы, чтение было бы не нужно. Но вы не можете контролировать, какие блоки ОС оставляет в кэше. Получить сведения о содержимом кэша позволяет инструмент *fincore*, который можно скачать со страницы *http://net.doit.wisc.edu/~plonka/fincore/*.

К сожалению, в версиях MySQL до 5.0 включительно невозможно сконфигурировать размер блока ключей. Однако, начиная с MySQL 5.1, можно избежать записи после чтения, сделав размер блока ключей MyISAM равным размеру страницы операционной системы. Величиной блока ключей управляет переменная `myisam_block_size`. Можно также задать размер блока для каждого ключа в отдельности с помощью параметра `KEY_BLOCK_SIZE` в командах `CREATE TABLE` и `CREATE INDEX`. Поскольку все ключи хранятся в одном и том же файле, то на самом деле нужно, чтобы для всех ключей размер блока был не меньше размера страницы ОС. Это позволяет избежать проблем с выравниванием, которые все равно могут привести к феномену записи после чтения (например, если для одного ключа используются блоки размером 1 Кбайт, а для другого – 4 Кбайт, то границы 4-Кбайтного блока могут не совпадать с границами страницы ОС.)

Пул буферов InnoDB

Если вы работаете преимущественно с таблицами InnoDB, то для пула буферов InnoDB, вероятно, потребуется больше памяти, чем для чего-либо другого. В отличие от кэша ключей MyISAM, в пуле буферов InnoDB кэшируются не только индексы, там также хранятся сами данные, адаптивный хеш-индекс (см. раздел «Хеш-индексы» на стр. 141), буфер вставок, блокировки и другие внутренние структуры. В InnoDB пул буферов используется также для реализации отложенных операций записи и позволяет объединить несколько таких процедур, чтобы затем выполнить их последовательно. Короче говоря, работа InnoDB *очень сильно* зависит от пула буферов, поэтому памяти в нем должно быть достаточно. В руководстве по MySQL рекомендуется отводить под пул буферов до 80% физической памяти, если сервер выделенный, а если компьютер располагает очень большим объемом памяти, то можно отдавать даже больше. Как и в случае буферов ключей MyISAM, для мониторинга производительности и использования памяти в пуле буферов можно проанализировать переменные, выводимые командами `SHOW` и инструментами типа *innotop*.

Для таблиц типа InnoDB не существует аналога команды `LOAD INDEX INTO CACHE`. Но если вы захотите «прогреть» сервер, подготовив его к высокой нагрузке, то можете отправить запросы, выполняющие полное сканирование таблицы или индекса.

Как правило, пул буферов InnoDB следует делать настолько большим, насколько позволяет доступная память. Однако в редких случаях очень большой пул (скажем, больше 50 Гбайт) может привести к длительным задержкам. Например, работа с большим пулом может замедлиться во время операций записи контрольной точки или объединения буфера вставок с деревьями вторичных индексов, а в результате блокирования иногда снижается степень конкурентности. Столкнувшись с такими проблемами, попробуйте уменьшить размер пула буферов.

Переменная `innodb_max_dirty_pages_pct` говорит InnoDB о допустимом количестве «грязных» (модифицированных) страниц в пуле буферов. Если разрешить большое количество таких страниц, то возрастает время останова InnoDB, поскольку перед остановом все «грязные» страницы нужно записать в файлы данных. Можно принудительно выполнить процедуру быстрого останова, но тогда InnoDB потребуется больше времени на восстановление при повторном запуске, так что уменьшить суммарное время останова и запуска не получится. Если вы заранее знаете, что придется останавливать сервер, присвойте этой переменной значение поменьше, дождитесь, пока поток сброса очистит пул буферов, и начинайте останов. Следить за количеством «грязных» страниц можно с помощью серверной переменной состояния `Innodb_buffer_pool_pages_dirty` или утилиты *innotop*, которая периодически выполняет команду `SHOW INNODB STATUS`.

Уменьшение переменной `innodb_max_dirty_pages_pct` еще не гарантирует, что InnoDB будет хранить в пуле буферов меньше «грязных» страниц. Она лишь управляет порогом, при котором InnoDB перестает «лениться». По умолчанию InnoDB сбрасывает «грязные» страницы на диск в отдельном фоновом потоке, который ради эффективности группирует операции записи и выполняет их последовательно. Такое поведение называется «ленивым», поскольку InnoDB откладывает сброс «грязных» страниц из пула до того момента, как понадобится место для других данных. Но если процент «грязных» страниц превысит заданный порог, то InnoDB начинает сбрасывать страницы на диск с максимально возможной скоростью, стремясь уменьшить их количество. По умолчанию значение этой переменной равно 90, то есть InnoDB будет «лениться», пока пул буферов не заполнится «грязными» страницами на 90%.

Настраивать пороговое значение имеет смысл, если вы хотите немного разнести операции записи во времени. Например, уменьшение его до 50 обычно приводит к тому, что InnoDB выполняет больше операций записи, поскольку сброс страниц производится чаще и, значит, качество группировки операций записи ухудшается. Но если рабочая нагрузка такова, что часто возникают всплески записи, то уменьшение порога помогает InnoDB лучше с ними справляться: у нее появляется больше «запасной» памяти для хранения «грязных» страниц, поэтому не приходится ждать, пока старые страницы будут сброшены на диск.

Кэш потоков

Кэш потоков содержит потоки, которые в данный момент не ассоциированы ни с одним соединением, но готовы к обслуживанию новых соединений. Если в кэше есть поток и поступает запрос на создание нового соединения, то MySQL забирает поток из кэша и передает его создаваемому соединению. Когда соединение закрывается, MySQL возвращает поток в кэш, если там есть место. Если места нет, поток уничтожается. Пока в кэше есть свободные потоки, MySQL отвечает на запросы об от-

крытии соединения очень быстро, поскольку ей не нужно создавать новый поток для обслуживания соединения.

Переменная `thread_cache_size` определяет максимальное количество потоков в этом кэше. Настраивать ее нужно лишь в случае, когда сервер получает очень много запросов на открытие соединений. Чтобы проверить, достаточен ли размер кэша, понаблюдайте за переменной состояния `Threads_created status`. Мы обычно стараемся делать кэш потоков настолько большим, чтобы в каждую секунду создавалось не более 10 новых потоков, но часто без труда удается снизить этот показатель, так что в секунду будет создаваться менее одного потока.

Разумный подход состоит в том, чтобы, понаблюдав за переменной `Threads_connected`, попытаться установить значение `thread_cache_size` достаточно большим для поглощения типичных флуктуаций рабочей нагрузки. Например, если `Threads_connected` обычно изменяется от 100 до 200, то можно сделать размер кэша равным 100. Если она изменяется от 500 до 700, то кэша размером 200 будет достаточно. Мы рассуждаем следующим образом: при 700 соединениях в кэше, вероятно, нет потоков; при 500 соединениях в кэше имеется 200 потоков, готовых к использованию, как только нагрузка вновь возрастет до 700.

В большинстве случаев делать кэш потоков очень большим нет надобности, но и уменьшение его сверх меры не дает заметной экономии памяти. Поток, который находится в кэше или «спит», обычно занимает около 256 Кбайт. Это совсем немного по сравнению с памятью, потребляемой потоком во время обработки запроса. В общем случае старайтесь поддерживать размер кэша на уровне, при котором переменная `Threads_created` не увеличивается слишком часто. Но если при таком условии объем кэша становится слишком велик (порядка нескольких тысяч), то лучше сделать его поменьше, так как некоторые операционные системы плохо справляются с чрезмерно большим количеством потоков, даже если большинство из них спит.

Кэш таблиц

Концептуально кэш таблиц похож на кэш потоков, но хранятся в нем объекты, представляющие таблицы. Каждый такой объект содержит разобранный *FRM*-файл, описывающий таблицу, и некоторые другие данные. Что именно содержится в объекте, зависит от типа таблицы (подсистемы хранения). Например, для таблиц типа MyISAM там находятся дескрипторы файла индекса и/или данных. Для объединенной таблицы в кэше, как правило, хранится несколько файловых дескрипторов, поскольку она может быть составлена из многих таблиц.

Кэш таблиц способствует повторному использованию ресурсов. Например, когда запрос обращается к таблице типа MyISAM, MySQL может вернуть файловый дескриптор из объекта в кэше, а не открывать файл заново. Кроме того, кэш таблиц позволяет избежать ряда операций вво-

да/вывода, необходимых для того, чтобы пометить MyISAM-таблицу признаком «in use» (используется) в заголовках индекса[1].

Архитектура кэша таблиц в MySQL ориентирована в большей степени на MyISAM, чем на другие подсистемы, – это одна из тех областей, в которых разделение между сервером и подсистемами хранения несовершенно, так уж сложилось исторически. Кэш таблиц не так важен для InnoDB, которая полагается на него в меньшей степени (скажем, файловые дескрипторы там не содержатся, для этой цели у InnoDB есть собственный кэш таблиц). Однако даже InnoDB выигрывает от кэширования разобранных *frm*-файлов.

В версии MySQL 5.1 кэш таблиц разделен на две части: кэш открытых таблиц и кэш определений таблиц (для их конфигурирования служат переменные `table_open_cache` и `table_definition_cache`). Таким образом, определения таблиц (разобранные *frm*-файлы) отделены от других ресурсов, к примеру, файловых дескрипторов. Открытые таблицы попрежнему локальны для соединения, но определения таблиц глобальны и могут совместно использоваться всеми соединениями. В общем случае переменной `table_definition_cache` можно присвоить значение, достаточно большое для кэширования определений всех таблиц. Если количество таблиц не исчисляется десятками тысяч, то это, наверное, проще всего.

Если переменная состояния `Opened_tables` велика или постоянно растет, то переменную `table_cache` (или `table_open_cache` в версии MySQL 5.1) следует увеличить. Единственный недостаток избыточно большого кэша таблиц заключается в том, что при наличии очень большого количества MyISAM-таблиц может замедлиться останов сервера, поскольку необходимо сбросить на диск блоки ключей и пометить все таблицы как «не открытые». По той же самой причине команда `FLUSH TABLES WITH READ LOCK` может выполняться довольно долго.

Если вы получаете сообщение об ошибке, в котором говорится, что MySQL не может открыть новые файлы (понять, что означает код ошибки, поможет утилита *perror*), то, возможно, следует увеличить количество файлов, которые MySQL может держать открытыми. Для этого предназначена серверная переменная `open_files_limit` в файле *my.cnf*.

---

[1] Идея «открытой таблицы» может вызвать путаницу. MySQL считает, что таблица открывается каждый раз, как к ней производятся обращения из одновременно выполняемых запросов, и даже из одного запроса, если таблица встречается в нем более одного раза, например в случае подзапроса или рефлексивного соединения. В индексных файлах MyISAM имеется счетчик, который MyISAM увеличивает при каждом открытии и уменьшает при закрытии. Это позволяет подсистеме хранения определить, что таблица не была закрыта корректно: это так, если при первом открытии таблицы счетчик отличен от нуля.

Кэши потоков и таблиц потребляют не так уж много памяти, а польза от них велика, поскольку они экономят ресурсы. Хотя создание потока и открытие файла – не очень дорогие операции по сравнению с тем, что еще приходится делать серверу MySQL, в условиях рабочей нагрузки с высокой степенью конкурентности накладные расходы могут суммироваться и становиться заметными. Таким образом, кэширование потоков и таблиц способно повысить эффективность.

Словарь данных InnoDB

В подсистеме хранения InnoDB имеется отдельный кэш, который называется *кэшем определений таблиц* или *словарем данных*; он не допускает конфигурирования. Открывая таблицу, InnoDB помещает соответствующий ей объект в словарь данных. На одну таблицу может отводиться 4 Кбайта или больше (хотя в версии MySQL 5.1 требуется гораздо меньше памяти). Объект не удаляется из словаря данных при закрытии таблицы.

С точки зрения производительности серьезной проблемой – помимо требований к ресурсам – является открытие и вычисление статистики для таблиц, что требует выполнения достаточно большого числа операций ввода/вывода. В отличие от MyISAM, InnoDB не хранит статистику в таблицах постоянно, а высчитывает заново при каждом запуске. В текущих версиях MySQL эта операция защищена глобальным мьютексом, поэтому не распараллеливается. Если в вашей базе много таблиц, то на полную загрузку и «прогрев» сервера может уйти несколько часов, в течение которых он только и будет делать, что ждать завершения последовательных операций ввода/вывода. Мы упоминаем этот факт просто для сведения, сделать тут вы ничего не можете. Обычно это составляет проблему лишь том случае, когда имеется очень много (тысячи или десятки тысяч) больших таблиц; в подобной ситуации процесс занимается в основном вводом/выводом.

При использовании параметра `innodb_file_per_table` (будет описан ниже в разделе «Конфигурирование табличного пространства» на стр. 363) действует отдельное ограничение на количество одновременно открытых *idb*-файлов. Оно контролируется подсистемой InnoDB, а не сервером и задается параметром `innodb_open_files`. InnoDB открывает файл не так, как MyISAM. Если MyISAM хранит файловые дескрипторы открытых таблиц в кэше, то в InnoDB нет прямой связи между открытыми таблицами и открытыми файлами. Эта подсистема использует один глобальный файловый дескриптор для каждого *idb*-файла. Если вы можете себе это позволить, то лучше присвоить параметру `innodb_open_files` настолько большое значение, чтобы сервер мог держать все *idb*-файлы открытыми одновременно.

Настройка ввода/вывода в MySQL

Ряд конфигурационных параметров управляет тем, как MySQL синхронизирует данные в памяти и на диске и выполняет восстановление. Они могут существенно повлиять на производительность, так как относятся к дорогостоящим операциям ввода/вывода. Кроме того, они определяют компромисс между производительностью и защитой данных. В общем случае стремление записывать данные на диск немедленно, чтобы не оставалось шанса на рассинхронизацию, обходится слишком дорого. Если вы готовы пойти на риск, обусловленный тем, что запись на диск еще не обеспечивает постоянного хранения, то можно будет повысить степень конкурентности и сократить время ожидания ввода/вывода. Но определить допустимый уровень риска вам придется самостоятельно.

Настройка ввода/вывода для MyISAM

Начнем с рассмотрения того, как MyISAM выполняет ввод/вывод для индексов. Обычно изменение индекса сбрасывается на диск после каждой записи. Но если вы производите много модификаций в таблице, то будет быстрее сгруппировать операции записи вместе.

Один из способов добиться этого – воспользоваться командой `LOCK TABLES`, которая откладывает запись до момента разблокировки таблиц. Этот прием способствует повышению производительности, так как позволяет точно управлять тем, какие операции записи откладываются и когда происходит сброс на диск. Можно отложить запись именно для интересующих вас команд.

Запись в индекс можно отложить также с помощью переменной `delay_key_write`. В этом случае блоки из буфера ключей не сбрасываются на диск до момента закрытия таблицы[1]. Допустимы следующие значения:

`OFF`

MyISAM сбрасывает измененные блоки из буфера ключей после каждой записи, если только таблица не блокирована командой `LOCK TABLES`.

`ON`

Включен режим отложенной записи ключей, но только для таблиц, созданных с параметром `DELAY_KEY_WRITE`.

`ALL`

Для всех таблиц типа MyISAM используется отложенная запись ключей.

[1] Таблица может быть закрыта по нескольким причинам. Например, сервер может закрыть таблицу, поскольку в кэше таблиц кончилось место, или кто-то другой выполнил команду `FLUSH TABLES`.

В некоторых случаях отложенная запись бывает кстати, но обычно она не приводит к резкому возрастанию производительности. Наиболее полезна она в ситуации, когда размер данных мал, коэффициент попадания в кэш при чтении высокий, а при записи – низкий. К тому же у этого режима есть ряд недостатков.

- Если сервер аварийно завершает работу, а блоки не были сброшены на диск, то индекс будет испорчен.
- Если было отложено много операций записи, то MySQL потратит больше времени на закрытие таблицы, поскольку вынуждена ждать завершения записи буферов на диск. В версии MySQL 5.0 это приводит к длительным блокировкам доступа к кэшу таблиц.
- По тем же причинам команда `FLUSH TABLES` может занимать много времени. А это, в свою очередь, может увеличить время выполнения команды `FLUSH TABLES WITH READ LOCK` при снятии мгновенного снимка для менеджера логических томов (LVM), а также других операций резервного копирования.
- Не сброшенные «грязные» блоки в буфере ключей могут не оставить места для новых блоков, считываемых с диска. В таком случае выполнение запроса будет приостановлено на время, пока MyISAM не освободит достаточно места в буфере ключей.

Помимо настройки ввода/вывода для индексов MyISAM можно сконфигурировать метод восстановления после порчи данных. Параметр `myisam_recover` управляет алгоритмом поиска и исправления ошибок. Его можно задавать как в конфигурационном файле, так и в командной строке. Чтобы просмотреть (но не изменить) его, воспользуйтесь следующей SQL-командой (это не опечатка – системная переменная действительно называется не так, как параметр в командной строке):

```
mysql> SHOW VARIABLES LIKE 'myisam_recover_options';
```

Если этот режим включен, то MySQL проверяет, не испорчены ли таблицы в момент открытия, и при обнаружении ошибок тут же исправляет их. Параметр может принимать следующие значения:

`DEFAULT` (*или не задан*)

MySQL попытается исправить таблицу, помеченную как сбойная, или не помеченную как корректно закрытая. Это режим по умолчанию, в котором никаких других действий при восстановлении не предпринимается. В отличие от большинства других переменных, значение `DEFAULT` не означает, что нужно восстановить режим, указанный на этапе компиляции, а интерпретируется просто как «не задано».

`BACKUP`

Заставляет MySQL записать резервную копию файла данных в файл с расширением *BAK*, который позже можно будет исследовать.

FORCE

Требует продолжить восстановление, даже если в *MYD*-файле будет потеряно более одной строки.

QUICK

Пропускает фазу восстановления, если только отсутствуют блоки удаления. Так называются блоки, занятые удаленными строками. Они все еще занимают место в файле, но могут быть повторно использованы во время выполнения команды INSERT. Этот режим может быть полезен, так как восстановление большой MyISAM-таблицы иногда занимает весьма длительное время.

Разрешается перечислять несколько значений через запятую. Например, BACKUP,FORCE принудительно продолжает восстановление и создает резервную копию.

Мы рекомендуем включать этот параметр, особенно если имеется всего несколько небольших таблиц MyISAM. Запускать сервер с испорченными таблицами MyISAM опасно, поскольку иногда это приводит к дальнейшей порче данных и даже аварийному останову сервера. Однако при наличии больших таблиц автоматическое восстановление может оказаться непрактичным: сервер должен проверить и исправить любую таблицу MyISAM в момент ее открытия, что, конечно, неэффективно. На протяжении этого времени MySQL обычно приостанавливает работу всех соединений. Если количество таблиц MyISAM велико, то, пожалуй, лучше воспользоваться менее навязчивой процедурой: выполнить команды CHECK TABLES и REPAIR TABLES после запуска сервера. В любом случае проверка и исправление таблиц очень важны.

Еще одна полезная настройка MyISAM – доступ к файлам данных в режиме проецирования в память. При этом MyISAM обращается к *MYD*-файлам непосредственно через кэш страниц операционной системы, обходясь без дорогостоящих системных вызовов. Начиная с версии MySQL 5.1 включить режим проецирования в память можно с помощью параметра myisam_use_mmap. В более старых версиях проецирование в память разрешено только для сжатых MyISAM-таблиц.

Настройка ввода/вывода для InnoDB

Подсистема хранения InnoDB сложнее, чем MyISAM. Как следствие, она позволяет управлять не только способом восстановления, но и тем, как открываются файлы и сбрасываются на диск данные кэшей. Это существенно повышает скорость работы и общую производительность. Процесс восстановления в InnoDB полностью автоматический и в обязательном порядке выполняется в момент запуска InnoDB, хотя у вас остается возможность повлиять на то, какие действия при этом предпринимаются. Более подробно об этом рассказано в главе 11.

Даже если оставить в стороне вопрос о восстановлении и считать, что никаких аварий не было и вообще все нормально, у вас в любом случае

остается множество возможностей для конфигурации InnoDB. В этой подсистеме имеется сложная цепочка буферов и файлов, спроектированная так, чтобы добиться высокой производительности и гарантировать свойства ACID. При этом каждое звено этой цепочки конфигурируемо. На рис. 6.1 показаны все файлы и буферы.

Из наиболее важных настроек, которые имеет смысл изменять, отметим размер журнала InnoDB, способ сброса журнального буфера на диск и то, как выполняется ввод/вывод.

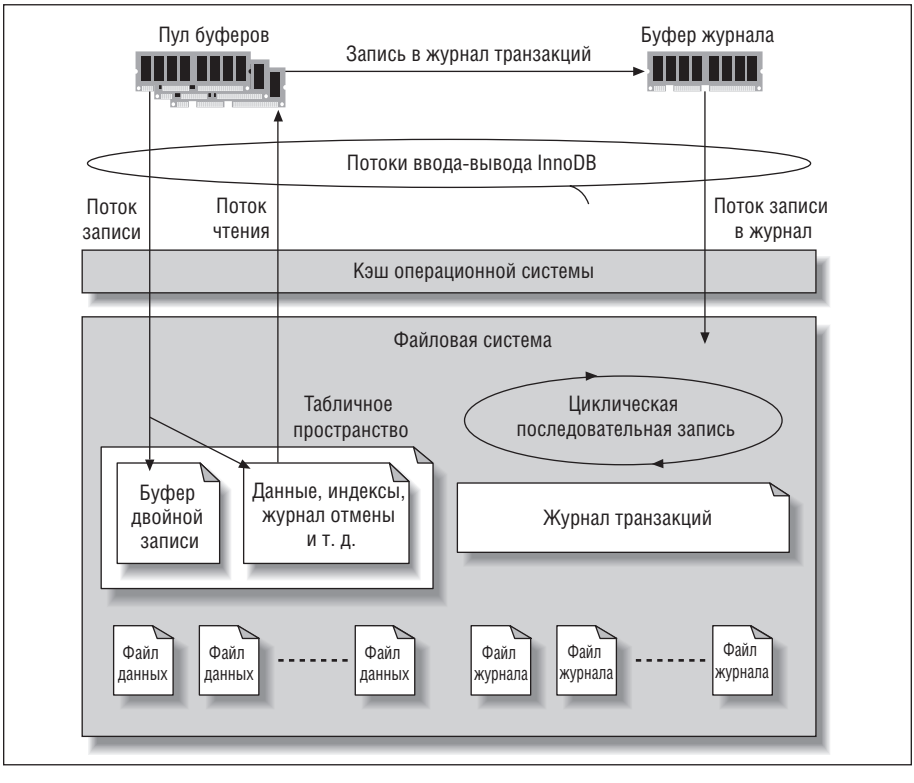


Рис. 6.1. *Буферы и файлы InnoDB*

Журнал транзакций InnoDB

В InnoDB журнал служит для того чтобы уменьшить стоимость фиксации транзакций. Вместо сброса пула буферов на диск после фиксации каждой транзакции InnoDB записывает транзакции в журнал. Изменения данных и индексов, произведенные внутри транзакции, часто относятся к разрозненным местам в табличном пространстве, поэтому для сброса потребуется помещать их в несмежные области диска. Как правило, ввод/вывод с произвольной выборкой гораздо дороже последо-

вательного из-за того, что система должна тратить время на позиционирование головки диска.

В InnoDB журнал используется для превращения произвольного ввода/вывода в последовательный. После того как запись в журнал произведена, транзакцию можно считать долговечной (одно из свойств ACID), пусть даже изменения еще не записаны в файлы данных. Если случится какая-то авария (например, пропадет питание), то InnoDB сможет воспроизвести журнал и восстановить зафиксированные транзакции.

Разумеется, InnoDB в конечном итоге должна записать изменения в файлы данных, поскольку размер журнала фиксирован. Запись в журнал производится циклически: по достижении конца журнала происходит переход в начало. InnoDB не может затереть запись журнала, если соответствующие ей изменения еще не были внесены в файлы данных, поскольку при этом была бы уничтожена зафиксированная, а следовательно долговечная транзакция.

В InnoDB имеется фоновый поток, который упорядочивает сброс изменений в файлы данных. Этот поток умеет группировать операции записи так, чтобы они выполнялись последовательно, с целью повышения его эффективности. Таким образом, журнал транзакций преобразует произвольный ввод/вывод, превращая его в преимущественно последовательный ввод/вывод с записью в файл журнала и файлы данных. Выполнение сброса в фоновом режиме ускоряет обработку запроса и помогает сгладить влияние всплесков нагрузки на подсистему ввода/вывода.

Полный размер файлов журнала, задаваемый параметрами `innodb_log_file_size` и `innodb_log_files_in_group`, очень важен с точки зрения производительности записи. Полный размер равен сумме размеров всех файлов. По умолчанию создается два файла по 5 Мбайт, то есть полный размер равен 10 Мбайт. Для высокой нагрузки этого явно недостаточно. Верхний предел полного размера составляет 4 Гбайт, но, как правило, для рабочих режимов с большим числом операций записи хватает нескольких сотен мегабайтов (быть может, 256 Мбайт). В следующих разделах объясняется, как выбрать размер, отвечающий вашей рабочей нагрузке.

В InnoDB несколько файлов образуют единый циклический журнал. Обычно не требуется изменять принимаемое по умолчанию количество журналов, настраивается лишь размер каждого файла. Чтобы изменить размер файла журнала, штатно остановите MySQL, переместите старые журналы в другое место, измените конфигурацию и перезапустите сервер. Очень важно остановить MySQL корректно, иначе в старых журналах останутся записи, которые нужно будет применить к файлам данных! Перед повторным запуском сервера загляните в журнал ошибок MySQL. После успешного перезапуска старые журналы можно будет удалить.

Размер файла журнала и буфер журнала

Чтобы определить идеальный размер журналов, нужно понимать, что за меньшую нагрузку, создаваемую типичными операциями изменения данных, придется расплачиваться бóльшим временем, необходимым для восстановления после аварийного останова. Если журнал слишком мал, то InnoDB придется чаще записывать контрольные точки, то есть количество записей в журнал увеличится. В предельном случае запрос на запись может быть приостановлен до завершения процесса выгрузки в файлы данных из-за того, что в журнале не осталось места. С другой стороны, если журнал слишком велик, то InnoDB придется выполнять много работы во время восстановления после некорректного завершения работы, поэтому время данной процедуры увеличится.

Время восстановления зависит также от объема данных и типа нагрузки. Предположим, что имеется терабайт данных и пул буферов размером 16 Гбайт, а полный размер журнала составляет 128 Мбайт. Если количество «грязных» страниц (данные в которых модифицированы, но еще не сброшены на диск) в пуле буферов велико и они равномерно распределены по всему терабайту, то восстановление после сбоя может занять много времени. InnoDB должна будет просканировать весь журнал, проанализировать файлы данных и применить к ним необходимые изменения. Это сколько же потребуется читать и писать! С другой стороны, если изменения локализованы – скажем, частым обновлениям подвергается только несколько гигабайтов, – то восстановление может пройти быстро, даже если файлы данных и журнала очень велики. Время, затраченное на этот процесс, зависит также от объема типичной модификации, который определяется средней длиной строки данных. Чем короче строки, тем больше модификаций умещается в журнал, поэтому при восстановлении InnoDB придется воспроизводить больше модификаций.

При модификации любых данных InnoDB помещает запись об изменении в *буфер журнала*, который хранится в памяти. InnoDB сбрасывает его в файлы журналов на диске в трех случаях: когда буфер заполняется, когда фиксируется транзакция или раз в секунду – в зависимости от того, что произойдет раньше. По умолчанию размер буфера равен 1 Мбайт, а его увеличение может улучшить производительность ввода/вывода в случае больших транзакций. Размером буфера журнала управляет переменная `innodb_log_buffer_size`.

Необязательно делать буфер очень большим. Рекомендуемый размер составляет от 1 до 8 Мбайт, и этого более чем достаточно, если только вы не вставляете много записей с гигантскими BLOB'ами. Записи в журнале очень компактны по сравнению с обычными данными InnoDB. Поскольку они не используют страницы, как единицу хранения, то и не нужно расходовать место на запись целых страниц. Кроме того, InnoDB всеми силами старается сделать записи в журнале как можно короче. Иногда в ней хранится только номер функции на языке C и ее параметры!

Следить за производительностью ввода/вывода в журнал и за его буфером позволяет секция LOG в результате, формируемом командой SHOW INNODB STATUS, а также переменная состояния Innodb_os_log_written. Хорошее эвристическое правило таково: наблюдайте за этой переменной с интервалом от 10 до 100 секунд и обращайте внимание на пиковые значения. На основании этой информации можно сделать вывод о том, насколько удачно выбран размер буфера журнала. Например, если в пиковые периоды в журнал записывается 100 Кбайт в секунду, то буфера размером 1 Мбайт хватит за глаза.

Эту же метрику можно использовать для выбора подходящего размера файлов журнала. Если в пиковый период пишется 100 Кбайт в секунду, то журнала размером 256 Мбайт хватит для хранения записей, по крайней мере, за 2560 секунд, а этого, скорее всего, достаточно. Дополнительную информацию о том, как вести мониторинг и интерпретировать состояние журнала и его буфера, см. в разделе «Команда SHOWINNODB STATUS» на стр. 691.

Как InnoDB сбрасывает буфер журнала

Когда InnoDB сбрасывает буфер в файлы журнала на диске, она блокирует доступ к буферу с помощью мьютекса, переписывает данные на диск вплоть до нужной точки, а затем перемещает оставшиеся записи в начало буфера. Может случиться, что к моменту освобождения мьютекса скопится несколько транзакций, готовых к записи в журнал. В InnoDB имеется механизм групповой фиксации, позволяющий записать их все в журнал одной операцией ввода/вывода, но появление двоичного журнала в версии MySQL 5.0 сделало его неработоспособным.

Буфер журнала *обязательно* должен быть сброшен на устройство постоянного хранения для обеспечения долговечности зафиксированных транзакций. Если производительность волнует вас больше долговечности, то можно изменить параметр innodb_flush_log_at_trx_commit, который контролирует, куда и как часто сбрасывается буфер журнала. Допустимы следующие значения:

0 Писать буфер в файл журнала и сбрасывать журнал на устройство постоянного хранения (диск) раз в секунду, но ничего не делать в момент фиксации транзакции.

1 Писать буфер в файл журнала и сбрасывать его на устройство постоянного хранения при каждой фиксации транзакции. Этот (самый безопасный) режим принимается по умолчанию, он гарантирует, что ни одна зафиксированная транзакция не будет потеряна, если только диск или операционная система не делают операцию сброса «фиктивной».

2 Писать буфер в файл журнала при каждой фиксации, но не сбрасывать его на устройство постоянного хранения. Тем не менее, этот режим не отменяет сброс на устройство постоянного хранения один

раз в секунду. Самое важное отличие от режима 0 (из-за которого режим 2 предпочтительнее) состоит в том, что в режиме 2 транзакции не теряются в случае аварийного завершения процесса MySQL. Однако, если «падает» весь сервер или пропадает питание, потеря транзакций все-таки возможна.

Важно понимать различие между *записью* буфера в файл журнала и *сбросом* журнала на устройство постоянного хранения. В большинстве операционных систем запись буфера в журнал сводится к простому копированию данных из буфера в памяти InnoDB в кэш операционной системы, который также находится в памяти. Никакой записи на реальное устройство при этом не происходит. Поэтому режимы 0 и 2 *обычно* приводят к утрате не более «одной секунды данных» в случае сбоя или пропадания питания, поскольку на протяжении этого времени информация, возможно, существует только в кэше операционной системы. Мы говорим «обычно», потому что InnoDB старается сбрасывать файл журнала на диск примерно раз в секунду при любых обстоятельствах, но в некоторых ситуациях можно потерять транзакции более чем за одну секунду, например когда поток сброса задерживается (gets stalled).

Сброс же журнала на устройство постоянного хранения означает, что InnoDB просит операционную систему действительно переписать данные из своего кэша *на диск*. Это блокирующий вызов, который не возвращает управление, пока данные не окажутся полностью выгруженными. Поскольку запись на диск – медленная операция, то в случае, когда параметр innodb_flush_log_at_trx_commit равен 1, количество транзакций, которые InnoDB способна зафиксировать в секунду, может резко уменьшиться. Современные высокоскоростные накопители[1] могут записать лишь пару сотен реальных дисковых транзакций[2] в секунду из-за ограничений на скорость вращения диска и время позиционирования головки.

Иногда контроллер жесткого диска или операционная система подтверждают сброс, но на деле лишь копируют данные *еще* в один кэш, например в собственный кэш жесткого диска. Это работает быстрее, но очень опасно, поскольку данные могут быть потеряны, если внезапно пропадет питание. Такая ситуация даже хуже, чем установка параметра innodb_flush_log_at_trx_commit в любое значение, кроме 1, поскольку может привести к порче данных, а не просто к потере транзакций.

Присваивание параметру innodb_flush_log_at_trx_commit значения, отличного от 1, может привести к потере транзакций. Однако если дол-

---

[1] Мы говорим об шпиндельных накопителях с вращающимися пластинами, а не о твердотельных дисках, у которых характеристики производительности совершенно иные.

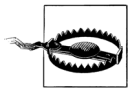
[2] **Под реальной дисковой транзакцией авторы подразумевают случайную запись на диск.** – *Прим. науч. ред.*

говечность (буква D в аббревиатуре ACID) вас не очень тревожит, то, возможно, вы сочтете другие значения полезными. Быть может, вас привлекают такие средства InnoDB, как кластерные индексы, устойчивость к порче данных и блокировка на уровне строк. Не так уж редки случаи, когда InnoDB используют вместо MyISAM исключительно из соображений производительности.

Наилучшая конфигурация для высокопроизводительных транзакционных приложений достигается, когда `innodb_flush_log_at_trx_commit` оставляют равным `1`, а файлы журналов помещают на том RAID-массива с резервным электропитанием кэша на запись. Это и безопасно, и быстро. Дополнительную информацию о RAID-массивах см. в разделе «Оптимизация производительности с помощью RAID» на стр. 396.

Как InnoDB открывает и сбрасывает файлы журнала и данных

Параметр `innodb_flush_method` позволяет указать, как InnoDB взаимодействует с файловой системой. Вопреки своему имени этот параметр относится к чтению, а не к записи данных. Значения, которые он может принимать в системах Windows и прочих, взаимно исключают друг друга: `async_unbuffered`, `unbuffered` и `normal` применимы только к Windows, другие значения в Windows не допускаются. По умолчанию на платформе Windows принимается значение `unbuffered`[1], а на всех остальных платформах — `fdatasync` (если команда `SHOW GLOBAL VARIABLES` показывает пустое значение этой переменной, значит, для нее действует значение по умолчанию).



При изменении режима ввода/вывода в InnoDB может существенно измениться производительность в целом. Не забывайте тщательно тестировать!

Параметр может принимать следующие значения:

`fdatasync`

Значение по умолчанию во всех системах, кроме Windows. InnoDB вызывает `fsync()` для сброса файлов данных и журнала.

Обычно InnoDB вызывает `fsync()`, а не `fdatasync()` несмотря на то, что название вроде бы свидетельствует о противоположном. Функция `fdatasync()` аналогична `fsync()`, но сбрасывает лишь данные файла, а не его метаданные (время последней модификации и т. д.). Поэтому `fsync()` выполняет больше операций ввода/вывода. Однако разработчики InnoDB, будучи очень осторожными, обратили внимание на то, что в некоторых случаях `fdatasync()` приводит к порче данных. Код InnoDB определяет, какие методы можно использовать безопасно;

[1] Документация говорит о том, что в Windows по умолчанию установлен режим async_unbuffered и он не может быть изменен. — *Прим. науч. ред.*

некоторые опции устанавливаются на этапе компиляции, другие – на этапе выполнения. InnoDB применяет самый быстрый из безопасных методов.

Недостаток `fsync()` заключается в том, что операционная система буферизует, по крайней мере, некоторые данные в собственном кэше. Теоретически это можно считать расточительной двойной буферизацией, поскольку InnoDB управляет своими буферами более интеллектуально, чем ОС. Но конечный эффект очень сильно зависит от операционной и файловой систем. Двойная буферизация – это необязательно плохо, если позволяет файловой системе более точно планировать и группировать операции ввода/вывода. Некоторые файловые и операционные системы умеют накапливать процедуры записи и выполнять их одним пакетом, переупорядочивать для повышения эффективности и осуществлять вывод на несколько устройств параллельно. Кроме того, они иногда реализуют упреждающее чтение, например просят диск заранее прочитать следующий по порядку блок, если уже поступали запросы на чтение нескольких последовательных блоков.

Иногда такие оптимизации помогают, иногда нет. Если вам интересно, как работает конкретная версия `fsync()`, можете прочитать страницу руководства `fsync(2)`.

Параметр `innodb_file_per_table` приводит к вызову `fsync()` для каждого файла в отдельности. Поэтому записи в несколько таблиц невозможно объединить в одну операцию ввода/вывода. Следовательно, InnoDB может вынужденно увеличивать общее количество операций `fsync()`.

`O_DIRECT`

InnoDB в зависимости от системы устанавливает флаг `O_DIRECT` или вызывает функцию `directio()` для файлов данных. Этот режим не распространяется на файлы журналов, и доступен не на всех UNIX-системах. Но, по крайней мере, GNU/Linux, FreeBSD и Solaris (начиная с поздних выпусков версии 5.0) его поддерживают. В отличие от флага `O_DSYNC`, он относится к операциям чтения и записи.

В этом режиме для сброса файлов на диск также используется функция `fsync()`, но операционной системе дается указание не кэшировать данные и не прибегать к опережающему чтению. Тем самым кэш операционной системы полностью отключается, и все операции чтения и записи направляются напрямую устройству хранения во избежание двойной буферизации.

В большинстве систем это реализуется путем обращения к системному вызову `fcntl()` для установки флага `O_DIRECT` на дескрипторе файла, так что с деталями вы можете ознакомиться на странице руководства `fcntl(2)`. В ОС Solaris данный режим подразумевает вызов функции `directio()`.

Если упреждающее чтение осуществляется на уровне RAID-контроллера, то отменить его с помощью этого значения параметра не удастся. Он отключает упреждающее чтение только на уровне операционной или файловой системы.

Как правило, в режиме O_DIRECT не следует отключать кэш записи RAID-контроллера, поскольку в типичной ситуации только это и позволяет поддерживать приемлемую производительность. Задание флага O_DIRECT в случае, когда между InnoDB и физическим устройством нет никакого буфера, например при отключенном кэше записи на RAID-контроллере, может привести к серьезному падению производительности.

В этом режиме может заметно возрасти время «прогрева» сервера, особенно если кэш операционной системы очень велик. Кроме того, при небольшом пуле буферов (например, если размер по умолчанию не изменялся) сервер может работать гораздо медленнее, чем в случае буферизованного ввода/вывода. Это связано с тем, что операционная система не «придет на выручку», сохраняя часть данных в собственном кэше. Если нужные данные отсутствуют в пуле буферов, то InnoDB будет вынуждена читать их прямо с диска.

Этот режим не влечет дополнительных накладных расходов при использовании совместно с параметром innodb_file_per_table.

O_DSYNC

В данном режиме при обращении к системному вызову open() для файлов журнала устанавливается флаг O_SYNC. При этом все операции записи становятся синхронными; иными словами, функция записи не возвращает управление, пока данные не будут перенесены на диск. Указанный режим не распространяется на файлы данных.

Разница между флагами O_SYNC и O_DIRECT заключается в том, что O_SYNC не отключает кэширование на уровне операционной системы. Поэтому он не устраняет двойную буферизацию и не приводит к записи прямо на диск. Если задан флаг O_SYNC, то операция записи сначала модифицирует данные в кэше, а потом их сохраняет.

Хотя синхронная запись с помощью O_SYNC выглядит очень похоже на то, что делает функция fsync(), реализация может существенно отличаться как на уровне операционной системы, так и на аппаратном уровне. При использовании O_SYNC операционная система может передать флаг «применять синхронный ввод/вывод» на нижележащий аппаратный уровень, потребовав тем самым, чтобы устройство не применяло кэширование. С другой стороны, fsync() говорит операционной системе, что нужно сбросить модифицированные буферы на устройство, а затем устройству посылаются команды сбросить собственные кэши (в тех случаях, когда это имеет смысл), чтобы данные гарантированно оказались записанными на физический носитель. Еще одно различие состоит в том, что при наличии фла-

га O_SYNC каждый вызов write() или pwrite() синхронизирует данные, и до окончания синхронизации не возвращает управление, блокируя тем самым вызывающий процесс. Напротив, запись без флага O_SYNC с последующим вызовом fsync() позволяет накапливать операции записи в кэше (при этом каждая операция завершается быстро), а потом сбрасывать их на диск пакетом.

И снова, вопреки названию, этот параметр приводит к установке флага O_SYNC, а не O_DSYNC, поскольку разработчики InnoDB обнаружили ошибки в реализации последнего. Различия во флагах O_SYNC и O_DSYNC аналогичны различиям в функциях fsync() и fdatasync(): O_SYNC синхронизирует данные и метаданные, а O_DSYNC – только данные.

async_unbuffered

Этот режим подразумевается по умолчанию в Windows. В нем InnoDB для большинства операций записи использует небуферизованный ввод/вывод с одним исключением: если параметр innodb_flush_log_at_trx_commit равен 2, то при записи в файлы журнала ввод/вывод буферизован.

В этом режиме InnoDB использует платформенный механизм асинхронного ввода/вывода (с перекрытием) для чтения и записи при работе в ОС Windows 2000, XP и более поздних. В предыдущих редакциях Windows InnoDB применяет собственный механизм асинхронного ввода/вывода, который реализован с помощью потоков.

unbuffered

Применяется только в Windows. Аналогичен режиму async_unbuffered, но платформенный механизм асинхронного ввода/вывода не используется.

normal

Применяется только в Windows. InnoDB не использует ни платформенный механизм асинхронного ввода/вывода, ни небуферизованный ввод/вывод.

nosync *и* littlesync

Только для экспериментального использования. Эти режимы не документированы, так что применять их в промышленном режиме небезопасно.

Если ваш RAID-контроллер оборудован кэшем записи с резервным батарейным питанием, то мы рекомендуем использовать режим O_DIRECT. В противном случае наилучшим выбором будет значение по умолчанию или O_DIRECT, хотя это и зависит от конкретного приложения.

В Windows и только в Windows можно сконфигурировать несколько потоков ввода/вывода. Если параметр innodb_file_io_threads больше 4, то InnoDB создаст дополнительные потоки чтения и записи для ввода/вывода данных. Но существует только один поток буфера вставок и один

поток записи в журнал, поэтому значение 8, например, означает, что будут созданы поток буфера вставок, поток записи в журнал, три потока чтения и три потока записи.

Табличное пространство InnoDB

InnoDB хранит данные в *табличном пространстве*, которое представляет собой некую виртуальную файловую систему, охватывающую один или несколько файлов на диске. Табличное пространство в InnoDB используется для разных целей, а не только для хранения таблиц и индексов. В нем же находятся журнал отмены (старые версии строк), буфер вставок, буфер двойной записи (описывается ниже) и другие внутренние структуры.

Конфигурирование табличного пространства

Файлы, помещаемые в табличное пространство, перечисляются в конфигурационном параметре `innodb_data_file_path`. Все они будут находиться в каталоге, который задается параметром `innodb_data_home_dir`. Например:

```
innodb_data_home_dir = /var/lib/mysql/
innodb_data_file_path = ibdata1:1G;ibdata2:1G;ibdata3:1G
```

В результате создается табличное пространство размером 3 Гбайт, содержащее три файла. Иногда задают вопрос, можно ли использовать несколько файлов для распределения нагрузки на несколько накопителей, например так:

```
innodb_data_file_path = /disk1/ibdata1:1G;/disk2/ibdata2:1G;...
```

При этом файлы действительно помещаются в разные каталоги, которые в данном случае находятся на разных дисках, но InnoDB конкатенирует файлы друг за другом. Поэтому в данном случае никакого реального выигрыша вы не получите. InnoDB сначала будет писать в первый файл, потом, когда он заполнится, – во второй и т. д., то есть нагрузка не распределяется так, как хотелось бы для обеспечения высокой производительности. Для этой цели лучше использовать RAID-контроллер.

Чтобы табличное пространство могло расти, когда место заканчивается, можно сделать последний файл автоматически расширяемым:

```
...ibdata3:1G:autoextend
```

По умолчанию создается один автоматически расширяемый файл размером 10 Мбайт. Если вы решите сделать файл автоматически расширяемым, то имеет смысл ограничить размер табличного пространства сверху, поскольку, достигнув определенного размера, файл уже не уплотняется (doesn't shrink). Например, в следующем примере размер автоматически расширяемого файла ограничен 2 Гбайт:

```
...ibdata3:1G:autoextend:max:2G
```

Обслуживание одного табличного пространства может вызвать сложности, особенно если оно автоматически расширяется, а вам нужно освободить место (по этой причине мы рекомендуем отключать режим автоматического расширения). В данном случае единственная возможность высвободить место – сформировать дамп данных, остановить MySQL, удалить все файлы, изменить конфигурацию, перезапустить сервер, позволить InnoDB создать новые пустые файлы и, наконец, восстановить в них данные. InnoDB очень строго относится к своему табличному пространству – вы не можете просто взять и удалить файлы или изменить их размеры. InnoDB не запустится, если обнаружит, что табличное пространство повреждено. Так же трепетно InnoDB относится к файлам журнала. Если вы привыкли без особых раздумий перемещать файлы MyISAM, будьте осторожны!

Параметр `innodb_file_per_table` позволяет сконфигурировать InnoDB с одним файлом на таблицу (начиная с версии MySQL 4.1). Данные хранятся в каталоге базы данных в файлах с именами вида *tablename.ibd*. Так проще освобождать место при удалении таблицы, к тому же этот режим полезен для распределения таблиц по разным дискам. Однако в случае хранения данных в нескольких файлах больше места растрачивается впустую, поскольку вы обмениваете внутреннюю фрагментацию в одном табличном пространстве на неиспользуемое место в IBD-файлах. Эта проблема больше касается маленьких таблиц, так как размер страницы в InnoDB составляет 16 Кбайт. Даже если в таблице хранится всего 1 Кбайт данных, на диске она все равно будет занимать 16 Кбайт.

Но и в режиме `innodb_file_per_table` главное табличное пространство все равно необходимо для файлов отмены и других системных данных. Если в нем не хранится информация, оно будет меньше по размерам, но, тем не менее, мы рекомендуем отключать автоматическое расширение, так как невозможно уплотнить файл без перезагрузки всех данных. Кроме того, вы в любом случае не сможете перемещать таблицы, делать их резервные копии и восстанавливать путем простого копирования файлов. Это возможно, но требует некоторых дополнительных шагов, а переносить таблицы на другой сервер простым копированием недопустимо в принципе. Дополнительную информацию по этому поводу см. в разделе «Восстановление из физических файлов» на стр. 617.

Некоторым режим `innodb_file_per_table` нравится просто потому, что он предоставляет дополнительные средства управления и делает структуру базы более наглядной. Например, гораздо быстрее определить размер таблицы, взглянув на соответствующий файл, нежели выполнять команду `SHOW TABLE STATUS`, которая должна заблокировать и просмотреть пул буферов, чтобы понять, сколько страниц выделено таблице.

Следует также отметить, что файлы InnoDB необязательно хранить в традиционной файловой системе. Как и многие другие СУБД, InnoDB позволяет размещать их на неформатированном (raw) разделе диска. Однако современные файловые системы умеют эффективно работать

с достаточно большими объектами, поэтому в таком режиме нет особого смысла. Использование неформатированных устройств может дать прирост производительности на несколько процентов, но мы не считаем, что такой мизерный выигрыш оправдывает неудобства, связанные с невозможностью манипулировать данными, как файлами. Если они находятся на неформатированном устройстве, то о командах *mv*, *cp* и прочих можно забыть. Мы также считаем, что средства снятия мгновенных снимков, например те, что предлагает менеджер логических томов (LVM) в системе GNU/Linux, – это колоссальное удобство. Вы, конечно, можете поместить неформатированное устройство на логический том, но при этом обессмысливается сама идея – такой накопитель уже не является неформатированным. В общем, из-за того крохотного выигрыша, который дают неформатированные устройства, не стоит затевать суету.

Старые версии строк и табличное пространство

В условиях интенсивной записи табличное пространство InnoDB может стать очень большим. Если транзакция долгое время остается открытой (даже не делая ничего полезного) и при этом установлен уровень изоляции `REPEATABLE READ`, то InnoDB не может удалить старые версии строк, поскольку они должны быть видны незафиксированным транзакциям. InnoDB хранит старые версии в табличном пространстве, поэтому оно продолжает расти по мере обновления данных. Иногда проблема связана не с открытыми транзакциями, а просто с характером рабочей нагрузки: чисткой занимается только один поток, он может просто не успевать удалять старые версии строк.

В любом случае команда `SHOW INNODB STATUS` поможет идентифицировать причину проблемы. Взгляните на первую и вторую строки в секции `TRANSACTIONS`, там показан текущий номер транзакции и точка, до которой дошла чистка. Если разница велика, значит, скопилось много невычищенных транзакций. Например:

```
------------
TRANSACTIONS
------------
Trx id counter 0 80157601
Purge done for trx's n:o <0 80154573 undo n:o <0 0
```

Идентификатор транзакции – это 64-разрядное число, составленное из двух 32-разрядных, поэтому для вычисления разности придется немного помучиться. В данном случае все просто, так как старшие разряды равны 0, следовательно, мы имеем 80157601 - 80154573 = 3028 потенциально невычищенных транзакций (утилита *innotop* проделает за вас все вычисления). Мы говорим «потенциально», поскольку большая разность еще не означает, что имеется много невычищенных строк. Старые версии строк создаются только для транзакций, которые изменяют данные, а может существовать множество транзакций, в которых дан-

ные не изменялись (и наоборот – одна транзакция может изменить много строк).

Если имеется много невычищенных транзакций и по этой причине табличное пространство продолжает расти, то можно принудительно замедлить работу MySQL, чтобы поток очистки InnoDB успевал справляться с нагрузкой. Звучит не слишком привлекательно, но альтернативы нет. В противном случае InnoDB будет продолжать писать данные и заполнять диск, пока на нем не кончится место или не будет достигнута заданная верхняя граница размера табличного пространства.

Чтобы «притормозить» запись, присвойте переменной `innodb_max_purge_lag` значение, отличное от 0. Оно равно максимальному количеству транзакций, ожидающих очистки; по достижении этого порога InnoDB начинает задерживать выполнение запросов на обновление данных. Чтобы выбрать подходящее значение, нужно знать конкретную рабочую нагрузку. Например, если типичная транзакция изменяет в среднем 1 Кбайт данных и вы готовы смириться со 100 Мбайт «невычищенных» строк в табличном пространстве, то можете задать значение 100000.

Имейте в виду, что наличие невычищенных версий строк отражается на всех запросах, поскольку из-за них увеличивается размер таблиц и индексов. Если поток очистки не справляется с нагрузкой, то производительность может заметно упасть. Задание параметра `innodb_max_purge_lag` тоже снижает скорость выполнения запросов, но это меньшее из двух зол.

Буфер двойной записи

В InnoDB *буфер двойной записи* применяется для того, чтобы избежать повреждения данных в случае частичной записи страницы. Частичная запись страницы возникает, когда операция записи на диск выполняется не полностью, так что записанной оказывается только часть страницы размером 16 Кбайт. Причин для этого много (сбои, ошибки и т. д.). Буфер двойной записи в данном случае предотвращает повреждение данных.

Буфер двойной записи представляет собой зарезервированную область в табличном пространстве, достаточно большую для хранения 100 страниц в одном непрерывном блоке. По существу, это резервная копия недавно записанных страниц. Когда InnoDB сбрасывает страницы из пула буферов на диск, она сначала записывает (и сбрасывает) их в буфер двойной записи, а уже потом в ту основную область данных, где им и место. Тем самым гарантируется, что каждая операция записи страницы атомарна и долговечна.

Означает ли это, что каждая страница записывается дважды? Да, означает, но поскольку InnoDB пишет в буфер двойной записи сразу несколько страниц и лишь потом вызывает `fsync()` для синхронизации с диском, то на производительности это почти не отражается – обыч-

но процесс отнимает не более нескольких процентов. Важнее тот факт, что эта стратегия позволяет гораздо эффективнее использовать файлы журнала. Поскольку буфер двойной записи дает InnoDB надежную гарантию того, что данные не повреждены, то в журнал помещаются не страницы целиком, а скорее двоичные дельты.

Если частичная запись произойдет при сохранении страницы в сам буфер двойной записи, то оригинальная страница все равно окажется на диске в том месте, где ей положено быть. На этапе восстановления InnoDB возьмет оригинальную страницу вместо ее поврежденной копии в буфере двойной записи. Если же запись в буфер завершилась успешно, а в саму таблицу – с ошибкой, то во время восстановления InnoDB воспользуется копией из буфера двойной записи. InnoDB понимает, что данные повреждены, так как в конце каждой страницы имеется контрольная сумма (она записывается последней) – если эта сумма не соответствует содержимому страницы, значит, имеет место проблема.

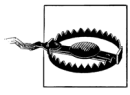
Поэтому в ходе восстановления InnoDB читает каждую страницу, находящуюся в буфере двойной записи, и проверяет контрольную сумму. Если она неверна, то считывается оригинальная страница.

В некоторых случаях без буфера двойной записи можно обойтись – например, его можно отключить на подчиненных серверах. Кроме того, некоторые файловые системы (в частности, ZFS) делают то же самое самостоятельно, поэтому излишне повторять процедуру на уровне InnoDB. Чтобы отключить буфер двойной записи, присвойте параметру `innodb_doublewrite` значение `0`.

Прочие параметры настройки ввода/вывода

Параметр `sync_binlog` управляет тем, как MySQL сбрасывает двоичный журнал на диск. По умолчанию он равен 0, то есть MySQL вообще не выполняет сброс, оставляя его на усмотрение операционной системы. Значение, большее 0, интерпретируется как количество операций записи в двоичный журнал между двумя последовательными сбросами (операцией считается одна команда, если режим `autocommit` включен, и одна транзакция в противном случае). Обычно этот параметр устанавливают в `0` или `1`.

Если `sync_binlog` отличен от `1`, то в случае сбоя двоичный журнал может оказаться не синхронизированным с транзакционными данными. Это вполне способно привести к сбою в репликации и сделать невозможным восстановление на определенный момент в прошлом. Однако уровень безопасности, достигаемый установкой этого параметра в `1`, обходится недешево. Для синхронизации двоичного журнала с журналом транзакций MySQL должна производить сброс двух файлов в двух разных местах. Для этого может потребоваться относительно медленная операция подвода головки к дорожке.



При одновременном использовании двоичного журнала и InnoDB в версии MySQL 5.0 и более поздних, а особенно после перехода с более ранней версии, следует очень осторожно подходить к недавно появившейся поддержке XA-транзакций. Она спроектирована с целью синхронизировать фиксацию транзакций между разными подсистемами хранения и двоичным журналом, но при этом отключает механизм групповой фиксации в InnoDB. Это может привести к резкому падению производительности из-за существенного возрастания количества вызовов `fsync()`, необходимых для фиксации транзакций. Чтобы разрешить эту проблему, рекомендуется отключить двоичный журнал *и* поддержку XA в InnoDB, задав параметр `innodb_support_xa=0`. При наличии в RAID-контроллере кэша с резервным питанием вызовы `fsync()` выполняются быстро, поэтому указанная проблема может и не возникнуть.

Как и в случае файла журнала InnoDB, помещение двоичного журнала на том RAID-массива, оборудованного кэшем записи с резервным питанием, нередко дает огромный прирост производительности.

Замечание о двоичном журнале, не относящееся к производительности: если вы собираетесь воспользоваться параметром `expire_logs_days` для автоматического удаления старых двоичных журналов, то не уничтожайте их вручную командой *rm*. В этом случае сервер запутается и откажется производить автоматическое удаление, а команда `PURGE MASTER LOGS` перестанет работать. Если вы окажетесь в такой ситуации, то нужно будет вручную синхронизировать файл *hostname-bin.index* со списком файлов на диске.

RAID-массивы мы будем обсуждать более подробно в главе 7, но сейчас стоит отметить, что высококачественные RAID-контроллеры, оборудованные кэшем с резервным батарейным питанием и работающие в режиме отложенной записи, могут справляться с *тысячами* операций в секунду и при этом обеспечить надежное хранение. Поскольку кэш запитан от батареи, то его содержимое не теряется даже при пропадании питания. После восстановления питания RAID-контроллер переписывает данные из кэша на диск еще до того, как сделать диск доступным для работы. Таким образом, хороший RAID-контроллер, оборудованный кэшем записи с резервным питанием, может резко повысить производительность и потому является отличным капиталовложением.

Настройка конкурентного доступа в MySQL

Если MySQL работает в режиме высокой конкуренции, то могут возникнуть узкие места, не встречающиеся при других условиях. В следующих разделах мы расскажем, как распознать подобные проблемы и как добиться наилучшей производительности при такой рабочей нагрузке в подсистемах хранения MyISAM и InnoDB.

Настройка конкурентного доступа для MyISAM

В условиях одновременного чтения и записи нужно тщательно следить за тем, чтобы пользователи не увидели несогласованные результаты. При некоторых условиях MyISAM допускает конкурентную вставку и чтение и позволяет «планировать» некоторые операции, чтобы по возможности уменьшить количество блокировок.

Но прежде чем мы приступим к описанию параметров настройки конкурентного доступа, важно понимать, как MyISAM удаляет и вставляет данные. Операция удаления не реорганизует всю таблицу, а лишь помечает строки соответствующим признаком, оставляя «дыры» в таблице. MyISAM старается по возможности заполнить эти дыры, используя освободившееся место для вставки новых строк. Если дыр нет, то следующие данные дописываются в конец таблицы.

Несмотря на то, что в подсистеме MyISAM есть табличные блокировки, она может дописывать новые строки одновременно с чтением. Для этого сервер следит, чтобы операции чтения останавливались на последней строке, которая существовала в момент их начала. Таким образом, удается избежать несогласованных операций.

Однако обеспечить согласованность чтения, когда что-то изменяется в середине таблицы, гораздо труднее. Популярный способ решения этой проблемы дает технология MVCC: она открывает читателям доступ к старым версиям строк, пока писатели создают новые версии. MyISAM не поддерживает MVCC, поэтому не поддерживает и конкурентные вставки, если только вставка не производится в конец таблицы.

Поведение конкурентной вставки в MyISAM можно сконфигурировать с помощью переменной `concurrent_insert`, которая может принимать следующие значения:

`0` MyISAM вообще не допускает конкурентных вставок; любая вставка монопольно блокирует таблицу.

`1` Это значение принимается по умолчанию. MyISAM допускает конкурентную вставку, если в таблице нет дыр.

`2` Это значение появилось в версии MySQL 5.0. В данном случае конкурентная вставка принудительно производится в конец таблицы, даже если в ней есть дыры. Если же ни один поток не читает из таблицы, то MySQL помещает новые строки в дыры. С использованием такого режима может возрасти фрагментация данных, поэтому в зависимости от характера рабочей нагрузки, возможно, придется чаще оптимизировать таблицы.

Можно также сконфигурировать MySQL так, чтобы некоторые операции откладывались на более позднее время, когда их можно будет сгруппировать для повышения эффективности. Например, переменная `delay_key_write` позволяет задать режим отложенной записи в индекс (мы уже говорили об этом выше в текущей главе). Тут возникает зна-

комый компромисс: писать в индекс немедленно (безопасно, но дорого) или подождать в надежде, что до момента записи не произойдет сбой электропитания (быстрее, но любая неисправность может привести к обширному повреждению индекса из-за его неактуальности). Параметр `low_priority_updates` позволяет назначать командам `INSERT`, `REPLACE`, `DELETE` и `UPDATE` более низкий приоритет, чем команде `SELECT`. Этот режим эквивалентен применению подсказки `LOW_PRIORITY` ко всем запросам обновления. Дополнительную информацию по этому поводу см. в разделе «Подсказки оптимизатору запросов» на стр. 250.

Наконец, хотя проблемы масштабируемости чаще обсуждаются в контексте InnoDB, у подсистемы MyISAM тоже в течение длительного времени возникали сложности с мьютексами. В версии MySQL 4.0 и более ранних любой доступ к буферу ключей был защищен глобальным мьютексом, что затрудняло масштабируемость в системах с несколькими процессорами и дисками. В версии MySQL 4.1 код работы с буфером ключей был усовершенствован, и теперь этой проблемы нет, но, тем не менее, каждый буфер ключей по-прежнему защищен мьютексом. Это вызывает сложности, когда поток копирует блоки ключей из буфера в собственную локальную память, а не читает их с диска. Диск перестает быть узким местом, но теперь таковым является доступ к данным в буфере ключей. Иногда эту проблему удается решить за счет организации нескольких буферов ключей, но подобный подход не всегда помогает. Например, ничего нельзя сделать, если все упирается в единственный индекс. В результате конкурентные запросы `SELECT` могут выполняться на машинах с несколькими процессорами гораздо медленнее, чем на машине с одним процессором, даже в тех случаях, когда кроме них вообще ничего не выполняется.

Настройка конкурентного доступа для InnoDB

Подсистема InnoDB изначально проектировалась для работы в условиях высокой конкуренции, но и она не идеальна. До сих пор видно, что своими корнями архитектура InnoDB уходит во времена систем с ограниченной памятью, одним процессором и одним диском. Некоторые характеристики InnoDB, относящиеся к производительности, резко ухудшаются в условиях высокой конкуренции, и тогда остается единственный выход – ограничить конкуренцию. Чтобы понять, испытывает ли InnoDB проблемы из-за конкуренции, обратите внимание на секцию `SEMAPHORES` в результатах выполнения команды `SHOW INNODB STATUS`. Подробнее см. раздел «Секция `SEMAPHORES`» на стр. 692.

В InnoDB имеется собственный «планировщик потоков», который управляет тем, как потоки входят в ядро для доступа к данным и что они могут делать, находясь в ядре. Самый простой способ ограничить степень конкуренции – воспользоваться переменной `innodb_thread_concurrency`, которая определяет, сколько потоков могут находиться в ядре одновре-

менно. Значение 0 символизирует отсутствие ограничения на число потоков. При возникновении проблем с конкурентным доступом в InnoDB на эту переменную следует обратить внимание в первую очередь.

Невозможно заранее сказать, какое значение лучше всего подходит для заданной архитектуры и рабочей нагрузки. Теоретически можно воспользоваться следующей формулой:

```
concurrency = Количество ЦП * Количество дисков * 2
```

Но на практике может оказаться, что лучше задавать гораздо меньшее число. Чтобы подобрать походящее значение этого параметра для своей системы, придется заняться экспериментированием и тестированием.

Если в ядре уже находится больше потоков, чем разрешено, то никакой другой поток не сможет в него войти. В InnoDB применяется двухшаговая процедура, смысл которой заключается в том, чтобы сделать вход в ядро максимально эффективным. Эта политика сокращает издержки на контекстные переключения, свойственные планировщику операционной системы. Поток сначала засыпает на `innodb_thread_sleep_delay` микросекунд, а потом пытается войти снова. Если это по-прежнему не получается, то поток становится в очередь ожидания и уступает управление операционной системе.

По умолчанию на первом шаге поток спит в течение 10 000 микросекунд. При высокой конкуренции, когда процессор не используется полностью, потому что множество потоков находится в состоянии «спит перед постановкой в очередь», можно попробовать изменить этот параметр. Принимаемое по умолчанию значение может оказаться слишком велико, если имеется множество мелких запросов, поскольку ко времени обработки каждого запроса добавляется 10 миллисекунд.

Если поток уже вошел в ядро, то он получает определенное количество «билетов», позволяющих ему вернуться в ядро «бесплатно», то есть без проверки условий конкуренции. Тем самым налагаются ограничения на объем работы, который поток может выполнить перед тем, как встать в очередь наравне с остальными ожидающими потоками. Переменная `innodb_concurrency_tickets` определяет количество билетов. Ее редко приходится изменять, разве что в случае, когда имеется большое количество долго выполняющихся запросов. Билеты выдаются на запрос, а не на транзакцию. После того как запрос выполнен, все оставшиеся билеты аннулируются.

Помимо узких мест, возникающих из-за пула буферов и других структур, существует узкое место и на этапе фиксации транзакции. Оно связано главным образом с вводом/выводом и вызвано операциями сброса. Переменная `innodb_commit_concurrency` определяет, сколько потоков могут одновременно выполнять фиксацию. Ее имеет смысл настраивать, если из-за слишком низкого значения переменной `innodb_thread_concurrency` потоки начинают пробуксовывать.

Создатели InnoDB работают над решением этих проблем, и в версиях MySQL 5.0.30 b 5.0.32 уже заметны значительные улучшения.

Настройка с учетом рабочей нагрузки

Конечная цель настройки сервера – адаптировать его к конкретной рабочей нагрузке. Для этого нужно хорошо знать все виды выполняющихся задач: их количество, типы и частоту. Причем речь идет не только о запросах, но и о таких операциях, как подключение к серверу и сброс таблиц. Вы должны также знать, как вести мониторинг и интерпретировать информацию о состоянии и активности MySQL и операционной системы; об этом см. главы 7 и 14.

Первое, что нужно сделать, если вы этого еще не сделали, – познакомиться со своим сервером. Узнать, какие запросы он выполняет. Последить за ним с помощью утилиты *innotop* или другого инструмента. Полезно знать не только, что он делает в общем, но и на что тратит основное время каждый запрос. Один из способов добыть эти сведения заключается в том, чтобы написать сценарий, агрегирующий выдачу команды `SHOW PROCESSLIST` по столбцу `Command` (в *innotop* эта возможность уже встроена), или просто изучить ее визуально. Обращайте внимание на потоки, которые много времени проводят в определенном состоянии.

Если в какие-то периоды времени сервер работает на полную мощность, попробуйте получить список процессов, поскольку это лучший способ понять, какие виды запросов больше всего страдают. Например, многие ли из них копируют результаты во временную таблицу или занимаются сортировкой результатов? Если да, значит, нужно обратить внимание на конфигурационные параметры, относящиеся к временным таблицам и буферам сортировки (возможно, понадобится оптимизировать и сами запросы).

Обычно мы рекомендуем использовать заплаты, которые разработаны для журналов MySQL. Они дают массу ценной информации о том, что делает каждый запрос, и позволяют анализировать нагрузку гораздо более подробно. Эти заплаты включены в последние официальные дистрибутивы MySQL, так что не исключено, что в вашем сервере они уже есть. Детали см. в разделе «Тонкая настройка протоколирования» на стр. 99.

Оптимизация работы с полями типа BLOB и TEXT

Столбцы типа `BLOB` и `TEXT` представляют для MySQL особый вид рабочей нагрузки. Мы будем для простоты называть те и другие просто `BLOB`, потому что они принадлежат к одному и тому же классу типов данных. `BLOB` имеют ряд ограничений, из-за которых сервер работает с ними несколько иначе, чем с другими типами. В частности, сервер не может использовать для `BLOB` временные таблицы в памяти. Следовательно, если запрос включает `BLOB`, то независимо от всех остальных факторов вре-

менная таблица создается на диске. Это крайне неэффективно, особенно если в остальных отношениях запрос маленький и мог бы быть выполнен очень быстро. На работу с временной таблицей может уходить львиная доля всего времени обработки запроса.

Уменьшить эти издержки можно двумя способами: преобразовать значение к типу `VARCHAR` с помощью функции `SUBSTRING()` (см. раздел «Строковые типы» на стр. 120) или ускорить работу с временными таблицами.

Последнее проще всего сделать, поместив временные таблицы в файловую систему, находящуюся целиком в памяти (`tmpfs` в случае GNU/Linux). Это устраняет часть накладных расходов, хотя все равно гораздо медленнее, чем временные таблицы. Использование файловой системы в памяти помогает, поскольку ОС стремится избежать записи на диск[1]. Обычные файловые системы тоже поддерживают кэш в памяти, но ОС должна каждые несколько секунд сбрасывать кэш на диск. К тому же при проектировании файловой системы `tmpfs` специально были поставлены две цели: низкие накладные расходы и простота. В частности, для этой файловой системы не выполняются действия, направленные на возможность ее восстановления. Это ускоряет работу.

Где именно создаются временные таблицы, определяет параметр `tmpdir`. Следите за заполнением файловой системы, чтобы в ней всегда было достаточно места для временных таблиц. При необходимости можно задать даже несколько мест для их размещения, MySQL будет использовать временные таблицы по кругу.

Если `BLOB` очень велики, а вы работаете с InnoDB, то, возможно, придется увеличить размер буфера журнала. Мы уже подробно обсуждали этот вопрос в настоящей главе. Для длинных столбцов переменной длины (например, типа `BLOB`, `TEXT` и `VARCHAR` большого размера) InnoDB хранит префикс длиной 768 байтов в самой странице вместе со строкой[2]. Если значение в столбце длиннее префикса, то для хранения остатка InnoDB может выделить внешнюю память вне строки. Эта память сегментируется страницами длиной 16 Кбайт (как и все остальные страницы в InnoDB), причем каждому столбцу отводится отдельная страница (столбцы не используют совместно внешнюю память). InnoDB выделяет столбцу внешнюю память по одной странице за раз до тех пор, пока не будет выделено 32 страницы; после этого за один раз выделяется сразу 64 страницы.

Отметим, что мы сказали: «*может* выделить внешнюю память». Если полная длина строки, включая все значение длинного столбца, меньше максимально допустимой в InnoDB длины строки (чуть меньше 8 Кбайт),

---

[1] Данные все равно могут писаться на диск, если ОС вынуждена прибегать к подкачке.

[2] Этого вполне достаточно для создания по столбцу индекса с ключом длиной 255 символов даже в кодировке utf-8, где один символ может кодироваться тремя байтами.

то InnoDB не станет выделять внешнюю память, даже если длина значения в длинном столбце превышает длину префикса.

Наконец, отметим, что, когда InnoDB обновляет длинный столбец, для которого выделена внешняя память, то существующее значение не изменяется. Вместо этого очередное значение записывается на новое место во внешней памяти, а старое значение удаляется.

Последствия описанного механизма таковы.

- На хранение длинных столбцов в InnoDB может впустую расходоваться много места. Например, если для размещения значения в строке не хватает всего лишь одного байта, то для этого единственного не поместившегося байта будет выделена целая страница, и большая ее часть окажется потраченной впустую. Аналогично, если значение лишь чуть-чуть длиннее 32 страниц, то для его хранения будет отведено 96 страниц на диске.
- Наличие внешней памяти отключает адаптивный хеш-индекс, поскольку его использование подразумевает сравнение полных длин столбцов для гарантии того, что найдены нужные данные. Хеш позволяет InnoDB очень быстро находить «примерно то, что нужно», но затем надо проверить, что «догадка» верна. Так как адаптивный хеш-индекс целиком находится в памяти и строится «поверх» страниц в пуле буферов, к которым часто производятся обращения, то с внешней памятью он работать не может.
- Из-за длинных значений могут медленно выполняться запросы, содержащие условие `WHERE`, для которого нет подходящего индекса. Перед применением `WHERE` MySQL читает все запрошенные столбцы, поэтому может потребоваться, чтобы InnoDB прочитала все внешние страницы. Затем проверяется выполнение условия `WHERE` и все прочитанные данные отбрасываются. Выбирать ненужные столбцы вообще никогда не следует, но в этом частном случае особенно важно не допускать такой ошибки. Если обнаруживается, что это ограничение относится к вашим запросам, можно попробовать использовать покрывающие индексы. Дополнительную информацию см. в разделе «Покрывающие индексы» на стр. 163.
- Если в одной таблице есть много длинных столбцов, то, возможно, будет лучше объединить их в один столбец, например в виде XML-документа. Тогда для объединенного значения будет выделена только одна область во внешней памяти, а не по отдельному набору страниц на каждый столбец.
- Иногда можно получить значительный выигрыш в пространстве и производительности, если поместить длинные столбцы в `BLOB` и сжать функцией `COMPRESS()` или сжимать данные на уровне приложения перед отправкой MySQL.

Оптимизация файловой сортировки (filesort)

Есть две переменные, помогающие управлять выполнением файловых сортировок.

Напомним, что в разделе «Оптимизация сортировки» (стр. 229) мы отмечали, что MySQL использует два алгоритма файловой сортировки. Двухпроходный алгоритм применяется, если суммарная длина всех столбцов, отбираемых запросом, плюс длина столбцов, упоминаемых во фразе `ORDER BY`, превышает `max_length_for_sort_data` байтов. Этот алгоритм задействован и тогда, когда хотя бы один из запрошенных столбцов – пусть даже он не встречается в `ORDER BY` – имеет тип `BLOB` или `TEXT`. Для преобразования таких столбцов в тип, к которому применим однопроходный алгоритм, можно воспользоваться функцией `SUBSTRING()`.

На выбор алгоритма можно повлиять, задав параметр `max_length_for_sort_data`. Поскольку в однопроходном алгоритме для каждой строки создается буфер фиксированного размера, то при подборе значения `max_length_for_sort_data` учитывают максимальную, а не фактическую длину столбцов типа `VARCHAR`. Поэтому мы рекомендуем не задавать для таких столбцов бóльшую длину, чем необходимо.

При сортировке по столбцам типа `BLOB` или `TEXT` MySQL принимает во внимание только префикс, а остаток значения игнорирует. Связано это с тем, что для значений нужно выделить структуру фиксированной длины, а затем скопировать в нее префикс из внешней памяти. Длина такого префикса задается параметром `max_sort_length`.

К сожалению, MySQL не сообщает никакой информации о выбранном алгоритме сортировки. Если после увеличения переменной `max_length_for_sort_data` интенсивность использования диска возрастает, потребление ЦП падает, а переменная состояния `Sort_merge_passes` начинает расти быстрее, чем раньше, то, скорее всего, количество сортировок однопроходным алгоритмом стало больше.

Дополнительную информацию о типах `BLOB` и `TEXT` см. в разделе «Строковые типы» на стр. 120.

Переменные состояния сервера MySQL

Один из самых продуктивных способов настройки MySQL под имеющуюся рабочую нагрузку – анализ информации, которую выводит команда `SHOW GLOBAL STATUS`, с тем, чтобы понять, какие параметры следует изменить. Если вы только приступаете к настройке сервера и знакомы с утилитой *mysqlreport*, то, запустив и изучив сгенерированный, очень удобный для восприятия отчет, сможете сэкономить массу времени. Этот отчет поможет найти те места, где возможны проблемы, а потом уже можно будет более тщательно проанализировать соответствующие переменные состояния с помощью `SHOW GLOBAL STATUS`. Обнаружив что-то поддающееся улучшению, вы сможете заняться настройкой. Затем запустите команду *mysqladmin extended -r -i60*, которая будет периодиче-

ски выводить информацию о текущем состоянии, и посмотрите на эффект ваших изменений. Лучше всего видеть и анализировать как абсолютные значения переменных состояния, так и их дельты.

В главе 13 предложено детальное описание всех переменных состояния, выводимых командой `SHOW GLOBAL STATUS`. Ниже мы приводим список тех переменных, на которые нужно смотреть в первую очередь:

`Aborted_clients`

Если эта переменная со временем растет, проверьте, корректно ли закрываются соединения. Если нет, обратите внимание на производительность сети, а также на конфигурационную переменную `max_allowed_packet`. Запросы, в которых значение этой переменной превышено, завершаются аварийно.

`Aborted_connects`

Значение этой переменной должно быть близко к нулю. Если это не так, то, возможно, имеют место проблемы с сетью. Несколько неудавшихся подключений – это не страшно. Например, так бывает, когда кто-то пытается соединиться с неизвестного сервера, вводит неверное имя пользователя или пароль либо указывает несуществующую базу данных.

`Binlog_cache_disk_use` *и* `Binlog_cache_use`

Если отношение `Binlog_cache_disk_use` к `Binlog_cache_use` велико, попробуйте увеличить значение `binlog_cache_size`. Желательно, чтобы большинство транзакций целиком умещались в кэш двоичного журнала, но ничего страшного, если временами какая-то транзакция «просочится» на диск.

Нельзя дать точных рекомендаций о том, как уменьшить количество непопаданий в кэш двоичного журнала. Самый лучший подход – увеличить параметр `binlog_cache_size` и посмотреть, уменьшится ли число непопаданий в кэш. После достижения определенного порога дальнейшее увеличение размера кэша может не привести к улучшению. Предположим, что было одно непопадание в секунду, а после увеличения размера удалось довести этот показатель до одного непопадания в минуту. Это достаточно хорошо, и вряд ли он еще уменьшится, но, даже если и получится это сделать, то выигрыш будет мизерным, поэтому лучше отвести память для чего-нибудь другого.

`Bytes_received` *и* `Bytes_sent`

Эти значения помогают понять, не слишком ли велик трафик в направлении к серверу или от него[1]. Возможно, причина таится где-то в вашем коде, например в запросе, который отбирает больше данных,

[1] Даже если пропускная способность сети достаточна, не отметайте с порога эту причину снижения производительности. Проблема может быть связана с большими сетевыми задержками.

чем необходимо. (См. раздел «Клиент-серверный протокол MySQL» на стр. 211.)

`Com_*`

Проверьте, вдруг значения таких переменных, как `Com_rollback` выше ожидаемых. Чтобы быстро узнать, какие значения разумны, воспользуйтесь режимом Command Summary утилиты *innotop* (подробнее об *innotop* см. главу 14).

`Connections`

Эта переменная отражает количество попыток соединения (а не число текущих соединений, которое хранится в переменной `Threads_connected`). Если значение быстро увеличивается – порядка сотен в секунду, – значит, имеет смысл настроить пул соединений или сетевой стек ОС (о конфигурировании сети см. следующую главу).

`Created_tmp_disk_tables`

Если это значение велико, то возможно одно из двух: либо запросы создают временные таблицы в результате выборки столбцов типа `BLOB` или `TEXT`, либо недостаточно велики значения конфигурационных параметров `tmp_table_size` и/или `max_heap_table_size`.

`Created_tmp_tables`

Единственный способ справиться со слишком большим значением этой переменной – оптимизировать сами запросы. Советы по оптимизации см. в главах 3 и 4.

`Handler_read_rnd_next`

Отношение `Handler_read_rnd_next / Handler_read_rnd` дает приблизительную оценку среднего размера полного сканирования таблиц. Если оно велико, то, возможно, следует оптимизировать схему, индексы или запросы.

`Key_blocks_used`

Если величина `Key_blocks_used * key_cache_block_size` гораздо меньше, чем параметр `key_buffer_size` на прогретом сервере, то размер буфера ключей (`key_buffer_size`) больше необходимого, и вы только впустую растрачиваете память.

`Key_reads`

Понаблюдайте за количеством операций чтения в секунду и посмотрите, насколько близко это значение приближается к предельным показателям подсистемы ввода/вывода. Дополнительную информацию см. в главе 7.

`Max_used_connections`

Если это значение совпадает с `max_connections`, то либо `max_connections` слишком мало, либо количество запросов на подключение в пиковые периоды превышает сконфигурированные лимиты сервера. Но не следует сей же момент увеличивать `max_connections`! Этот параметр

не позволяет серверу захлебнуться в условиях слишком большой нагрузки. Если вы наблюдаете всплеск, то для начала проверьте, нет ли ошибки в приложении, правильно ли настроен сервер и хорошо ли спроектирована схема. Лучше исправить приложение, чем просто увеличивать значение `max_connections`.

`Open_files`

Следите за тем, чтобы это значение не приближалось к величине `open_files_limit`. Если такое происходит, то, вероятно, стоит увеличить этот лимит.

`Open_tables` *и* `Opened_tables`

Сравните это значение с величиной параметра `table_cache`. Если количество открываемых таблиц (`Opened_tables`) в секунду велико, то, вероятно, размер кэша таблиц (`table_cache`) недостаточен. Однако явное создание временных таблиц также может привести к увеличению количества открытых таблиц, хотя кэш при этом используется не полностью, поэтому не исключено, что основания для беспокойства нет.

`Qcache_*`

Дополнительную информацию о кэше запросов см. в разделе «Кэш запросов MySQL» на стр. 261.

`Select_full_join`

Полное соединение – это соединение без индексов, такая операция может очень сильно «посадить» производительность. Лучше, чтобы их вовсе не было, даже одного в минуту может быть много. Обнаружив соединение без индексов, примите все меры к оптимизации запросов.

`Select_full_range_join`

Если это число велико, значит, имеется слишком много запросов, в которых для соединения таблиц применяется стратегия поиска по диапазону. Это медленно, здесь есть простор для оптимизации.

`Select_range_check`

Эта переменная отслеживает планы запросов, в которых повторно анализируется выбор ключей для каждой строки в соединении. Накладные расходы такого плана велики. Если значение достаточно большое или растет, значит, для некоторых запросов не удается подобрать хорошего индекса.

`Slow_launch_threads`

Если данная переменная состояния велика, значит, что-то «тормозит» создание новых потоков в ответ на запрос о соединении. Это служит указанием на наличие какой-то проблемы с сервером, но ничего не говорит о ее природе. Обычно все объясняется перегрузкой

операционной системы, из-за которой она не может выделить процессорное время вновь создаваемым потокам.

`Sort_merge_passes`

Большое значение этой переменной означает, что надо бы увеличить размер буфера сортировки (`sort_buffer_size`), быть может, только ради некоторых запросов. Проверьте запросы и найдите среди них те, которые приводят к сортировке (filesort). Возможно, их удастся оптимизировать.

`Table_locks_waited`

Эта переменная говорит о том, сколько раз пришлось ждать освобождения табличной блокировки на уровне сервера (ожидание блокировок, реализованных в подсистеме хранения, например блокировок строк в InnoDB, не приводит к увеличению этой переменной). Если значение велико и растет, значит, имеется серьезная проблема с конкурентным доступом. Возможно, имеет смысл подумать об использовании InnoDB или какой-то другой подсистемы хранения, в которой реализована блокировка на уровне строк, о секционировании больших таблиц вручную или о встроенном в MySQL 5.1 механизме секционирования, а затем об оптимизации запросов, об использовании конкурентных вставок или о настройке параметров блокирования.

MySQL ничего не говорит о том, как долго пришлось ждать. На момент написания этой книги получить ответ на этот вопрос было проще всего, изучив журнал медленных запросов с микросекундной точностью. Подробнее об этом см. в разделе «Профилирование MySQL» главы 2 на стр. 96.

`Threads_created`

Если это значение велико или растет, то, возможно, стоит увеличить параметр `thread_cache_size`. Переменная `Threads_cached` показывает, сколько потоков уже находится в кэше.

Настройка параметров уровня соединения

Не следует глобально увеличивать параметр, относящийся к соединению, если вы не уверены в правильности такого решения. Некоторые буферы выделяются одним куском, даже если они не нужны, поэтому глобальное изменение параметра может привести к растрачиванию памяти впустую. Лучше увеличивать значение только, когда это необходимо для конкретного запроса.

Типичный пример переменной, которую лучше оставить небольшой, а увеличивать только для некоторых запросов, – это `sort_buffer_size` (управляет размером буфера для сортировки (filesort)). Этот буфер выделяется целиком даже для очень маленьких сортировок, поэтому увеличение его размера сверх необходимого для средней сортировки толь-

ко приводит к пустой трате памяти и росту накладных расходов на ее выделение.

Обнаружив запрос, который мог бы выполняться быстрее при наличии большого буфера сортировки, вы можете увеличить значение `sort_buffer_size` непосредственно перед его выполнением для конкретной сессии, а затем вернуться к значению `DEFAULT`. Вот как это делается:

```
SET @@session.sort_buffer_size := <value>;
-- Выполнить запрос...
SET @@session.sort_buffer_size := DEFAULT;
```

Для такого рода кода могут пригодиться функции-обертки. К числу других переменных, которые разумно устанавливать на уровне соединения, относятся `read_buffer_size`, `read_rnd_buffer_size`, `tmp_table_size` и `myisam_sort_buffer_size` (для исправления таблиц).

Чтобы сохранить и восстановить значения переменной можно написать примерно такой код:

```
SET @saved_<unique_variable_name> := @@session.sort_buffer_size;
SET @@session.sort_buffer_size := <value>;
-- Выполнить запрос...
SET @@session.sort_buffer_size := @saved_<unique_variable_name>;
```

7

Оптимизация операционной системы и оборудования

Сервер MySQL в целом работает не лучше, чем самое слабое звено всего аппаратно-программного комплекса, и зачастую лимитирующими факторами являются операционная система и оборудование. Емкость диска, объем доступной памяти, число процессоров, сеть и компоненты, которые связывают все воедино, – все это может ограничивать общую производительность системы.

В предыдущих главах мы говорили об оптимизации самого сервера MySQL и приложений. Такая настройка очень важна, но следует также принимать во внимание оборудование и правильно конфигурировать операционную систему. Например, если при текущей рабочей нагрузке узким местом является ввод/вывод, то можно, конечно, попробовать перепроектировать приложение, так чтобы уменьшить объем подобных операций на уровне MySQL. Но чаще разумнее модернизировать подсистему ввода/вывода, установить дополнительную память или переконфигурировать существующие диски.

Аппаратура изменяется очень быстро, поэтому в этой главе мы не станем сравнивать разные продукты или упоминать какие-то конкретные компоненты. Наша цель – дать рекомендации и наметить подходы к расшивке узких мест, обусловленных оборудованием и операционной системой.

Мы начнем с рассмотрения факторов, ограничивающих производительность MySQL. Самые типичные проблемы – это процессор, память и ввод/вывод, но проявление их может быть далеко не очевидным. Мы рассмотрим вопрос о выборе ЦП для серверов MySQL, а затем перейдем к задаче подбора баланса между оперативной памятью и дисками. Мы исследуем различные типы ввода/вывода (произвольный и последовательный, чтение и запись) и объясним, как определить характеристики своего рабочего множества. Зная это, вы сможете определить, при ка-

ком соотношении оперативной памяти к дисковой достигается максимальная эффективность. Затем мы перейдем к вопросу о выборе дисков для серверов MySQL и к важнейшей теме, посвященной оптимизации RAID-массива. Обсуждение внешней памяти мы завершим обзором некоторых имеющихся технологий (например, сетей хранения данных – SAN) и рекомендациями о том, как и когда использовать разные дисковые тома для данных и журналов MySQL.

От внешней памяти мы перейдем к вопросам производительности сети и выбора операционной и файловой систем. Затем мы посмотрим, в какой поддержке потоков нуждается MySQL, и расскажем, как избежать подкачки страниц. И в завершение этой главы мы приведем примеры распечаток состояния операционной системы.

Что ограничивает производительность MySQL?

На производительность MySQL могут оказывать влияние многие аппаратные компоненты, но чаще всего узким местом оказывается перегрузка процессора и подсистемы ввода/вывода. Перегрузка процессора возникает, когда MySQL работает с данными, которые целиком помещаются в оперативной памяти или могут считываться с диска с необходимой скоростью. В качестве примеров можно привести криптографические операции или соединение с помощью индексов, сводящееся к вычислению декартова произведения.

Что касается перегрузки подсистемы ввода/вывода, то оно обычно имеет место, когда нужно прочитать больше данных, чем помещается в память. Если приложение является распределенным или выполняет очень много запросов и при этом требуется низкая задержка, то это узкое место может переместиться в сеть.

Когда вам кажется, что вы нашли узкое место, не позволяйте себе останавливаться на очевидном с первого взгляда. Слабое звено в одном сегменте часто воздействует на какую-то другую подсистему, которая и кажется источником проблем. Например, в случае нехватки памяти MySQL может быть вынуждена сбрасывать кэши, чтобы освободить место, а спустя мгновение снова читать только что записанные на диск данные (это относится как к операциям чтения, так и к операциям записи). В таком случае недостаток памяти проявится как насыщение подсистемы ввода/вывода. Аналогично перегрузка шины памяти может выглядеть, как недостаток мощности ЦП. На самом деле, когда мы говорим, что «ЦП – узкое место» или что приложение «ограничено возможностями процессора», то имеем в виду, что оно занято главным образом вычислениями. Ниже мы займемся этим вопросом глубже.

Как выбирать процессор для MySQL

При модернизации или покупке нового оборудования необходимо решить, ограничена ли рабочая нагрузка возможностями процессора.

Определить, какова нагрузка, можно, посмотрев на уровень использования процессора, но лучше следить не за общей загруженностью ЦП, а попытаться изучить соотношение загруженности ЦП и подсистемы ввода/вывода для наиболее важных запросов, обращая особое внимание на то, все ли процессоры загружены равномерно. Чтобы понять, что именно ограничивает производительность сервера, можно воспользоваться такими инструментами, как *mpstat*, *iostat* и *vmstat*.

Что лучше: быстрый процессор или много процессоров?

Если рабочая нагрузка приводит к перегрузке ЦП, то, как правило, для MySQL лучше, когда имеется *более быстрый* процессор (в противоположность большему количеству процессоров).

Не всегда это так, поскольку все зависит от характера рабочей нагрузки и количества ЦП. Однако нынешняя архитектура MySQL плохо масштабируется на большое число процессоров, и MySQL не умеет распараллеливать выполнение одного запроса на несколько ЦП. В результате время обработки запроса с большим объемом вычислений ограничено именно быстродействием процессора.

Вообще говоря, существует два вида желательной производительности:

Низкая задержка (быстрое время отклика)

Для минимизации задержки нужны более быстрые ЦП, так как каждый запрос выполняется только на одном процессоре.

Высокая пропускная способность

Если одновременно выполняется много запросов, то их обслуживание можно ускорить за счет увеличения количества процессоров. Однако на практике это зависит от многих факторов. Так как MySQL плохо масштабируется с ростом числа процессоров, то часто лучше использовать меньшее количество более быстрых ЦП.

Если имеется несколько ЦП, а запросы выполняются не одновременно, то MySQL все же может задействовать дополнительные процессоры для таких фоновых задач, как вытеснение буферов InnoDB, сетевые операции и т. д. Но обычно эти задачи – ничто по сравнению с выполнением запросов. В системе с двумя процессорами, которая занята главным образом выполнением одного запроса с большим объемом вычислений, второй ЦП будет простаивать примерно 90% времени.

Репликация в MySQL (обсуждается в следующей главе) также выигрывает от наличия быстрого процессора, а не от большого числа процессоров. Если рабочая нагрузка ограничена мощностью процессора, то распараллеливание ее на главном сервере после сериализации легко может привести к такой нагрузке на подчиненный сервер, с которой тот не справится, даже если он мощнее главного. Впрочем, обычно узким местом на подчиненном сервере оказывается подсистема ввода/вывода, а не процессор.

Если рабочая нагрузка ограничена возможностями процессора, есть и другой подход к вопросу о том, что лучше: более быстрый процессор или большее число ЦП? Необходимо проанализировать, что именно делают запросы. На аппаратном уровне запрос может либо выполняться, либо находиться в состоянии ожидания. Наиболее распространенными причинами ожидания являются пребывание в очереди на выполнение (когда процесс готов выполняться, но все процессоры заняты), ожидание защелки или блокировки и ожидание завершения операции ввода/вывода или сети. Подумайте, чего ждут запросы. Если они стоят в очереди на выполнение либо ожидают освобождения защелки или блокировки, то, как правило, помогает увеличение быстродействия процессора (бывают и исключения, например ожидание мьютекса, защищающего буфер журнала InnoDB, который не освобождается до полного завершения операции ввода/вывода, – это может указывать на недостачную пропускную способность подсистемы ввода/вывода).

При этом MySQL может эффективно задействовать несколько процессоров для некоторых разновидностей рабочей нагрузки. Пусть, например, имеется множество соединений, выполняющих запросы к разным таблицам (следовательно, не возникает конкуренции за табличные блокировки – проблема, характерная для таблиц типа MyISAM и Memory), и общая пропускная способность сервера важнее времени отклика для отдельных запросов. При таком сценарии пропускная способность может быть очень высока, поскольку все потоки работают одновременно, не конкурируя между собой. Но опять-таки заметим, что на практике все может обстоять хуже, чем в теории: в InnoDB проблемы масштабирования проявляются вне зависимости от того, обращаются запросы к одной таблице или к разным, а в MyISAM имеются глобальные блокировки на каждый буфер ключей. Примером запроса, который может выполняться одновременно с ему подобными безо всякой конкуренции, является полное сканирование таблицы типа MyISAM.

Компания MySQL AB объявила, что подсистема хранения Falcon проектировалась так, чтобы в полной мере задействовать возможности серверов, имеющих, по меньшей мере, восемь ЦП, так что в будущем MySQL, возможно, сумеет использовать процессоры более эффективно, чем сейчас. Однако только время и опыт покажут, насколько хорошо в действительности масштабируется подсистема Falcon.

Архитектура ЦП

Ныне 64-разрядные архитектуры распространены куда шире, чем всего несколько лет назад. MySQL неплохо работает на 64-разрядных процессорах, хотя некоторые его внутренние компоненты еще не переписаны для полной поддержки такой архитектуры. Например, в версии MySQL 5.0 объем буфера ключей MyISAM ограничен 4 Гбайт – размер, который можно адресовать 32-разрядным числом (впрочем, для преодоления этого ограничения можно создать несколько буферов ключей).

Мы рекомендуем при покупке нового оборудования отдавать предпочтение 64-разрядной архитектуре. Без 64-разрядного процессора и 64-разрядной операционной системы невозможно эффективно использовать оперативную память большого объема; хотя некоторые 32-разрядные системы и поддерживают очень большой объем памяти, они все же неспособны работать с ней столь же эффективно, как 64-разрядная система, поэтому и MySQL будет страдать от аналогичного недостатка.

Масштабирование на несколько процессоров и ядер

Где несколько ЦП можно задействовать с пользой, так это в системах оперативной обработки транзакций. В таких системах обычно приходится иметь дело с большим количеством мелких операций, которые можно выполнять на разных процессорах, поскольку они поступают по разным соединениям. В эту категорию попадает большинство веб-приложений.

На серверах для оперативной обработки транзакций (OLTP) обычно применяется подсистема хранения InnoDB, в которой есть ряд нерешенных проблем с конкурентным доступом при наличии нескольких ЦП. Однако узким местом может стать не только InnoDB. Любой разделяемый ресурс – потенциальный источник конкуренции. InnoDB привлекает столь пристальное внимание просто потому, что чаще всего применяется в условиях высокой конкуренции, но MyISAM выглядит ничуть не лучше, если ее как следует нагрузить, даже без какого-либо изменения данных. Многие узкие места, относящиеся к конкурентному доступу, например блокировки строк в InnoDB и табличные блокировки в MyISAM, невозможно устранить в принципе; единственное решение – выполнить операцию как можно быстрее, чтобы освободить блокировку и позволить работать тем потокам, которые ее ожидают. Неважно, сколько имеется процессоров, если все они вынуждены ждать освобождения одной-единственной блокировки. Таким образом, даже при наличии рабочей нагрузки с высокой степенью конкурентности иногда удается получить выигрыш от наличия более быстрых процессоров.

На самом деле, в СУБД есть два вида проблем, относящихся к конкурентному доступу, и для их решения нужно применять разные подходы.

Логическая конкуренция

Конкуренция за ресурсы, видимые приложению, например за блокировки на уровне строки или таблицы. Для решения подобных проблем требуются тактические ухищрения, например изменение логики приложения, использование другой подсистемы хранения, изменение конфигурации сервера или применение других блокировочных подсказок оптимизатору или уровней изоляции транзакций.

Внутренняя конкуренция

Конкуренция за такие ресурсы, как семафоры, доступ к страницам пула буферов InnoDB и т. д. Можно попробовать найти обходное ре-

шение путем изменения конфигурации сервера, параметров операционной системы или установки другого оборудования, но, как правило вам придется смириться с имеющимися ограничениями. Иногда проблему удается смягчить, воспользовавшись другой подсистемой хранения или наложив заплату на уже используемую.

Количество процессоров, которые MySQL может эффективно задействовать, и масштабируемость MySQL при возрастании нагрузки зависят как от характера рабочей нагрузки, так и от системной архитектуры. Говоря о «системной архитектуре», мы имеем в виду операционную систему и оборудование, а не приложение, в котором используется MySQL. На масштабируемость MySQL оказывают влияние такие факторы, как архитектура процессора (RISC, CISC, глубина конвейера и т. д.), модель процессора и операционная система. Именно поэтому так важно эталонное тестирование: некоторые системы продолжают прекрасно работать при увеличении степени конкурентности, тогда как показатели других резко ухудшаются.

Иногда увеличение количества процессоров даже приводит к ухудшению общей производительности. Причем это весьма распространенная проблема: мы знаем много людей, которые пытались модернизировать 4-ядерную систему до 8-ядерной, но были вынуждены откатиться (или предоставить в распоряжение процесса MySQL только четыре ядра из восьми) из-за снижения производительности. Вы должны прогнать эталонные тесты и посмотреть, как система ведет себя при данной рабочей нагрузке.

Некоторые узкие места, препятствующие масштабируемости, находятся в самом сервере, тогда как другие – на уровне подсистем хранения. Очень важно, для чего проектировалась конкретная подсистема хранения; иногда переход на другую подсистему позволяет более эффективно задействовать большее количество ЦП.

Войны за увеличение быстродействия ЦП, полыхавшие на рубеже столетий, сейчас несколько поутихли, производители процессоров теперь больше озабочены увеличением количества ядер и такими вариациями на эту тему, как гипетрединг (multithreading). Вполне может статься, что процессор будущего будет обладать несколькими сотнями ядер; уже сегодня 4-ядерный ЦП не вызывает удивления. Внутренняя архитектура процессоров разных производителей настолько сильно отличается, что невозможно высказать общие соображения о взаимодействии между потоками, процессорами и ядрами. Очень важно и то, как спроектированы память и шина доступа к ней. Ну и, наконец, от архитектуры зависит и ответ на вопрос, что лучше: несколько ядер или несколько физических процессоров.

Поиск баланса между памятью и дисками

Иметь много памяти важно, прежде всего, не для того, чтобы уместить в ней как можно больше данных, а для того, чтобы избежать доступа

к диску, который на несколько порядков медленнее, чем доступ к оперативной памяти. Хитрость в том, чтобы найти правильный баланс между объемом оперативной и дисковой памяти, быстродействием, стоимостью и другими характеристиками, которые обеспечили бы высокую производительность при данной рабочей нагрузке. Но, прежде чем заняться решением этой задачи, вернемся ненадолго к основам.

Любой компьютер содержит «пирамиду» кэшей, причем на каждом уровне объем уменьшается, а быстродействие и стоимость растут (рис. 7.1).

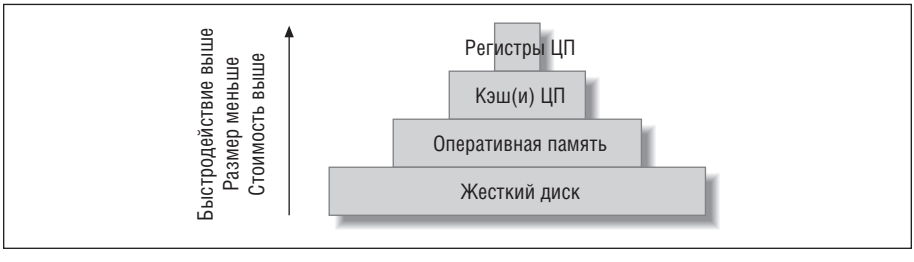


Рис. 7.1. *Иерархия кэшей*

Чем выше в иерархии находится кэш, тем он лучше приспособлен для хранения часто используемых данных с целью увеличения скорости доступа к ним. Обычно применяются такие эвристики, как «данные, к которым обращались недавно, скорее всего, потребуются снова» или «данные, расположенные близко к недавно использованным, вероятно, тоже скоро понадобятся». Подобные эвристики работают благодаря пространственной и временной *локальности ссылок*.

Для программиста регистры ЦП и кэши прозрачны и архитектурно-зависимы. Ими управляет компилятор и сам процессор. Однако разница между оперативной памятью и жестким диском для программиста весьма ощутима, и программы работают с этими видами памяти совершенно по-разному[1].

В особенности это относится к серверам баз данных, поведение которых зачастую идет вразрез с вышеупомянутыми эвристиками. Удачно спроектированный кэш базы данных (такой, как пул буферов в InnoDB) обычно оказывается эффективнее кэша операционной системы, ориентированного на задачи общего характера. Кэш базы данных знает о самих данных гораздо больше и его логика «заточена» под обслуживание специфических потребностей СУБД. Кроме того, для доступа к данным, хранящимся в кэше БД, не нужен системный вызов.

Вследствие указанных особенностей специализированного кэша вы должны настраивать иерархию кэшей в соответствии с типичными спо-

[1] Впрочем, программы могут полагаться на то, что операционная система кэширует в памяти большой объем данных, которые концептуально находятся «на диске». Именно так, кстати, и поступает MySQL.

собами доступа к данным в базе. Поскольку регистры и кэши на материнской плате не могут быть сконфигурированы пользователем, то в вашем распоряжении остается только оперативная память и жесткий диск.

Произвольный и последовательный ввод/вывод

В серверах баз данных применяется как последовательный, так и произвольный ввод/вывод, причем наибольший выигрыш от кэширования получается в последнем случае. В этом легко убедиться, мысленно представив себе смешанную рабочую нагрузку, в которой сочетаются операции поиска одиночных строк и поиска по диапазону, возвращающему множество строк. Как правило, часто используемые («горячие») данные распределены случайным образом, поэтому их кэширование позволяет избежать дорогостоящих операций поиска на диске. Напротив, при последовательном доступе данные обычно считываются только один раз, поэтому кэшировать их бессмысленно (разве что они целиком помещаются в память).

Операции последовательного чтения мало выигрывают от кэширования еще и потому, что они выполняются быстрее операций произвольного чтения. Тому есть две причины.

Последовательный ввод/вывод быстрее произвольного

Последовательные операции выполняются быстрее произвольных и в оперативной памяти, и на диске. Предположим, что диск способен выполнить 100 операций ввода/вывода с произвольной выборкой в секунду и последовательно прочитать 50 Мбайт в секунду (примерно такие показатели характерны для большинства современных дисков потребительского класса). Если длина строки составляет 100 байтов, то при произвольном доступе можно будет прочитать 100 строк в секунду, а при последовательном – 500 000. Разница в 5000 раз, то есть на несколько порядков. Поэтому при таком сценарии кэширование результатов операций произвольного доступа оказывается гораздо полезнее.

Последовательный доступ к строкам в оперативной памяти также быстрее произвольного. Современные микросхемы памяти обычно способны прочитать в секунду примерно 250 000 строк длиной 100 байтов при доступе с произвольной выборкой или 5 миллионов таких же строк последовательно. Отметим, что произвольный доступ к памяти в 2500 раз быстрее такого же доступа к диску, тогда как последовательный быстрее всего лишь в 10 раз.

Подсистемы хранения могут выполнять последовательное чтение быстрее произвольного

Для произвольного доступа подсистема хранения обычно должна воспользоваться индексом. Из этого правила есть исключения, но для InnoDb и MyISAM оно справедливо. Как правило, это требует навигации по B-дереву и сравнения значений. С другой стороны, после-

довательное чтение обычно сводится к обходу гораздо более простой структуры данных, например связанного списка. Работы в этом случае гораздо меньше, а значит, последовательное чтение происходит быстрее.

Кэширование результатов последовательного чтения позволяет кое-что сэкономить, но эффективность кэширования результатов произвольного чтения многократно выше. Иными словами, *для решения проблем, возникающих из-за ввода/вывода с произвольным доступом, лучше всего добавить память* (если есть такая возможность).

Кэширование, чтение и запись

Если памяти достаточно, то при чтении можно вообще обойтись без доступа к диску. Коль скоро все данные помещаются в памяти, то после «прогрева» сервера любая операция чтения будет удовлетворяться из кэша. *Логические операции чтения*, конечно же, останутся, но *физических* не будет. С записью же все обстоит по-другому. Операцию записи, как и операцию чтения, можно выполнить в памяти, но рано или поздно данные должны быть зафиксированы на диске. Иными словами, кэш позволяет отложить запись на диск, но не устранить ее полностью, как в случае чтения.

Помимо откладывания записи кэш позволяет группировать несколько операций двумя важными способами.

Много операций записи, одна операция сброса

Один и тот же элемент данных может быть многократно изменен в памяти без записи на диск всех новых значений. Когда в конечном итоге данные все же сбрасываются на диск, то фиксируется результат всех модификаций, имевших место с момента последней физической записи. Например, хранящийся в памяти счетчик может обновляться несколькими командами. Если он был увеличен 100 раз, а затем записан на диск, то получается, что 100 изменений сгруппированы в одну операцию физической записи.

Объединение операций ввода/вывода

В памяти можно модифицировать несколько разных элементов данных, а затем собрать все изменения вместе и выполнить их одной операцией физической записи на диск.

Именно поэтому во многих транзакционных системах применяется *упреждающая запись в журнал*. Эта технология позволяет производить изменения в страницах, хранящихся в оперативной памяти, не сбрасывая их на диск, так как последнее потребовало бы операций ввода/вывода с произвольным доступом, что очень медленно. Вместо этого протокол изменений записывается в последовательный файл журнала, – данная операция выполняется гораздо быстрее. Впоследствии фоновый поток записывает модифицированные страницы в нужное место, причем попутно может оптимизировать запись.

Существенно ускорить операции записи позволяет буферизация, поскольку она преобразует произвольный ввод/вывод в последовательный. Асинхронная (буферизованная) запись обычно возлагается на операционную систему, которая может производить пакетный сброс на диск наиболее оптимальным образом. Именно поэтому такой выигрыш дает буферизация на уровне RAID-контроллера, оборудованного кэшем записи с резервным электропитанием (мы будем обсуждать технологию RAID ниже).

Что такое рабочее множество?

У каждого приложения есть «рабочее множество» данных, то есть те данные, которые ему реально нужны для работы. В большинстве баз данных имеется также информация – и много, – не входящая в рабочее множество. Базу можно представлять себе как стол с выдвижными ящиками. Рабочее множество состоит из тех документов, которые должны лежать на столе для работы. В этом случае поверхность стола является аналогом оперативной памяти, а ящики – аналогами жестких дисков.

Чтобы выполнить свою работу, вам вовсе необязательно выкладывать на стол *все* бумажки. Точно так же для достижения оптимальной производительности не нужно загружать в память всю базу данных – достаточно рабочего множества.

Размер рабочего множества существенно зависит от приложения. Для одних программ рабочее множество составляет всего 1% от общего объема данных, а для других приближается к 100%. Если рабочее множество не умещается целиком в памяти, то сервер вынужден перемещать информацию между диском и памятью. Поэтому нехватка памяти может выглядеть как проблема с вводом/выводом. Иногда поместить все рабочее множество в память в принципе невозможно, а иногда это и не нужно (например, если приложение занято преимущественно последовательным вводом/выводом). Вся архитектура приложения может существенно зависеть от того, можно ли поместить рабочее множество целиком в оперативную память.

При более тщательном анализе выявляется расплывчатость термина «рабочее множество». Например, возможно, что в течение каждого часа производится доступ только к 1% всех данных, но за 24 часа это может вылиться в 20%. Что в такой ситуации называть рабочим множеством? Быть может, полезнее рассуждать о рабочем множестве, как об объеме данных, которые необходимо кэшировать для того, чтобы рабочая нагрузка была ограничена лишь процессорными мощностями. Если вы не можете поместить в кэш достаточно информации, значит, рабочее множество не умещается в памяти.

Рабочее множество и единица кэширования

Рабочее множество состоит из данных и индексов, а исчислять его следует в *единицах кэширования*. Единицей кэширования называется

наименьший объем данных, которым может оперировать подсистема хранения. У разных подсистем хранения величина единицы кэширования различна, а потому различен и размер рабочего множества. Например, InnoDB оперирует только страницами размером 16 Кбайт. Если при поиске одиночной строки InnoDB должна обратиться к диску, то в пул буферов будет считана вся содержащая ее страница. Иногда это расточительно.

Предположим, что произвольным образом считываются 100-байтные строки. При этом InnoDB запомнит в кэше очень много ненужных данных, так как для каждой строки придется прочитать и сохранить полную страницу длиной 16 Кбайт. А так как в рабочее множество входят и индексы, то InnoDB вынуждена будет прочитать и закэшировать также и те части дерева индекса, которые требуются для поиска строки. Длина индексных страниц в InnoDB также составляет 16 Кбайт, следовательно, для доступа всего лишь к одной 100-байтной строке может потребоваться сохранить в кэше 32 Кбайта (а то и больше – в зависимости от глубины дерева индекса). Поэтому единица кэширования – это еще одна причина, по которой в InnoDB так важно правильно выбирать кластерные индексы. Кластерный индекс позволяет не только оптимизировать доступ к диску, но и хранить взаимосвязанные данные в пределах одной страницы, из-за чего в кэше помещается более весомая доля рабочего множества.

С другой стороны, в подсистеме хранения Falcon единицей кэширования является не страница, а строка. Поэтому Falcon может оказаться эффективнее для кэширования небольших, случайно распределенных строк. Возвращаясь к метафоре рабочего стола, можно сказать, что InnoDB требует достать из стола целую папку (страницу базы данных) всякий раз, когда возникает необходимость в одной находящейся в ней бумажке. Если кластерного индекса нет (или он выбран неудачно), то это будет крайне неэффективно. Подсистема же Falcon позволяет извлечь из папки только нужный листок, не выкладывая ее целиком на стол.

У каждого подхода есть свои плюсы и минусы. Например, InnoDB хранит в памяти всю страницу длиной 16 Кбайт, поэтому если впоследствии понадобится другая строка из той же страницы, то она уже тут как тут. А Falcon поддерживает два кэша: строк и страниц, поэтому налицо оба преимущества: кэш страниц сокращает количество операций доступа к диску, а кэш строк позволяет более эффективно использовать память. Однако дуальный кэш по сути своей расточителен, так как некоторые данные содержатся в памяти в двух экземплярах. Этот механизм называется *двойной буферизацией*.

Теоретически любая стратегия может оказаться очень эффективной для одной рабочей нагрузки и совсем неэффективной для другой. Как обычно, решение зависит от того, с чем выбранная подсистема хранения справляется наилучшим образом.

Отыскание эффективного соотношения память–диск

Приемлемое соотношение объема оперативной и дисковой памяти лучше всего определять путем экспериментирования или эталонного тестирования. Если можно загрузить в оперативную память вообще все, то думать больше не о чем. Но, как правило, это не так, поэтому необходимо прогнать эталонные тесты для некоторого подмножества данных и посмотреть, что получится. Ваша цель – найти приемлемый *коэффициент непопадания в кэш*. Непопадание имеет место, когда для выполнения запроса нужны данные, отсутствующие в кэше, так что серверу приходится читать их с диска.

Коэффициент непопадания в кэш определяет то, насколько эффективно задействован процессор, поэтому для его оценки проще всего взглянуть на показатели использования ЦП. Например, если 90% всего времени ЦП работает, а остальные 10% ожидает завершения ввода-вывода, то коэффициент непопадания можно считать хорошим.

Рассмотрим, как рабочее множество влияет на коэффициент непопадания. Важно понимать, что рабочее множество – не просто число; в действительности оно представляет собой статистическое распределение, по отношению к которому коэффициент непопадания нелинеен. Например, если имеется 10 Гбайт памяти, а коэффициент непопадания составляет 10%, то может показаться, что стоит добавить еще 11% памяти[1], и коэффициент непопадания обратится в нуль. Но на самом деле из-за таких деталей, как размер единицы кэширования, для достижения коэффициента непопадания 1% может потребоваться аж 50 Гбайт памяти. И даже при точном совпадении с единицей кэширования теоретическая оценка может оказаться неверной: ситуация нередко осложняется, например из-за порядка доступа к данным. Чтобы получить коэффициент непопадания 1%, может не хватить и 500 Гбайт памяти!

Очень легко увлечься оптимизацией того, что не дает большого выигрыша. Например, значение коэффициента непопадания в 10% может уже означать, что ЦП занят 80% времени, а это совсем неплохо. Предположим, что путем добавления памяти вы смогли сократить коэффициент непопадания до 5%. Сильно упрощая картину, можно сказать, что вы «подбросили» процессору еще 6% данных. Пойдя еще на одно упрощение, скажем, что вы довели коэффициент использования процессора до 84,8%. Но это не такая уж большая победа, если принять во внимание деньги, потраченные на приобретение необходимой для этого памяти. К тому же на самом деле различия в скорости доступа к памяти и к диску, характер обработки данных процессором и целый ряд других факторов могут привести к тому, что снижение коэффициента непопадания до 5% вообще не изменит коэффициента использования ЦП.

---

[1] Именно 11%, а не 10%. Если коэффициент непопадания равен 10%, то коэффициент попадания – 90%, поэтому необходимо разделить 10 Гбайт на 90%, что дает 11,111 Гбайт.

Поэтому мы в самом начале сказали, что стремиться нужно к *приемлемому* коэффициенту непопадания в кэш, а не к нулевому. Невозможно точно указать желаемое значение, поскольку что считать «приемлемым» зависит от конкретного приложения и рабочей нагрузки. Некоторые задачи прекрасно ведут себя при коэффициенте непопадания 1%, тогда как для нормальной работы других необходим коэффициент 0,01%. «Хороший коэффициент непопадания в кэш», как и «рабочее множество», – расплывчатое понятие, а тот факт, что подсчитать коэффициент непопадания можно разными способами, только усложняет дело.

Оптимальное отношение память-диск зависит и от других компонентов системы. Предположим, что в вашем компьютере есть 16 Гбайт оперативной памяти, 20 Гбайт данных и очень много свободного места на диске. Система прекрасно работает, и процессор загружен на 80%. Пусть объем данных увеличился вдвое; что нужно сделать для поддержания прежнего уровня производительности? Первое, что приходит в голову, – удвоить количество процессоров и объем памяти. Однако даже если все компоненты системы идеально масштабируются с учетом возросшей нагрузки (совершенно нереалистичное допущение), то такое решение, скорее всего, ничего не даст. Система, в которой хранится 20 Гбайт данных, вероятно, использует более 50% пропускной способности какого-то компонента, например, не исключено, что количество операций ввода/вывода уже находится на уровне 80% от максимума. Справиться с удвоенной нагрузкой она уже не сможет. Таким образом, наилучшее отношение память-диск определяется самым слабым компонентом системы.

Выбор жестких дисков

Если нужное количество данных поместить в память не удается, например, анализ показал, что при имеющейся подсистеме ввода/вывода для полной загрузки процессора необходимо 500 Гбайт памяти, то стоит подумать о приобретении более мощной подсистемы ввода/вывода, пусть даже за счет ОЗУ. А приложение нужно проектировать с учетом задержек ввода/вывода.

Этот подход может показаться противоречащим интуиции. Ведь совсем недавно мы говорили, что дополнительная память может снизить нагрузку на подсистему ввода/вывода и сократить время ожидания. Так зачем же увеличивать мощность этой подсистемы, если проблему можно решить добавлением памяти? А все дело в балансе между различными факторами: соотношением операций чтения и записи, длиной каждой операции ввода/вывода, количеством таких операций в секунду. Например, если нужно, чтобы запись в журнал производилась быстро, то невозможно исключить из уравнения диск простым увеличением объема оперативной памяти. В таком случае лучше потратить деньги на приобретение высокопроизводительной подсистемы ввода/вывода, оборудованной кэшем записи с резервным питанием.

Напомним, что операция считывания с обычного жесткого диска состоит из трех шагов.

1. Подвести головку считывания к нужной дорожке.
2. Ждать, пока в результате вращения диска нужные данные не окажутся под головкой.
3. Ждать, пока все нужные данные не пройдут под головкой.

Скорость выполнения этих операций диском оценивается двумя показателями: *время доступа* (шаги 1 и 2 вместе) и *скорость передачи*. Эти же два числа определяют *задержку* и *пропускную способность*. Что важнее – время доступа, скорость передачи или сочетание того и другого – зависит от характера выполняемых запросов. Если говорить о полном времени, необходимом для завершения чтения с диска, то время поиска произвольной одиночной строки определяется в первую очередь шагами 1 и 2, а последовательное считывание большого объема данных – шагом 3.

Есть еще ряд факторов, которые могут повлиять на выбор дисков, а какие из них существенны, зависит от конкретного приложения. Представьте, что требуется выбрать диски для какого-нибудь онлайнового приложения, например популярного новостного сайта, для которого характерно большое количество операций произвольного чтения, возвращающих сравнительно мало данных. Тогда стоит принять во внимание следующие факторы:

Емкость диска

Для оперативных приложений это редко бывает камнем преткновения, так как емкости современных дисков более чем достаточно. Если это не так, то общепринятой практикой является объединение дисков в RAID-массив[1].

Скорость передачи

Мы уже видели выше, что современные диски способны передавать данные очень быстро. Точная величина зависит главным образом от скорости вращения шпинделя и плотности записи данных на поверхности диска, а также от ограничений интерфейса с хост-компьютером (многие современные диски могут читать данные быстрее, чем интерфейс способен их передавать). Так или иначе, не скорость передачи лимитирует производительность оперативных приложений, поскольку последние обычно выполняют много коротких операций с произвольным доступом.

[1] Интересно отметить, что некоторые сознательно покупают большие диски, но используют лишь 20–30% их емкости. Тем самым повышается локальность данных и сокращается время подвода головки, что иногда оправдывает более высокую цену.

Время доступа

Обычно именно этот параметр определяет производительность произвольной выборки, поэтому нужно искать диски с минимальным временем доступа.

Скорость вращения шпинделя

Сейчас наиболее часто встречаются диски с частотой вращения 7200, 10000 и 15000 оборотов в минуту. Скорость как последовательной, так и произвольной выборки в немалой степени зависит от скорости вращения.

Габариты

При прочих равных условиях габариты диска тоже играют роль: чем меньше диск, тем меньше времени занимает перемещение головки для считывания. Диски диаметром 2.5 дюйма, предназначенные для серверов, часто оказываются быстрее своих более крупных собратьев. К тому же они потребляют меньше электроэнергии, и в один блок (стоечный юнит) помещается больше дисков.

Технологии производства дисков быстро прогрессируют, поэтому наши рекомендации могут уже устареть. Так во время работы над книгой горячо обсуждались твердотельные накопители. Принцип их работы не имеет ничего общего с вращающимися накопителями. Но пока они очень дороги и распространены не слишком широко. Нам известно несколько проектов, в которых такие диски успешно применяются, но наш собственный опыт недостаточен для того, чтобы давать конкретные рекомендации.

Как и в случае с ЦП, возможности масштабирования MySQL на несколько дисков зависят от подсистемы хранения и от рабочей нагрузки. InnoDB обычно хорошо масштабируется на 10–20 дисков. С другой стороны, степень масштабирования подсистемы MyISAM при записи ограничивается табличными блокировками, поэтому, если для рабочей нагрузки, включающей много операций записи, используется MyISAM, то существенного выигрыша от наличия большого количества дисков не добиться. В какой-то мере могут помочь буферизация на уровне операционной системы и фоновое распараллеливание записи, но все же масштабируемость записи в MyISAM принципиально хуже, чем в InnoDB.

Как и в случае процессоров, больше дисков не всегда лучше. Некоторые приложения, для которых важна низкая задержка, нуждаются в более быстрых дисках, а не в большем их количестве. Например, производительность репликации обычно выше при использовании скоростных дисков, поскольку обновления на подчиненном сервере производятся одним потоком. Чтобы понять, даст ли увеличение количества дисков какой-нибудь эффект при конкретной рабочей нагрузке, имеет смысл посмотреть на показатели загруженности дисков, которые выдает ути-

лита *iostat*. Если количество ожидающих запросов велико, то дополнительные диски не помешали бы. В конце главы мы поместили несколько примеров выдачи *iostat*.

Выбор оборудования для подчиненного сервера

При выборе оборудования для подчиненного сервера репликации руководствуются в основном теми же принципами, что и для главного сервера, хотя есть и некоторые отличия. Если вы планируете использовать подчиненный сервер репликации для переключения на него в случае сбоя основного, то, как правило, он должен быть не менее мощным, чем главный сервер. И вне зависимости от того, будет ли подчиненный сервер выступать в роли резервной замены главного, он должен быть достаточно мощным для выполнения всех операций записи, произведенных на главном сервере, с учетом того, что они должны еще и выполняться последовательно (дополнительная информация по этому поводу приведена в следующей главе).

Основное соображение при выборе оборудования для подчиненного сервера – стоимость: нужно ли тратить на него столько же денег, сколько на главный сервер? Можно ли сконфигурировать подчиненный сервер по-другому, чтобы выжать из него дополнительную производительность?

Зависит от обстоятельств. Если подчиненный сервер используется в качестве резервного, то оборудование и конфигурацию главного и подчиненного сервера имеет смысл делать одинаковыми. Но если единственное назначение репликации – увеличить пропускную способность системы в части операций чтения, то возможны самые разные компромиссы. Например, на подчиненном сервере можно использовать другую подсистему хранения, оборудовать его более дешевыми компонентами или сконфигурировать RAID-массив уровня 0, а не 5 или 10. Кроме того, можно пожертвовать некоторыми гарантиями непротиворечивости и долговечности, чтобы подчиненному серверу приходилось делать меньше работы. Дополнительную информацию см. в разделе «Настройка ввода/вывода в MySQL» на стр. 351.

На больших объемах эти меры могут оказаться рентабельными, но на малых лишь усложняют работу.

Оптимизация производительности с помощью RAID

Подсистемы хранения часто размещают все данные и индексы в одном большом файле, а это означает, что для хранения данных наиболее под-

ходящим вариантом является технология RAID[1]. Она может решить проблемы резервирования, емкости, кэширования и быстродействия. Но, как и для всех остальных рассматриваемых оптимизаций, RAID-массив можно конфигурировать по-разному, и важно выбрать уровень, отвечающий именно вашим потребностям.

Мы не станем здесь ни разбирать все уровни RAID, ни вдаваться в детали того, как организовано хранение данных на каждом уровне. Эта тема прекрасно изложена в различных книгах и в сети[2]. Мы же займемся вопросом о том, как различные конфигурации RAID удовлетворяют требования, предъявляемые СУБД. Ниже перечислены наиболее важные уровни RAID.

RAID 0

Уровень RAID 0 – самая дешевая и наиболее производительная конфигурация RAID, по крайней мере, при упрощенном подходе к оценке стоимости и производительности (если включить в рассмотрение также и восстановление данных, то картина перестает быть такой радужной). Поскольку этот уровень не обеспечивает никакой избыточности, то мы рекомендуем его лишь для серверов, которые вам более-менее безразличны, например для подчиненных серверов или серверов, которые по той или иной причине считаются «одноразовыми». Типичный пример – подчиненный сервер, который легко клонировать из другого подчиненного сервера.

Еще раз подчеркнем, что уровень RAID 0 *не обеспечивает никакой избыточности*, несмотря на то, что в акрониме RAID первая буква означает «redundant» (избыточный, или резервированный). На самом деле, вероятность выхода из строя RAID-массива уровня 0 даже *выше*, а не ниже вероятности отказа одиночного диска!

RAID 1

Уровень RAID 1 обеспечивает неплохую производительность во многих ситуациях, а, так как данные дублируются, то присутствует также и избыточность. На операциях чтения RAID 1 чуть быстрее,

---

[1] Неплохой альтернативой является также секционирование (см. главу 5), поскольку оно позволяет разбить один большой файл на много более мелких, которые можно разместить на разных устройствах. Однако по сравнению с секционированием RAID оказывается более простым решением в случае сверхбольших объемов данных. От пользователя не требуется балансировать нагрузку вручную или вмешиваться, когда распределение нагрузки изменяется. Кроме того, эта технология обеспечивает избыточность, которую не получить за счет разнесения секций по разным дискам.

[2] Упомянем лишь два хороших источника информации по **RAID: статью в википедии** (*http://ru.wikipedia.org/wiki/RAID*) и учебное руководство AC&NC по адресу *http://www.acnc.com/04_00.html*.

чем RAID 0. Он хорошо подходит для серверов, занятых журналированием и другими аналогичными операциями, поскольку в случае последовательной записи редко бывает необходимо, чтобы все составляющие массив диски показывали высокое быстродействие (в отличие от произвольной записи, для которой распараллеливание может дать заметный эффект). Этот уровень также часто выбирают для маломощных серверов, которым нужна избыточность, но жестких дисков всего два.

Уровни RAID 0 и RAID 1 очень просты и часто реализуются программно. Многие операционные системы предоставляют простые средства для создания томов типа RAID 0 или RAID 1.

RAID 5

Уровень RAID 5 выглядит более сложным, но для некоторых приложений это единственный выход – вследствие ценовых параметров или ограничений на количество дисков, которые физически можно разместить в сервере. В этом случае данные распределяются по нескольким дискам и дополнительно поддерживаются распределенные блоки с контрольной суммой, так что при отказе одного диска хранившаяся на нем информация может быть восстановлена по данным на других дисках и контрольным блокам. С точки зрения цены за единицу хранения, это самая экономичная из всех конфигураций с избыточностью, поскольку из всего массива на обеспечение избыточности расходуется пространство эквивалентное одному диску.

Произвольная запись на уровне RAID 5 – дорогостоящая операция, так как реально требуется произвести две процедуры записи и две операции RAID для блоков с контрольной суммой. Процедуры записи выполняются чуть быстрее, если они последовательные или количество физических дисков велико. С другой стороны, чтение – как последовательное, так и произвольное – производится весьма быстро. Уровень RAID – приемлемый выбор для томов, содержащих только данные или данные и журналы, при различных характеристиках рабочей нагрузки.

В случае отказа диска накладные расходы на уровне RAID 5 максимальны, так как информацию приходится реконструировать путем чтения всех остальных дисков. Это весьма ощутимо сказывается на производительности. Если вы попытаетесь сохранить доступ к серверу во время процедуры реконструкции, то не ждите ни хорошего быстродействия, ни быстрого завершения восстановления. К другим недостаткам можно отнести ограничения масштабируемости из-за блоков с контрольной суммой (RAID 5 начинает хуже масштабироваться уже при 10 дисках) и проблемы кэширования. Производительность RAID 5 в значительной мере зависит от кэша на RAID-контроллере, который может вступать в конфликт с потребностями СУБД. Мы вернемся к этой теме чуть позже.

Один из факторов, помогающих примириться с уровнем RAID 5, – его широкая распространенность. Поэтому RAID-контроллеры часто оптимизируют именно для этого уровня и, несмотря на теоретические ограничения, интеллектуальные контроллеры с кэшем при некоторых рабочих нагрузках работают почти так же хорошо, как контроллеры RAID 10. Правда, это может всего лишь свидетельствовать о недостаточной оптимизированности последних, но как бы то ни было, мы такую картину наблюдали.

RAID 10

Уровень RAID 10 – прекрасный выбор для хранения информации, если только он вам по средствам. Он состоит из зеркалированных пар с данными, записанными с чередованием (striped), так что отлично масштабируется и для чтения, и для записи. По сравнению с RAID 5 он работает и реконструируется очень быстро. К тому же, этот уровень довольно легко реализуется программно.

Падение производительности при отказе одного диска все равно остается заметным, поскольку узким местом становятся находящиеся на нем данные (stripe). В зависимости от рабочей нагрузки снижение производительности может достигать 50%. При выборе RAID-контроллера нужно обращать внимание, не используется ли в реализации RAID 10 «зеркалирование с конкатенацией» (concatenated mirror). Это не оптимальное решение, так как отсутствует чередование(striping): может оказаться, что данные, к которым сервер чаще всего обращается, находятся только на одной паре дисков, а не распределены по многим парам, а это снижает производительность.

RAID 50

Уровень RAID 50 состоит из «чередующихся» (striped) массивов RAID 5. Такое решение может явиться неплохим компромиссом между экономичностью RAID 5 и высокой производительностью RAID 10, если имеется достаточное количество дисков. Такой метод наиболее полезен для очень больших наборов данных, например при организации хранилищ или в необычайно крупных OLTP-системах.

Различные конфигурации RAID сведены в табл. 7.1.

Таблица 7.1. Сравнение уровней RAID

Уровень	Характеристики	Избыточность	Требуется дисков	Быстрое чтение	Быстрая запись
RAID 0	Дешевый, быстрый, ненадежный	Нет	N	Да	Да
RAID 1	Быстрое чтение, простой, надежный	Да	2 (обычно)	Да	Нет

Таблица 7.1 (продолжение)

Уровень	Характеристики	Избыточность	Требуется дисков	Быстрое чтение	Быстрая запись
RAID 5	Компромисс между надежностью, скоростью и ценой	Да	N + 1	Да	По-разному
RAID 10	Дорогой, быстрый, надежный	Да	2N	Да	Да
RAID 50	Для сверхбольших наборов данных	Да	2(N + 1)	Да	Да

Отказ, восстановление и мониторинг RAID

Все уровни RAID, кроме RAID 0, обеспечивают избыточность. Это, конечно, важно, но не следует недооценивать вероятность отказа сразу нескольких дисков. Не нужно думать, что RAID дает полную гарантию сохранности данных.

RAID не отменяет – и даже не уменьшает – необходимость резервного копирования. В случае возникновения проблемы время восстановления зависит от контроллера, уровня RAID, размера массива, скорости дисков и от того, требуется ли сохранять доступ к данным во время реконструкции массива.

Может случиться, что несколько дисков откажут одновременно. Например, всплеск напряжения или перегрев вполне способны вывести из строя два и более диска. Но чаще бывает, что два отказа разделены небольшим промежутком времени. Нередко это остается незамеченным. Типичная ситуация – повреждение поверхности носителя в том месте, где хранятся редко используемые данные. Дефект может не проявляться много месяцев, пока кто-то не попытается прочитать данные или RAID-контроллер не воспользуется ими для реконструкции массива. Чем диск больше, тем более вероятно такое развитие событий.

Именно поэтому так важно вести мониторинг состояния RAID-массивов. Обычно вместе с контроллером поставляется программное обеспечение для формирования отчетов о состоянии массива, и им надо пользоваться, поскольку в противном случае вы так и будете пребывать в неведении относительно отказа диска. Возможность вовремя восстановить данные будет упущена, а о проблеме вы узнаете только в момент отказа второго диска, когда уже слишком поздно принимать меры.

Уменьшить риск можно путем регулярных активных проверок исправности массива. Кроме того, в некоторых контроллерах реализована функция Background Patrol Read (фоновое контрольное чтение), которая ищет дефекты дисков и устраняет их, не прерывая доступа к данным. Но, как и в случае восстановления, проверка очень больших массивов может занимать длительное время, поэтому планируйте ее заблаговременно.

Можно также добавить диск для «горячей замены» (hot spare), который не используется, а сконфигурирован как резервный, чтобы контроллер мог автоматически задействовать его для восстановления. Это разумная мысль, если все серверы очень важны. Для серверов всего с двумя-тремя дисками такое решение дороговато, но если дисков много, то просто глупо не раскошелиться еще на один для горячей замены. Не забывайте, что вероятность отказа быстро возрастает с увеличением количества дисков.

Выбор между аппаратной и программной реализациями RAID

Между операционной системой, файловой системой и количеством дисков, которые видит операционная система, существуют сложные зависимости. Ошибки, ограничения, да и просто неверная конфигурация могут привести к тому, что производительность будет сильно недотягивать до теоретически возможной величины.

Если имеется 10 жестких дисков, то в идеале они могли бы параллельно обслуживать 10 запросов, но иногда файловая система, операционная система или RAID-контроллер сериализуют запросы. Одно из возможных решений этой проблемы – попробовать различные конфигурации RAID. Например, если вы хотите воспользоваться зеркалированием для обеспечения избыточности и высокой производительности, то можно сконфигурировать имеющиеся 10 дисков следующими способами:

- Сконфигурировать один том уровня RAID 10, состоящий из пяти зеркалированных пар. Операционная система будет видеть единый большой том, а RAID-контроллер скроет 10 составляющих его дисков
- Сконфигурировать пять зеркалированных пар уровня RAID 1, так что операционная система будет адресовать пять томов вместо одного
- Сконфигурировать пять зеркалированных пар уровня RAID 1 в RAID-контроллере, а затем использовать программную реализацию RAID 0, чтобы представить все пять томов в виде одного логического тома. В результате получится массив RAID 10, реализованный отчасти программно, а отчасти аппаратно.

Какой вариант предпочесть? Зависит от взаимодействия между всеми компонентами системы. Возможно, эти конфигурации будут работать с одинаковой производительностью, а, возможно, и нет.

Мы наблюдали сериализацию в различных конфигурациях. Один такой пример (в устаревшем дистрибутиве GNU/Linux) – комбинация файловой системы ext3 и подсистемы хранения InnoDB с параметром `innodb_flush_method=O_DIRECT`. Похоже, что эта конфигурация приводила к блокировкам на уровне индексных узлов в файловой системе, поэтому в каждый момент времени одному файлу мог быть отправлен толь-

ко один запрос на ввод/вывод. В данном случае имела место сериализация на уровне файла, а ошибка была исправлена в следующей версии системы.

Аналогичная ситуация наблюдалась, когда мы работали с томом RAID 10 из 10 дисков, файловой системой ReiserFS и подсистемой хранения InnoDB с включенным параметром `innodb_file_per_table`, но теперь сериализации подвергались запросы к каждому *устройству*. После перехода на уровень RAID 0 поверх аппаратного RAID 1 пропускная способность возросла в пять раз, поскольку подсистема хранения стала вести себя так, будто вместо одного диска появилось пять. Это явление тоже было вызвано ошибкой, впоследствии исправленной, но сам факт служит хорошей иллюстрацией того, что вообще может происходить.

Сериализация может иметь место на любом уровне программного или аппаратного стека. Столкнувшись с такой проблемой, можно попытаться изменить файловую систему, обновить ядро, предъявить операционной системе больше устройств или организовать другое сочетание программной и аппаратной реализации RAID. Следует также посмотреть, что утилита *iostat* говорит о конкурентном доступе к устройству, и убедиться, что ввод/вывод действительно распараллелен (см. раздел «Как интерпретировать выдачу iostat» на стр. 422).

И не забывайте об эталонном тестировании! Оно позволит доказать, что фактическая производительность совпадает с ожидаемой. Например, если один диск может выполнять 200 операций с произвольным доступом в секунду, то том RAID 10 из 8 дисков должен выполнять примерно 1600 таких операций. Если наблюдается существенно меньшая величина, например всего 500 операций чтения, то проблему необходимо исследовать. Следите за тем, чтобы эталонные тесты нагружали подсистему ввода/вывода так же, как это делает MySQL, – например, установите флаг `O_DIRECT` и тестируйте производительность ввода/вывода на одном файле, если используется InnoDB с отключенным режимом `innodb_file_per_table`. Для таких целей отлично подходит инструмент SysBench (об эталонном тестировании см. главу 2).

Конфигурация RAID и кэширование

Обычно для конфигурирования контроллера RAID нужно войти в его программу настройки на стадии начальной загрузки компьютера. Большинство контроллеров предлагают множество разнообразных параметров, но мы остановимся только на *размере фрагмента* (chunk size) для массивов с чередованием и *кэше контроллера* (on-controller cache) (его еще называют RAID-кэшем, мы будем считать эти термины синонимами).

Размер фрагмента для слоя RAID

Оптимальный размер фрагмента для чередования зависит от рабочей нагрузки и оборудования. Теоретически для ввода/вывода с произволь-

ной выборкой больше подходит большой размер фрагмента, поскольку это означает, что один диск способен удовлетворить более длинные запросы на чтение.

Чтобы понять, почему это так, рассмотрим типичную операцию произвольного ввода/вывода для имеющейся рабочей нагрузки. Если размер фрагмента не меньше длины этой операции и данные не пересекают границу между фрагментами, то в чтении может участвовать только один диск. Но если размер фрагмента меньше длины считываемых данных, то придется задействовать несколько дисков.

Но оставим теорию. На практике многие RAID-контроллеры плохо работают с большими фрагментами. Например, контроллер может использовать фрагмент как единицу кэширования в своем кэше, а это бывает расточительно. Контроллер может также сопоставить размер фрагмента, размер кэша и размер единицы считывания (объем данных, считываемых за одну операцию). Если единица считывания оказывается слишком велика, то кэш используется неэффективно, так как в него попадает гораздо больше данных, чем необходимо, даже для крохотных запросов.

Кроме того, на практике сложно узнать, расположен ли некоторый элемент данных на одном или нескольких дисках. Даже при размере фрагмента 16 Кбайт (таким же, как у страницы InnoDB) нет уверенности в том, что все операции считывания выровнены по границе 16 Кбайт. Файловая система может фрагментировать файл и обычно выравнивает его фрагменты по границам блока файловой системы, который чаще всего составляет 4 Кбайта. Некоторые файловые системы ведут себя более разумно, но рассчитывать на это не стоит.

RAID-кэш

RAID-кэш представляет собой относительно небольшую область памяти, которая физически находится на плате RAID-контроллера. Его можно использовать для буферизации данных на пути между диском и хост-системой. Ниже перечислены некоторые причины, по которым RAID-контроллер может воспользоваться кэшем.

Кэширование результатов чтения

Прочитав какие-то данные с дисков и отправив их хост-системе, контроллер может эти данные сохранить; тогда последующие запросы тех же самых данных можно будет удовлетворить, не обращаясь снова к диску.

Обычно такое использование RAID-кэша никому не нужно. Почему? Потому что у операционной системы и СУБД есть свои, куда более обширные кэши. Если произойдет попадание в один из них, то до RAID-кэша дело даже не дойдет. И наоборот, если не было попадания ни в один из вышележащих кэшей, то шансы на то, что данные обнаружатся в RAID-кэше, исчезающе малы. Поскольку RAID-кэш

намного меньше, то почти наверняка нужные данные уже вытеснены из него и заменены новыми. В общем, как ни крути, сохранять результаты чтения в RAID-кэше – пустая трата памяти.

Упреждающее кэширование данных при чтении

Если RAID-контроллер обнаруживает запросы на чтение последовательных данных, то он может прибегнуть к упреждающему чтению, то есть заранее выбрать данные, которые, скорее всего, скоро понадобятся. Однако до тех пор, пока запрос не поступил, эти данные нужно где-то хранить. Для этой цели вполне подходит RAID-кэш. Влияние такой тактики на производительность может варьироваться в широких пределах, поэтому следует проверять, дала ли она что-нибудь в вашей ситуации. Упреждающее чтение на уровне RAID-контроллера может не иметь никакого эффекта, если СУБД реализует собственный алгоритм интеллектуального упреждающего чтения (как, например, InnoDB). К тому же оно может воспрепятствовать гораздо более важному механизму буферизации синхронных операций записи.

Кэширование операций записи

RAID-контроллер может буферизовать в своем кэше операции записи, откладывая их выполнение на более позднее время. У такой методики есть два достоинства: во-первых, контроллер может вернуть хост-системе признак успешного завершения быстрее, чем если бы физически производил запись на диски, а, во-вторых, операции записи можно объединить и выполнить более эффективно.

Внутренние операции

Некоторые операции RAID-контроллера очень сложны, особенно запись в случае RAID 5, когда нужно вычислять контрольную сумму, которая позволит реконструировать данные в случае отказа диска. Для выполнения таких операций контроллеру необходима память. В частности, по этой причине в некоторых контроллерах уровень RAID 5 работает медленно: для обеспечения высокой производительности контроллер должен прочитать в кэш много данных, но не все контроллеры умеют разумно распределять память кэша между операциями записи и операциями вычисления контрольной суммы для RAID 5.

Вообще говоря, память на плате RAID-контроллера – это дефицитный ресурс, которым нужно распорядиться с толком. Использовать его для кэширования результатов чтения – обычно откровенное расточительство, а вот использование кэша для кэширования операций записи может ощутимо повысить производительность ввода/вывода. Многие контроллеры позволяют указать, как распределить эту память. Например, можно выбрать, какую часть отвести под кэширование операций записи, а какую – для кэширования результатов чтения. В случае RAID 0, RAID 1 и RAID 10 лучше всего выделить все 100% памяти контроллера

под кэширование операций записи. В случае же RAID 5 следует зарезервировать часть памяти контроллера для внутренних операций. В общем случае эта рекомендация неплоха, но применима не всегда: разные RAID-контроллеры конфигурируются по-разному.

Если RAID-кэш применяется для кэширования операций записи, то многие контроллеры позволяют указать, на какое время можно задерживать запись (1 секунда, 5 секунд и т. д.). Чем больше задержка, тем больше операций записи удастся сгруппировать и сбросить на диск оптимальным образом. Недостаток же заключается в том, что запись оказывается «пульсирующей». В этом нет ничего страшного, если только приложение не отправит пачку запросов на запись как раз в том момент, когда кэш контроллера заполнен и должен быть сброшен на диск. Если для запросов приложения не осталось места, ему придется ждать. Если уменьшить задержку, то физических операций записи будет больше, и их группировка окажется менее эффективной, зато пики удастся сгладить, и кэш сумеет справиться с внезапным всплеском количества запросов от приложения. Это упрощенное изложение – в контроллерах часто реализуются сложные патентованные алгоритмы балансировки, но мы пытаемся объяснить основополагающие принципы.

Кэш записи очень полезен для синхронных операций, например системных вызовов `fsync()` при записи в журнал транзакций и при создании двоичного журнала в режиме `sync_binlog`, но его не следует активировать, если контроллер не оборудован блоком аварийного электропитания (BBU). В противном случае внезапное отключение напряжения может легко привести к повреждению базы данных и даже транзакционной файловой системы. Однако при наличии BBU включение кэша записи может повысить производительность в 20 и более раз, если рабочая нагрузка подразумевает большое количество сбросов в журнал, например в момент фиксации транзакций.

И в завершение следует отметить, что многие накопители на жестких дисках оборудованы собственным кэшем записи, который может «обманывать» операцию `fsync()`, сообщая контроллеру, будто данные записаны на физический носитель, хотя в действительности этого не произошло. Некоторые накопители, будучи подключены напрямую (а не через RAID-контроллер), позволяют операционной системе управлять своими кэшами, но это работает не всегда. Такие кэши обычно сбрасываются при вызове `fsync()` и обходятся при синхронном вводе/выводе, но все-таки накопитель может «лгать». Следует либо убедиться в том, что кэш действительно сбрасывается в момент вызова `fsync()`, либо отключить его вовсе, поскольку аварийное питание от батареи для него не предусмотрено. Накопители, которыми операционная система или микропрограмма RAID-контроллера не может корректно управлять, не раз становились причиной потери данных.

По этой и другим причинам мы всячески приветствуем идею о проведении настоящих «краш-тестов» (в самом буквальном смысле – выдернув

шнур из розетки). Зачастую это единственный способ найти трудноуловимые ошибки в конфигурации или воспроизвести коварное поведение накопителя. На странице *http://brad.livejournal.com/2116715.html* имеется удобный скрипт для этих целей.

Если вам нужна абсолютная уверенность в надежности BBU, которым оснащен RAID-контроллер, оставьте шнур выдернутым на достаточно длительное время. Некоторые батарейные источники поддерживают питание не так долго, как заявлено в спецификации. В этом случае, как и во многих других, одно слабое звено может привести к бесполезности всей цепочки компонентов хранения данных.

Сети хранения данных и сетевые системы хранения данных

Сети хранения данных (Storage area networks – SAN) и сетевые системы хранения данных (network-attached storage – NAS) – это два взаимосвязанных, но при этом совершенно различных способа подключения внешних устройств хранения файлов к серверу. SAN предоставляет такой интерфейс на уровне блоков, что серверу кажется, будто устройство подключено напрямую. Что же касается NAS-устройств, то они работают по протоколу файлового уровня, например NFS или SMB. SAN обычно подключается к серверу по протоколу Fibre Channel Protocol (FCP) или iSCSI, тогда как NAS-устройства – по стандартному сетевому соединению.

Сети хранения данных

К числу достоинств SAN следует отнести более гибкое управление хранением и возможность масштабирования хранилища. Многие решения на базе этой технологии предлагают такие полезные функции, как мгновенный снимок и интегрированное непрерывное резервное копирование. Они дают серверу доступ сразу к очень большому количеству дисков – часто 50 и более – и, как правило, обладают очень большим интеллектуальным кэшем для буферизации операций записи. Экспортируемый ими интерфейс блочного уровня выглядит для сервера как набор номеров логических устройств (logical unit numbers – LUNs) или виртуальных томов. Многие SAN также позволяют «кластеризовать» несколько узлов для повышения производительности.

Хотя SAN отлично работают в условиях, когда имеется много одновременных запросов и требуется высокая пропускная способность, ожидать от них чудес не стоит. На нижнем уровне SAN остается совокупностью накопителей на жестких дисках, способных выполнить лишь ограниченное число операций ввода/вывода в секунду, а, поскольку SAN является внешней по отношению к серверу и требует дополнительной обработки, то к каждому запросу ввода/вывода добавляется некоторая задержка. Эта задержка снижает эффективность работы SAN в случае,

когда требуется очень высокая производительность синхронного ввода/вывода, поэтому размещать журналы транзакций в SAN-сети обычно не следует, лучше использовать для этой цели RAID-контроллер, подключенный напрямую.

В общем случае внешняя память, подключенная напрямую, быстрее чем логические устройства в SAN при идентичном числе одинаковых накопителей. Кроме того, использование одних и тех же физических накопителей несколькими логическими устройствами (LUN) усложняет анализ производительности, поскольку LUN'ы оказывают друг на друга влияние, с трудом поддающееся измерению. Если распределить накопители по разным LUN'ам, этот эффект становится менее заметным, но иногда его все же можно наблюдать – например, при использовании протокола iSCSI иногда имеет место конкуренция за некоторый сегмент сети. У программного обеспечения SAN тоже есть свои ограничения, из-за которых реальная производительность может несколько отличаться от теоретически ожидаемой.

У SAN есть один серьезный недостаток: их стоимость гораздо выше, чем у сравнимой внешней памяти, подключенной напрямую (особенно размещаемой внутри корпуса).

Для большинства веб-приложений SAN-сети не используются, зато они весьма популярны в так называемых приложениях масштаба предприятия. Тому есть несколько причин.

- Бюджет приложений масштаба предприятия обычно не так ограничен, в то же время немногие веб-приложения могут позволить себе такую «роскошь», как SAN.
- На предприятии часто работает много приложений или много экземпляров одного и того же приложения, причем рост количественных показателей непредсказуем. SAN дает возможность закупить большой объем внешней памяти, использовать ее совместно и наращивать по мере необходимости.
- Большие буферы SAN способствуют сглаживанию пиковых нагрузок и обеспечивают быстрый доступ к «горячим» данным, при этом SAN, как правило, балансирует нагрузку между значительным числом накопителей. Все это обычно необходимо для кластерных систем, масштабируемых по вертикали[1], но не слишком полезно веб-приложениям. Для веб-приложений не характерны периоды низкой активности, за которыми следуют резкие пики записи; большинство таких программных комплексов постоянно записывают большие объемы данных, так что буферизация операций записи мало чем поможет. Ни к чему и буферизация результатов чтения, так как у СУБД есть собственные (большие и «заточенные» под конкретную задачу) кэши. Поскольку наиболее распространенной и успеш-

[1] Кластеризация – это горизонтальное масштабирование. – *Прим. науч. ред.*

ной стратегией построения очень больших веб-приложений является секционирование приложений (sharding), то нагрузка и так распределена на множество дисков.

Сетевые системы хранения данных

NAS-устройство обычно представляет собой усеченный файловый сервер, который, как правило, оснащен веб-интерфейсом, но не имеет ни мыши, ни монитора, ни клавиатуры. Это экономичный способ без особых хлопот предоставить множество дисковой памяти, причем для обеспечения избыточности обычно применяется RAID-массив.

Однако NAS-устройства не очень быстры, так как монтируются по сети. Поскольку за ними давно тянется целый шлейф проблем, связанных с поддержкой синхронного ввода/вывода и блокировки файлов, мы не рекомендуем использовать их для хранения баз данных общего назначения. Но в особых случаях, когда присущие NAS недостатки несущественны (например, для хранения таблиц типа MyISAM, предназначенных только для чтения), такие устройства использовать можно.

Использование нескольких дисковых томов

Рано или поздно возникает вопрос о том, где размещать данные. MySQL создает различные файлы:

- Файлы данных и индексов
- Журналы транзакций
- Двоичные журналы
- Журналы общего назначения (журнал ошибок, журнал запросов, журнал медленных запросов)
- Временные файлы и таблицы

В MySQL встроено не так уж много средств для хитроумного управления табличным пространством. По умолчанию все файлы, принадлежащие одной базе данных (схеме), помещаются в один каталог. Для более точного указания местоположения данных есть несколько возможностей. Так, можно задать, куда поместить индекс над таблицей типа MyISAM, а в версии MySQL 5.1 к вашим услугам секционированные таблицы.

В подразумеваемой по умолчанию конфигурации InnoDB все данные и индексы размещаются в одном наборе файлов, а в каталоге базы данных хранятся лишь файлы с определениями таблиц. Поэтому обычно все данные и индексы помещают на один том.

Однако в некоторых случаях, для того чтобы справиться с высокой нагрузкой, имеет смысл задействовать несколько томов. Например, если имеется некоторое пакетное задание, которые записывает данные в массивную таблицу, то лучше разместить ее на отдельном томе, чтобы

не отнимать у других запросов драгоценную пропускную способность подсистемы ввода/вывода. В идеале следует проанализировать доступ к разным частям данных и разместить их соответствующим образом, но сделать это проблематично, если только данные уже не находятся на разных томах.

Возможно, вам встречалась стандартная рекомендация: помещать журналы транзакций и файлы данных на разные тома, чтобы последовательная запись в журналы не мешала произвольному вводу/выводу. Однако если у вас не слишком много дисков (скажем, меньше 20), стоит хорошенько подумать, прежде чем следовать этому совету.

Реальная выгода от разделения журналов и файлов данных состоит в том, что уменьшаются шансы потерять одновременно то и другое в случае отказа. Помещение их на разные диски следует всячески приветствовать, если RAID-контроллер не оснащен кэшем записи с аварийным питанием. Но, если такой кэш есть, то выделение отдельного тома оправдано не так часто, как могло бы показаться. Производительность редко является определяющим фактором. Объясняется это тем, что хотя в журналы транзакций запись производится часто, но большинство операций очень короткие. Поэтому RAID-кэшу обычно удается объединять несколько запросов, так что в результате наблюдается всего-то два-три запроса на последовательный ввод/вывод в секунду. Этому вряд ли может помешать ввод/вывод в файлы данных с произвольным доступом. Для журналов общего назначения характерна асинхронная последовательная запись и низкая интенсивность, так что они тоже вполне могут разделять один том с данными.

На эту проблему можно взглянуть и под другим углом зрения, которым часто пренебрегают. Улучшается ли производительность в результате размещения журналов на отдельных томах? Как правило, да – но стоит ли оно того? Ответ часто отрицательный.

И вот почему: выделять специальные диски для журналов транзакций *дорого*. Предположим, что всего есть шесть дисков. Напрашивающиеся решения – объединить все шесть в один RAID-том или отдать четыре под данные, а два под журналы транзакций. Но если выбрать второй вариант, то количество дисков для хранения данных уменьшится на треть, а это немало; при этом два диска выделено под тривиальную рабочую нагрузку (в предположении, что RAID-контроллер оборудован кэшем записи с резервным питанием).

С другой стороны, если дисков много, то относительная стоимость выделения части из них под журналы транзакций уменьшается, и такое решение может стать оправданным. Например, если всего имеется 30 дисков, то, отведя под журналы 2 из них (сконфигурированных как том уровня RAID 1), можно будет обеспечить максимально быструю запись. Можно даже еще повысить производительность, зарезервировав часть RAID-кэша на запись только для этого тома.

Экономичность – не единственное соображение. Еще одна причина хранить данные и журналы транзакций InnoDB на одном томе состоит в том, что при такой стратегии вы сможете использовать функцию мгновенного снимка подсистемы управления логическими томами (LVM) для резервного копирования без блокировки. Некоторые файловые системы допускают снятие согласованных мгновенных снимков с нескольких томов, и для таких систем указанное преимущество не очень существенно, но в случае системы ext3 его следует иметь в виду.

В режиме `sync_binlog` двоичные журналы с точки зрения производительности аналогичны журналам транзакций. Однако хранить двоичные журналы отдельно от данных – это действительно *здравая* мысль, поскольку в этом случае они могут уцелеть даже после потери данных и, следовательно, ничто не помешает использовать их для восстановления на определенный момент в прошлом. Это соображение не применимо к журналам транзакций InnoDB, так как последние без файлов данных бесполезны, – невозможно применить журналы транзакций к резервной копии, снятой прошлой ночью. Это различие между журналами транзакций и двоичными журналами может показаться искусственным администратору, привыкшему к другой СУБД, где они представляют собой одно и то же.

И есть еще один распространенный случай, когда имеет смысл выделять для некоторых файлов отдельное место. Речь идет о специальном каталоге, который MySQL использует для сортировок (filesorts) и создания временных таблиц на диске. Если генерируемые при этом файлы не слишком велики, то, быть может, имеет смысл размещать их во временной файловой системе в памяти, например такой как `tmpfs`. Это самый быстрый вариант. Если для вас он не подходит, то создавайте временный каталог на том же устройстве, где находится операционная система.

Типичное распределение дискового пространства таково: операционная система, раздел свопинга и двоичные журналы – на томе уровня RAID 1, а для всего остального – том уровня RAID 5 или RAID 10.

Конфигурация сети

Точно так же, как производительность жесткого диска ограничена временем задержки и пропускной способностью, для сетевого соединения лимитирующими факторами являются задержка и полоса пропускания (это просто другое название пропускной способности). Для большинства приложений основную проблему составляет время задержки; типичное приложение выполняет много коротких операций передачи по сети, поэтому незначительные задержки для каждой транзакции суммируются.

Серьезным узким местом может стать плохо работающая сеть. Даже одного процента потерянных пакетов достаточно для существенного снижения производительности, так как различные уровни стека прото-

колов будут пытаться исправить ошибку, для чего ненадолго перейдут в режим ожидания, а затем начнут отправлять пакет повторно, – на все это уходит лишнее время. Еще одна типичная проблема – неправильно сконфигурированный или медленно работающий DNS-сервер.

Разрешение доменных имен – это такая Ахиллесова пята, что ее одной хватает для включения режима `skip_name_resolve` на промышленных серверах. Неработающий или медленно работающий DNS-сервер – беда для многих приложений, но для MySQL особенно. Получая запрос на соединение, MySQL выполняет прямой и обратный поиск в DNS. Он может завершиться неудачно по самым разным причинам, и в таком случае в установлении соединения будет вообще отказано или данная процедура займет недопустимо много времени, что в общем случае может привести к полному хаосу, в том числе послужить способом DoS-атаки. Если включить параметр `skip_name_resolve`, то MySQL не будет посылать никаких запросов DNS-серверу. Но это означает, что во всех учетных записях пользователей столбец `host` может содержать только IP-адрес (возможно, с метасимволами) или строку «localhost». Пользователь, для которого в учетной записи указано доменное имя, не сможет соединиться с сервером.

Необходимо специально настраивать сеть для достижения хорошей производительности, а не соглашаться с параметрами по умолчанию. Для начала проанализируйте количество переходов между узлами и начертите карту физического расположения сети. Пусть, например, имеется 10 веб-серверов, подключенных к коммутатору «Web» посредством гигабитной сети Ethernet (1 GigE), и этот коммутатор соединен с коммутатором «Database» такой же гигабитной сетью. Не потратив время на трассировку соединений, вы так никогда и не узнаете, что полная пропускная способность сети, соединяющей веб-серверы с серверами баз данных, ограничена одним гигабитом! Кроме того, каждый переход добавляет задержку.

Было бы очень неплохо вести мониторинг производительности и ошибок сети на всех портах. Это относится к портам на серверах, маршрутизаторах и коммутаторах. Программа Multi Router Traffic Grapher, или MRTG (*http://oss.oetiker.ch/mrtg/*), – проверенное на практике решение по мониторингу устройств. Упомянем также две утилиты для мониторинга производительности сети (а не самих устройств): Smokeping (*http://oss.oetiker.ch/smokeping/*) и Cacti (*http://www.cacti.net*).

В организации сетей большое значение имеет физическая удаленность. В междугородных сетях задержка намного больше, чем в локальной сети центра обработки данных, пусть даже технически пропускная способность одинакова. Если узлы находятся очень далеко друг от друга, то приходится учитывать и скорость распространения света. Например, центры обработки данных на западном и восточном побережье США отстоят примерно на 4800 км. Поскольку скорость света составляет 300 000 км/с, то прохождение пакета в одну сторону займет никак не

меньше 16 мс, а в обе стороны – по крайней мере 32 мс. Но физическое расстояние – это еще не все: на маршруте пакета существуют и другие устройства. Повторители, маршрутизаторы, коммутаторы – все они в какой-то мере снижают производительность. К тому же чем больше расстояние между узлами сети, тем менее предсказуемо и надежно поведение соединяющих их каналов связи.

Мы рекомендуем всеми силами избегать операций, требующих передачи данных между центрами обработки в реальном масштабе времени[1]. Если это невозможно, проектируйте приложение так, чтобы оно адекватно реагировало на сетевые ошибки. Например, следует ограничить количество процессов, порождаемых веб-сервером Apache, поскольку они могут долго ждать соединения с удаленным центром обработки данных по каналу связи с большим процентом потерь пакетов.

В локальных сетях применяйте хотя бы технологию 1 GigE. Для прокладки магистрального канала между коммутаторами может потребоваться и сеть типа 10 GigE. Если нужна еще большая пропускная способность, то можно воспользоваться транкингом: соединить несколько сетевых карт для повышения величины канала. Транкинг – это, по сути дела, распараллеливание потоков данных в сети, он также может оказаться очень полезным как часть стратегии обеспечения высокой доступности.

Если необходима чрезвычайно высокая пропускная способность, то, возможно, удастся улучшить производительность путем настройки сетевых параметров операционной системы. Если соединений немного, но зато запросы или результирующие наборы велики, то попробуйте увеличить размер буфера TCP. В разных системах это делается по-разному, но в большинстве дистрибутивов GNU/Linux следует изменить значения параметров в файле */etc/sysctl.conf* и выполнить команду *sysctl -p*, либо воспользоваться файловой системой */proc*, записав новые значения в файлы, находящиеся в каталоге */proc/sys/net/* с помощью команды *echo*. Хорошие пособия по этой теме можно найти в Сети по запросу «TCP tuning guide» («Тонкая настройка TCP»).

Но чаще возникает необходимость в эффективной работе с большим количеством соединений и маленькими запросами. Очень распространена настройка диапазона локальных портов. Вот как система конфигурируется по умолчанию:

```
[root@caw2 ~]# cat /proc/sys/net/ipv4/ip_local_port_range
32768 61000
```

Иногда диапазон портов нужно расширить, например:

---

[1] Репликация не считается передачей данных в реальном времени. Поэтому вполне здравой выглядит мысль реплицировать данные в удаленный центр обработки ради повышения безопасности. Эту тему мы будем рассматривать в следующей главе.

```
[root@caw2 ~]# echo 1024 65535 > /proc/sys/net/ipv4/ip_local_port_range
```

Можно также увеличить размер очереди соединений:

```
[root@caw2 ~]# echo 4096 > /proc/sys/net/ipv4/tcp_max_syn_backlog
```

Если сервер базы данных используется только локально, то можно уменьшить величину тайм-аута после закрытия сокета, в течение которого система не закрывает свою сторону соединения, страхуясь от аварийного завершения клиента. По умолчанию в большинстве систем эта величина составляет одну минуту, что довольно долго.

```
[root@caw2 ~]# echo <value> > /proc/sys/net/ipv4/tcp_fin_timeout
```

Как правило, для этих параметров можно оставить значения по умолчанию. Изменять их имеет смысл только тогда, когда происходит что-то экстраординарное, например производительность сети необычайно низка или количество соединений очень велико. Поиск в Интернете по запросу «TCP variables» («параметры TCP») даст обширнейший материал по этим и многим другим переменным.

Выбор операционной системы

В настоящее время для высокопроизводительных приложений MySQL чаще всего устанавливается на ОС GNU/Linux, хотя может работать и во многих других системах.

На компьютерах SPARC обычно работает ОС Solaris, она нередко служит основой для приложений, требующих высокой надежности. Сложилось мнение, что с системой Solaris работать в некоторых отношениях сложнее, чем с GNU/Linux, но тем не менее это надежная, высокопроизводительная ОС с целым рядом передовых возможностей. В частности, набирает популярность система Solaris 10. К числу ее особенностей можно отнести наличие собственной файловой системы (ZFS), разнообразие развитых инструментов для поиска неполадок (например, DTrace), высокопроизводительную многопоточность и технологию виртуализации Solaris Zones, упрощающая управление ресурсами. Компания Sun также предлагает отличную поддержку для MySQL.

Еще одна возможность – ОС FreeBSD. В прошлом MySQL испытывал с ней много проблем, в основном из-за поддержки потоков, но современные версии существенно улучшены. Сегодня не редкость встретить крупномасштабный проект с использованием MySQL, развернутый на платформе FreeBSD.

ОС Windows обычно используется для разработки или когда MySQL является частью персонального приложения. Существуют проекты масштаба предприятия на платформе Windows с применением MySQL, но чаще для этих целей все же используется UNIX. Не желая ввязываться в споры по поводу операционных систем, отметим, что MySQL прекрасно работает и в гетерогенной среде. Абсолютно ничто не мешает за-

пустить сервер MySQL в системе UNIX, а веб-серверы – на платформе Windows, используя для подключения к базе высококачественный коннектор ADO.NET (предлагаемый бесплатно вместе с MySQL). Установить соединение между клиентом в UNIX и MySQL-сервером в Windows столь же просто, как с сервером на другой UNIX-машине.

Если вы работаете на 64-разрядной аппаратной архитектуре, то выбирайте 64-разрядную операционную систему. Хотя этот совет звучит нелепо, нам неоднократно приходилось встречать 32-разрядные ОС, по ошибке установленные на машину с 64-разрядным процессором. Процессор-то будет исполнять код без возражений, но стандартные ограничения, присущие 32-разрядной системе (такие как меньший объем адресуемой памяти), не дадут использовать все возможности.

Выбор конкретного дистрибутива GNU/Linux – дело вкуса. Мы полагаем, что лучше всего остановиться на дистрибутиве, специально разработанном для серверных, а не персональных приложений. Принимайте во внимание время существования дистрибутива, политику выпуска версий и обновлений, проверьте, оказывает ли производитель техническую поддержку. Репутацию качественного и стабильного продукта заслужил дистрибутив Red Hat Enterprise Linux; популярен совместимый с ним на двоичном уровне (и бесплатный) дистрибутив CentOS; набирает очки также Ubuntu.

Выбор файловой системы

Выбор файловой системы, конечно, зависит от выбора ОС. Во многих системах, например в Windows, есть всего один или два варианта. С другой стороны, GNU/Linux поддерживает много разных файловых систем.

Часто задают вопрос, какая файловая система обеспечивает наилучшую производительность для комбинации MySQL с GNU/Linux, или даже более конкретно: что лучше для InnoDB, а что – для MyISAM? Эталонные тесты показывают, что во многих отношениях большинство файловых систем очень близки, но полагаться на файловую систему в деле повышения производительности – пустое занятие. Производительность файловой системы сильно зависит от рабочей нагрузки, и ни одна из них не является панацеей. Как правило, каждая конкретная файловая система работает не хуже и не лучше любой другой. Исключение составляет случай, когда вы приближаетесь к какому-то лимиту файловой системы, например возникает высокая конкурентность, образуется много файлов, нарастает фрагментация и т. д.

Более важным фактором является время восстановления после сбоя и возможность наткнуться на конкретные ограничения, например низкая производительность при работе с каталогами, которые содержат много файлов (этим отличаются системы ext2 и ext3, хотя современные версии ext3 в этом плане стали получше). Выбор файловой системы имеет первостепенное значение для гарантии надежного хранения дан-

ных, поэтому мы настоятельно рекомендуем не экспериментировать на промышленных серверах.

По возможности предпочитайте файловые системы с журналированием, например ext3, ReiserFS, XFS, ZFS или JFS. В противном случае проверка файловой системы после сбоя может занять много времени. Если этот аспект не очень важен, то файловые системы без журналирования могут работать несколько быстрее транзакционных. Например, ext2 в этом смысле лучше, чем ext3, хотя при желании можно отключить журналирование в ext3 командой *tunefs*. Для некоторых файловых систем имеет также значение время монтирования. Так, на многотерабайтных разделах система ReiserFS монтируется и восстанавливается довольно долго.

Для файловой системы ext3 существует три варианта журналирования данных, соответствующие параметры монтирования задаются в файле */etc/fstab*:

`data=writeback`

Журналируются только изменения метаданных. Изменение метаданных не синхронизированы с записью информации. Это самая быстрая конфигурация и *обычно* она безопасна при работе с InnoDB, так как последняя ведет собственный журнал транзакций. Есть, правда, одно исключение: сбой в неподходящий момент может привести к повреждению *frm*-файла.

Посмотрим, при каких обстоятельствах такая конфигурация может привести к беде. Предположим, что программа решила расширить файл, увеличив его размер. Метаданные (размер файла) записываются в журнал до записи самих данных в файл (который теперь стал больше). В результате в конце файла – добавленной области – окажется мусор.

`data=ordered`

В этом режиме тоже журналируются только метаданные, но обеспечивается хоть какая-то согласованность за счет того, что данные пишутся раньше метаданных. Производительность лишь немного уступает режиму `writeback`, зато система ведет себя гораздо надежнее при сбоях.

Если в этом режиме программа захочет расширить файл, то в его метаданных не будет отражен новый размер до тех пор, пока во вновь выделенную область не будут записаны данные.

`data=journal`

В этом режиме журнал ведется атомарно, то есть данные пишутся сначала в журнал, а только потом туда, где им надлежит быть. Обычно это излишне, а накладные расходы гораздо выше, чем в двух других режимах. Впрочем, в ряде случаев производительность даже увеличивается, так как наличие журнала позволяет файловой системе отложить записи данных в сам файл.

Вне зависимости от файловой системы существует ряд параметров, которые лучше отключить, поскольку они не дают никаких преимуществ, но сопряжены с издержками. В этой связи чаще всего упоминают регистрацию времени последнего доступа, потому что она подразумевает операцию записи даже в том случае, когда вы просто читаете файл. Чтобы отключить этот режим, добавьте в файл */etc/fstab* флаг монтирования `noatime`; иногда таким образом удается получить выигрыш в 5–10% в зависимости от рабочей нагрузки и файловой системы (хотя в других случаях никаких ощутимых изменений не наблюдается).

Приведем пример строки файла */etc/fstab* с вышеупомянутым флагом для файловой системы ext3:

```
/dev/sda2 /usr/lib/mysql ext3 noatime,data=writeback 0 1
```

Можно также отключить в файловой системе упреждающее чтение, поскольку иногда оно оказывается избыточным. Например, InnoDB самостоятельно прогнозирует, к каким страницам может быть доступ в ближайшем будущем. Отключение или ограничение упреждающего чтения особенно благотворно сказывается на файловой системе UFS в ОС Solaris. Использование флага `O_DIRECT` автоматически отключает упреждающее чтение.

Некоторые файловые системы могут не поддерживать необходимых функций. Например, при использовании метода сброса `O_DIRECT` для InnoDB (см. раздел «Как InnoDB открывает и сбрасывает файлы журнала и данных» на стр. 359) важна поддержка прямого ввода/вывода. Кроме того, одни файловые системы лучше работают с большим количеством накопителей, чем другие; так XFS часто в этом отношении гораздо лучше ext3. Наконец, если вы планируете применять мгновенные снимки менеджера логических томов (LVM) для инициализации подчиненных серверов или снятия резервных копий, убедитесь, что выбранная файловая система хорошо «ладит» с LVM.

В табл. 7.2 приведены характеристики некоторых наиболее часто используемых файловых систем.

Таблица 7.2. Характеристики наиболее часто используемых файловых систем

Файловая система	**Операционная система**	**Журналирование**	**Большие каталоги**
ext2	GNU/Linux	Нет	Нет
ext3	GNU/Linux	По желанию	По желанию / частично
HFS Plus	Mac OS	По желанию	Есть
JFS	GNU/Linux	Есть	Нет
NTFS	Windows	Есть	Есть

Файловая система	Операционная система	Журналирование	Большие каталоги
ReiserFS	GNU/Linux	Есть	Есть
UFS (Solaris)	Solaris	Есть	Настраивается
UFS (FreeBSD)	FreeBSD	Нет	По желанию / частично
UFS2	FreeBSD	Нет	По желанию / частично
XFS	GNU/Linux	Есть	Есть
ZFS	Solaris, FreeBSD	Есть	Есть

Многопоточность

Начиная с версии 5.0 в MySQL выделяется по одному потоку на каждое соединение. Дополнительно существуют служебные потоки, потоки специального назначения и потоки, создаваемые подсистемами хранения. Поэтому необходимо, чтобы операционная система эффективно поддерживала много потоков. Более того, для результативной работы с несколькими процессорами MySQL нуждается в потоках на уровне ядра, а не в адресном пространстве пользователя. Вдобавок требуется наличие эффективных примитивов синхронизации, например мьютексов. Все это должны предоставлять библиотеки, входящие в состав операционной системы.

В ОС GNU/Linux есть две библиотеки для работы с потоками: LinuxThreads и более новая Native POSIX Threads Library (NPTL). Библиотека LinuxThreads еще кое-где используется, но большинство современных дистрибутивов перешли на NPTL, а во многие дистрибутивы LinuxThreads уже и не входит. Библиотека NPTL обычно потребляет меньше памяти, работает более эффективно и не страдает от многих проблем, присущих LinuxThreads. В ней имеется несколько вопросов, связанных с производительностью, но по большей части дефекты уже исправлены.

ОС FreeBSD также поставляется с несколькими потоковыми библиотеками. Исторически поддержка потоков в этой системе была слабой, но сейчас она заметно улучшилась и на некоторых тестах даже превосходит по производительности GNU/Linux в SMP-системах (с симметричной многопроцессорностью). В версии FreeBSD 6 и более поздних мы рекомендуем библиотеку *libthr*, а в более ранних – библиотеку *linuxthreads*, которая является переносом LinuxThreads на платформу FreeBSD.

В ОС Solaris поддержка потоков выше всяких похвал.

Свопинг

Свопинг (подкачка) имеет место, когда операционная система выгружает какую-то область виртуальной памяти на диск, поскольку она не помещается в физической памяти[1]. Свопинг незаметен работающим процессам. Только ОС знает, находится ли некий адрес виртуальной памяти в физической памяти или на диске.

Свопинг очень плохо отражается на производительности MySQL. Он сводит на нет весь смысл кэширования, и эффективность оказывается *ниже*, чем в случае, когда для кэшей отведено слишком мало памяти. В сервере MySQL и подсистемах хранения есть немало алгоритмов, которые поразному работают с данными, находящимися в памяти и на диске, поскольку предполагается, что доступ к хранящимся в ОЗУ данным обходится дешево. Поскольку свопинг не виден пользовательским процессам, ни MySQL, ни подсистема хранения не знают, что данные, которые они считают находящимися в памяти, на самом деле выгружены на диск.

Это может очень сильно снизить производительность. Например, полагая, что данные все еще в ОЗУ, подсистема хранения может захватить глобальный мьютекс (например, мьютекс, защищающий пул буферов в InnoDB) на время «короткой» операции с памятью. Но если эта операция выливается в дисковый ввод/вывод, то все остальное замирает в ожидании его завершения. Следовательно, свопинг приводит к гораздо более тяжким последствиям, чем обычный ввод/вывод, выполняемый по мере необходимости.

В ОС GNU/Linux за свопингом можно следить с помощью утилиты *vmstat* (в следующем разделе мы приведем несколько примеров). Интерес представляют столбцы `si` и `so`, отражающие динамику свопинга, а не столбец `swpd`, в котором показан объем использованного пространства в файле подкачки. Величина в столбце `swpd` может включать информацию о процессах, которые были загружены в память, но сейчас не работают, а, стало быть, и проблем не создают. Желательно, чтобы в столбцах `si` и `so` стояли нули, и уж, во всяком случае, эти показатели не должны превышать 10 блоков в секунду.

В редких случаях слишком активный свопинг может привести к исчерпанию места в файле подкачки. Когда такое происходит, нехватка виртуальной памяти обычно кончается аварийным завершением MySQL. Но даже если в файле подкачки есть место, чрезмерно интенсивный свопинг может замедлить работу ОС до такой степени, что невозможно будет даже войти в систему и принудительно завершить процесс MySQL.

Многие проблемы, связанные со свопингом, можно решить путем правильного конфигурирования буферов MySQL, но иногда операционная система все-таки решает выгрузить соответствующий процесс. Обычно

---

[1] Иногда свопинг называют страничной подкачкой (paging). Строго говоря, это разные вещи, но многие употребляют их как *синонимы*.

это происходит, когда ОС видит, что MySQL выдает слишком много запросов на ввод/вывод, и пытается увеличить файловый кэш, чтобы он вмещал больше данных. Если памяти недостаточно, что-то приходится выгружать на диск, и этим «чем-то» вполне может оказаться сам MySQL. В некоторых старых версиях ядра Linux к тому же установлены контрпродуктивные приоритеты, из-за которых выгружается то, что выгружать бы не надо, но впоследствии эта ошибка была исправлена.

Некоторые считают, что файл подкачки вообще следует отключить. В некоторых крайних случаях, когда иначе ядро просто отказывается вести себя «порядочно», это помогает, но вообще-то может привести и к снижению производительности ОС (теоретически не должно, но на практике встречается). Кроме того, это попросту опасно, так как отключение свопинга означает установку жесткого ограничения на объем виртуальной памяти. Если MySQL испытывает кратковременную потребность в большом количестве памяти или на той же машине время от времени запускаются процессы, потребляющие много ресурсов (скажем, ночные пакетные задания), то память у MySQL может кончиться, что приведет к аварийному завершению или снятию процесса операционной системой.

Обычно операционные системы предоставляют какие-то средства контроля над виртуальной памятью и подсистемой ввода/вывода. Упомянем лишь некоторые инструменты, присутствующие в GNU/Linux. Самый простой способ – уменьшить значение параметра */proc/sys/vm/swappiness* до 0 или 1. Тем самым вы говорите ядру, что оно не должно прибегать к свопингу до тех пор, пока потребность в виртуальной памяти не станет критической. Ниже показано, как узнать текущее значение параметра и изменить его.

```
$ cat /proc/sys/vm/swappiness
60
$ echo 0 > /proc/sys/vm/swappiness
```

Другой способ заключается в том, чтобы изменить порядок чтения и записи данных подсистемой хранения. Например, установка режима `innodb_flush_method=O_DIRECT` снижает давление на подсистему ввода/вывода. Прямой ввод/вывод не кэшируется, поэтому ОС не считает его причиной для увеличения кэша. Но этот параметр применим только к InnoDB, хотя и в подсистеме Falcon реализована поддержка прямого ввода/вывода. Можно также использовать страницы большого размера, которые не выгружаются в своп. Это работает для подсистем MyISAM и InnoDB.

Еще один способ – воспользоваться конфигурационным параметром MySQL `memlock`, который фиксирует MySQL в оперативной памяти. Такой процесс выгружаться не будет, но возникает другая опасность: если невыгружаемая память кончится, то MySQL может завершиться аварийно при попытке получить дополнительную память. Проблема может возникнуть и тогда, когда невыгружаемой памяти выделяется слишком много, так что ничего не остается для самой операционной системы.

Есть немало трюков, зависящих от конкретной версии ядра, поэтому будьте осторожны, особенно при переходе на новую версию. При некоторых рабочих нагрузках бывает сложно заставить операционную систему вести себя разумно, и тогда единственный выход – уменьшить размеры буферов до субоптимальных значений.

Состояние операционной системы

В вашей операционной системе, скорее всего, имеются инструменты, позволяющие выяснить, чем заняты сама система и оборудование. Мы приведем примеры использования двух широко распространенных утилит: *iostat* и *vmstat*. Если в вашей системе нет какой-нибудь из них[1], то почти наверняка существует нечто аналогичное. Мы ставим себе целью не превратить вас в эксперта по *iostat* или *vmstat*, а просто показать, на что нужно обращать внимание, пытаясь найти проблемы.

Помимо этих инструментов в вашей ОС могут быть и другие, к примеру, *mpstat* или *sar*. Если вас интересуют иные части системы, скажем, сеть, то имеет смысл воспользоваться такими утилитами, как *ifconfig* (показывает, в частности, количество сетевых ошибок) или *netstat*.

По умолчанию *vmstat* и *iostat* выдают лишь отчет о средних значениях различных счетчиков с момента запуска сервера – это не слишком интересная информация. Но обе программы принимают в качестве аргумента интервал времени. В этом случае генерируются инкрементные отчеты, показывающие, что сервер делает в настоящий момент, а это уже куда полезнее для настройки (в первой строке показана статистика с момент запуска системы, на нее можно не обращать внимания).

Как интерпретировать выдачу vmstat

Сначала рассмотрим пример работы *vmstat*. Следующая команда будет выводить отчет каждые пять секунд:

```
$ vmstat 5
procs ---------memory---------- --swap-- ----io---- -system- ----cpu----
  r b   swpd  free  buff  cache   si  so    bi    bo   in cs  us sy id wa
  0 0   2632 25728 23176 740244    0   0   527   521   11  3  10  1 86  3
  0 0   2632 27808 23180 738248    0   0     2   430  222 66   2  0 97  0
```

[1] Мы остановились на утилитах *vmstat* и *iostat*, потому что они широко доступны и по умолчанию включаются в состав многих UNIX-подобных систем. Но у обеих имеются ограничения, например путаница в единицах измерения, опрос с интервалами, не соответствующими моментам обновления статистики операционной системой, и невозможность вывести все показатели сразу. Если эти инструменты не отвечают вашим потребностям, посмотрите, не подойдут ли программы *dstat* (*http://dag.wieers.com/home-made/dstat/*) или *collectl* (*http://collectl.sourceforge.net/*).

Чтобы остановить *vmstat*, нажмите Ctrl-C. Вид отчета зависит от операционной системы, поэтому для получения точной информации обратитесь к странице руководства. Как упоминалось выше, несмотря на то, что мы запрашивали инкрементный отчет, в первой строке показаны средние значения за время, прошедшее с момента запуска сервера. Значения во второй строке отражают текущие показатели, а последующие строки печатаются с пятисекундным интервалом. Столбцы объединены в следующие группы.

`procs`

В столбце `r` показано, сколько процессов ожидает выделения процессора, а в столбце `b` – сколько процессов находятся в состоянии непрерываемого «сна»; обычно это означает, что процесс ожидает завершения ввода/вывода (дискового, сетевого, ввода информации пользователем и т. д.).

`memory`

В столбце `swpd` показано, сколько блоков выгружено на диск в результате страничного обмена. Оставшиеся три столбца – это количество свободных (неиспользуемых), отведенных под буферы и выделенных для кэша операционной системы блоков.

`swap`

В столбцах из этой группы показана активность подкачки: сколько блоков в секунду подгружается с диска и выгружается на диск. Эта информация важнее, чем значение в столбце `swpd`.

Желательно, чтобы значения в столбцах `si` и `so` оставались равными 0 и уж точно они не должны превышать 10 блоков в секунду. Кратковременные всплески активности тоже не сулят ничего хорошего.

`io`

Значения в этих столбцах показывают, сколько блоков в секунду считывается с (`bi`) и записывается на (`bo`) блочные устройства. Обычно речь идет о дисковом вводе/выводе.

`system`

Здесь показаны количество прерывания в секунду (`in`) и количество контекстных переключений в секунду (`cs`).

`cpu`

Значения в этих столбцах показывают, какую часть времени (в процентах) процессор выполняет пользовательский код (вне ядра), системный код (в ядре), простаивает и ожидает завершения ввода/вывода. Если используется виртуализация, то может присутствовать и пятый столбец (`st`), показывающий, сколько времени «украдено» у виртуальной машины. Речь идет о том времени, в течение которого на виртуальной машине существовал процесс, готовый к исполне-

нию, но гипервизор решил использовать процессор для других целей. Время, когда у виртуальной машины не было ничего готового к исполнению, и гипервизор отобрал у нее процессор, не считается украденным.

Формат отчета *vmstat* системно-зависимый, поэтому если вы наблюдаете картину, отличную от описанной выше, почитайте страницу руководства vmstat(8). Важное замечание: величины в разделах, относящихся к памяти, свопингу и вводу/выводу, измеряются в блоках, а не в байтах. В ОС Linux блок обычно составляет 1024 байта.

Как интерпретировать выдачу iostat

Теперь перейдем к утилите *iostat*[1]. По умолчанию она демонстрирует некоторые показатели работы ЦП – те же, что *vmstat*. Но обычно нас интересует лишь статистика ввода/вывода, поэтому мы запускаем команду с флагами, при которых выводится лишь расширенная статистика устройств:

```
$ iostat -dx 5
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda         1.6    2.8 2.5 1.8  138.8   36.9     40.7      0.1  23.2   6.0   2.6
```

Как и в случае с *vmstat*, в первой строке показаны средние значения за период с момента запуска сервера (далее для экономии места мы будем ее опускать), а в последующих инкрементные средние. В каждой строке выводятся данные об одном устройстве.

Существуют различные флаги, позволяющие показать или скрыть отдельные столбцы. Мы вывели следующие из них.

`rrqm/s` ***и*** `wrqm/s`

Количество сгруппированных (merged) запросов на чтение и запись в секунду. «Сгруппированный» означает, что операционная система объединила несколько логических запросов в один физический запрос к устройству.

`r/s` ***и*** `w/s`

Количество запросов на чтение и запись, отправленных устройству в секунду.

`rsec/s` ***и*** `wsec/s`

Количество прочитанных и записанных секторов в секунду. В некоторых системах выводятся также столбцы rkB/s и wkB/s – количество прочитанных и записанных килобайтов в секунду. Мы их для краткости опустили.

---

[1] Примеры, относящиеся к *iostat*, немного переформатированы для удобства печати. Чтобы избежать переноса строк, мы уменьшили количество знаков после запятой.

`avgrq-sz`

Размер запроса в секторах.

`avgqu-sz`

Количество запросов, стоящих в очереди к устройству.

`await`

Количество миллисекунд, потребовавшихся для получения ответа на запрос, включая время ожидания в очереди и время обслуживания. К сожалению, *iostat* не показывает статистику времени обслуживания отдельно для запросов на чтение и на запись, хотя они настолько отличаются, что не должны усредняться вместе. Впрочем, большое время ожидания, скорее всего, можно отнести на счет чтения, поскольку операции записи часто можно буферизовать, тогда как чтение обычно должно производиться непосредственно с физического накопителя.

`svctm`

Количество миллисекунд, затраченных на обслуживание запросов от начала до конца, включая время ожидания в очереди и время, потребовавшееся устройству для выполнения запроса.

`%util`

Процентная доля времени ЦП, в течение которого устройству отправлялись запросы. Этот показатель характеризует загруженность устройства, поскольку значение 100% означает, что устройство загружено полностью.

Сгенерированный отчет позволяет сделать некоторые заключения о работе подсистемы ввода/вывода. К числу наиболее важных показателей относится количество одновременно обслуживаемых запросов. Поскольку количество операций чтения/записи приведено в секунду, а время обслуживания измеряется в тысячных долях секунды, то количество запросов, одновременно обслуживаемых устройством, вычисляется по формуле[1]:

```
concurrency = (r/s + w/s) * (svctm/1000)
```

Ниже приведен пример выдачи *iostat*:

```
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda         105    311 298 820   3236   9052       10      127   113     9    96
```

[1] По-другому охарактеризовать уровень конкурентности можно с помощью средней длины очереди, времени обслуживания и среднего времени ожидания: `(avuqu_sz * svctm) / await`.

Подставив эти значения в предыдущую формулу, получим, что коэффициент конкурентности равен примерно 9,6[1]. Это означает, что в среднем за время между двумя опросами устройства одновременно обрабатывалось 9,6 запросов. Данные получены для массива RAID 10 из 10 дисков, так что система вполне эффективно распараллеливает запросы к этому устройству. С другой стороны, вот пример устройства, которое, похоже, выполняет запросы строго последовательно:

```
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sdc          81       0 280   0   3164      0       11        2     7     3    99
```

Расчет по приведенной выше формуле показывает, что устройство обслуживает всего один запрос в секунду. Оба устройства близки к насыщению, но производительность их различна. Если вы видите, что устройство почти все время занято, как в этих примерах, то посчитайте коэффициент конкурентности. Он должен быть близок к количеству физических накопителей. Если значение заметно меньше, значит, есть какие-то проблемы.

Машина с нагруженным процессором

Если процессор используется «на полную катушку», то *vmstat* обычно показывает большое значение в столбце `us` – доля времени, в течение которого процессор занят выполнением пользовательского кода. В большинстве случаев вы увидите также, что в очереди к ЦП стоят несколько процессов (столбец `r`). Например:

```
$ vmstat 5
procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
 r b   swpd  free  buff     cache  si   so     bi    bo   in    cs us sy id wa
10 2 740880 19256 46068  13719952   0    0   2788 11047 1423 14508 89  4  4  3
11 0 740880 19692 46144  13702944   0    0   2907 14073 1504 23045 90  5  2  3
 7 1 740880 20460 46264  13683852   0    0   3554 15567 1513 24182 88  5  3  3
10 2 740880 22292 46324  13670396   0    0   2640 16351 1520 17436 88  4  4  3
```

Обратите также внимание на большое количество контекстных переключений (столбец `cs`). Контекст переключается, когда операционная система приостанавливает один процесс и замещает его другим.

Запустив на той же машине *iostat* (верхняя строка со средними за все время с момента запуска опущена), мы увидим, что коэффициент загруженности диска менее 50%:

```
$ iostat -dx 5
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda           0   3859  54 458   2063  34546       71        3     6     1    47
dm-0          0      0  54 4316  2063  34532        8       18     4     0    47
```

[1] Проделав вычисления самостоятельно, вы получите примерно 10, поскольку мы округлили значения, которые в действительности выводит *iostat*. Поверьте на слово, что на самом деле результат равен 9,6.

```
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda          0   2898  52 363   1767  26090       67         3     7     1    45
dm-0         0      0  52 3261  1767  26090        8        15     5     0    45
```

Хотя основное время эта машина тратит на процессорные операции, объем ввода/вывода также велик, что характерно для серверов баз данных. С другой стороны, типичный веб-сервер потребляет очень много ресурсов ЦП, но очень слабо загружает подсистему ввода/вывода, поэтому для веб-сервера картина будет отличаться от показанной выше.

Машина с нагруженной подсистемой ввода/вывода

Если рабочая нагрузка сопряжена преимущественно с вводом/выводом, то ЦП проводит много времени в ожидании завершения запросов ввода/вывода. Это означает, что *vmstat* покажет много процессов в состоянии непрерываемого сна (столбец `b`), и значение в столбце `wa` тоже будет велико. Например:

```
$ vmstat 5
procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
r b     swpd  free  buff    cache   si   so    bi    bo    in   cs  us sy id wa
5 7   740632 22684 43212 13466436    0    0  6738 17222  1738 16648 19  3 15 63
5 7   740632 22748 43396 13465436    0    0  6150 17025  1731 16713 18  4 21 58
1 8   740632 22380 43416 13464192    0    0  4582 21820  1693 15211 16  4 24 56
5 6   740632 22116 43512 13463484    0    0  5955 21158  1732 16187 17  4 23 56
```

iostat демонстрирует, что на этой машине диски полностью загружены:

```
$ iostat -dx 5
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda          0   5396 202 626   7319  48187       66        12    14     1   101
dm-0         0      0 202 6016  7319  48130        8        57     9     0   101
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda          0   5810 184 665   6441  51825       68        11    13     1   102
dm-0         0      0 183 6477  6441  51817        8        54     7     0   102
```

Величина `%util` может быть больше 100% из-за ошибок округления.

Когда можно считать, что подсистема ввода/вывода сильно нагружена? Если емкости буферов хватает для обслуживания запросов на запись, то обычно (но не всегда) это означает, что диски не справляются с запросами на *чтение*, даже если выполняется очень много операций записи. Это кажется противоречащим интуиции, но давайте задумаемся о природе чтения и записи.

- Запросы на запись могут быть либо буферизованными, либо синхронными. Как мы уже отмечали, буферизация возможна на различных уровнях: операционная система, RAID-контроллер и т. д.
- Запросы на чтение по природе своей синхронны. Программа, конечно, может предположить, что в ближайшем будущем потребуются некоторые данные, и отправить асинхронный запрос на упреждающее чтение. Однако чаще программа обнаруживает, что некоторые

данные необходимы, и без них не может продолжить работу. Поэтому чтение обязано быть синхронным: процесс блокируется до завершения запроса.

Взгляните на ситуацию следующим образом: вы можете отправить запрос на запись, который сохраняется в каком-то буфере и выполняется позже. Можно даже отправить много таких запросов в течение одной секунды. Если буфер работает корректно и в нем достаточно места, то каждый запрос завершается очень быстро, а реальные операции записи на физический диск впоследствии можно будет сгруппировать и выполнить в другом порядке для повышения эффективности.

Но с чтением так поступить нельзя – неважно, короткий запрос или длинный, диск не может ответить: «Вот твои данные, а прочту я их попозже». Поэтому ожидание ввода/вывода вызвано прежде всего операциями чтения.

Машина с интенсивным свопингом

Если машина активно выгружает и подгружает данные в своп, значение в столбце `swpd` может быть как большим, так и маленьким. Однако значения в столбцах `si` и `so` будут велики, а этого как раз допускать не следует. Вот как может выглядеть выдача *vmstat* на машине с интенсивным свопингом:

```
$ vmstat 5
procs ----------memory---------- ---swap---- -----io---- --system-- ----cpu---
r  b    swpd  free  buff    cache    si   so     bi    bo   in   cs us sy id wa
0 10 3794292 24436 27076 14412764 19853  9781 57874  9833 4084 8339  6 14 58 22
4 11 3797936 21268 27068 14519324 15913 30870 40513 30924 3600 7191  6 11 36 47
0 37 3847364 20764 27112 14547112   171 38815 22358 39146 2417 4640  6  8  9 77
```

Простаивающая машина

Для полноты картины приведем выдачу *vmstat*, полученную на простаивающей машине. Обратите внимание, что здесь нет ни готовых к выполнению, ни блокированных процессов, а столбец `idle` показывает, что процесс простаивает 100% времени. Эти данные были получены на компьютере под управлением Red Hat Enterprise Linux 5; присутствует столбец `st`, в котором отображается время, «украденное» у виртуальной машины:

```
$ vmstat 5
procs --------memory--------- --swap-- ---io--- -system- -------cpu-------
r b   swpd   free buff  cache   si   so   bi  bo   in   cs   us sy  id wa st
0 0    108 492556 6768 360092    0    0  345 209    2   65    2  0  97  1  0
0 0    108 492556 6772 360088    0    0    0  14  357   19    0  0 100  0  0
0 0    108 492556 6776 360084    0    0    0   6  355   16    0  0 100  0  0
```

8

Репликация

Встроенные в MySQL средства репликации составляют основу для построения крупных высокопроизводительных приложений. Они позволяют сконфигурировать один или несколько серверов в качестве подчиненных другому серверу; такие серверы называют репликами. Это полезно не только при создании высокопроизводительных приложений, но и во многих других случаях, например для совместного использования данных с удаленным офисом, для поддержания «горячей замены» или для хранения копии актуальных данных на другом сервере с целью тестирования или обучения.

В этой главе мы рассмотрим все аспекты репликации. Начав с обзора принципов работы, мы затем перейдем к простейшей настройке сервера и далее продемонстрируем более сложные конфигурации и расскажем о том, как управлять реплицированными серверами и оптимизировать их. Хотя эта книга посвящена, прежде всего, вопросам производительности, в деле репликации не менее важны корректность и надежность, поэтому мы остановимся и на том, как организовать правильную работу репликации. Мы обсудим также планируемые изменения и улучшения в механизме репликации MySQL, например любопытные заплаты, созданные в компании Google.

Обзор репликации

Основная задача, которую призвана решить репликация, – это синхронизация данных одного сервера с данными другого. К одному главному серверу можно подключить несколько подчиненных (slave), причем подчиненный сервер может, в свою очередь, выступать в роли главного. Топология сети главных и подчиненных серверов зачастую сильно различается. Можно реплицировать сервер целиком, или только некоторые базы данных, или даже определенные таблицы.

MySQL поддерживает две разновидности репликации: покомандную и построчную. Покомандная (или «логическая») репликация существует еще со времен версии 3.23, в настоящее время именно она обычно используется в промышленной эксплуатации. Построчная репликация появилась в версии MySQL 5.1. В обоих случаях изменения записываются в двоичный журнал[1] на главном сервере и воспроизводятся на подчиненном, причем оба варианта *асинхронны*, то есть не гарантируется, что копия данных на подчиненном сервере хотя бы в какой-то момент полностью актуальна[2]. Относительно величины отставания также не дается никаких гарантий. Если запросы сложны, то подчиненный сервер может отставать от главного на секунды, минуты и даже часы.

Репликация в MySQL в основном обратно совместима. Это означает, что сервер более поздней версии может быть подчинен серверу, на котором установлена ранняя версия MySQL. Однако старые версии обычно не могут выступать в роли подчиненных для более свежих; они не распознают новые синтаксические конструкции SQL, да и форматы файлов репликации могут отличаться. Например, невозможно реплицировать главный сервер версии MySQL 5.0 на подчиненный версии 4.0. Мы рекомендуем протестировать схему репликации до перехода на новую версию, в которую были внесены серьезные изменения, например при переходе с 4.1 на 5.0 или с 5.0 на 5.1.

Вообще говоря, накладные расходы репликации на главном сервере невелики. Правда, на нем требуется включить двоичный журнал, что само по себе весьма ощутимо, но это так или иначе нужно сделать, если вы хотите снимать нормальные резервные копии. Помимо записи в двоичный журнал небольшую нагрузку (в основном, на сеть) дает добавление каждого подчиненного сервера.

Репликация вполне применима для масштабирования операций чтения, которые можно адресовать подчиненному серверу, но для масштабирования записи она не очень подходит, если только не учесть это требование при проектировании системы. Подключение большого количества подчиненных серверов просто приводит к тому, что запись выполняется многократно, по одному разу на каждом из них. Система в целом ограничена количеством операций записи, которые может выполнить самое слабое ее звено.

При наличии нескольких подчиненных серверов репликация становится расточительным удовольствием, так как данные без нужды дублируются несколько раз. Например, если к одному главному серверу подключено 10 подчиненных, то на главном сервере оказывается 11 одинаковых копий данных, которые дублируются в 11 разных кэшах. Это можно считать аналогом RAID 1 с 11 дисками. Подобное использова-

---

[1] Информацию о двоичном журнале вы можете найти в главе 6, ниже в этой главе и в главе 11.

[2] Подробнее см. раздел «Синхронная репликация в **MySQL» на стр. 558.**

ние оборудования неэкономно, тем не менее такая конфигурация встречается на удивление часто. Ниже мы обсудим различные способы сгладить эту проблему.

Проблемы, решаемые репликацией

Перечислим несколько типичных случаев применения репликации.

Распространение данных

Обычно репликация в MySQL потребляет не очень большую часть пропускной способности сети[1], к тому же ее можно в любой момент остановить и затем возобновить. Это полезно, если хранение копии данных происходит в географически удаленном пункте, например в другом центре обработки данных. Удаленный подчиненный сервер может работать даже с непостоянным (намеренно или по другим причинам) соединением. Однако если вы хотите обеспечить минимальное отставание реплики, то следует использовать надежный канал с малым временем задержки.

Балансировка нагрузки

С помощью репликации можно распределить запросы на чтение между несколькими серверами MySQL; в приложениях с интенсивным чтением эта тактика работает очень хорошо. Реализовать несложное балансирование нагрузки можно, внеся совсем немного изменений в код. Для небольших приложений достаточно просто «зашить» в программу несколько доменных имен или воспользоваться циклическим (round-robin) разрешением DNS-имен (когда с одним доменным именем связано несколько IP-адресов). Возможны и более изощренные решения. Стандартные технологии балансирования нагрузки, в частности сетевые балансировщики, прекрасно послужат для распределения нагрузки между несколькими серверами MySQL. Неплохо зарекомендовал себя и проект Linux Virtual Server (LVS). Подробнее о балансировании нагрузки мы будем говорить в главе 9.

Резервное копирование

Репликация – это ценное подспорье для резервного копирования. Однако подчиненный сервер все же не может использоваться в качестве резервной копии и не является заменой настоящему резервному копированию.

Высокая доступность и аварийное переключение на резервный сервер (failover)

Репликация позволяет исправить ситуацию, при которой сервер MySQL является единственной точкой отказа приложения. Хорошая система аварийного переключения при отказе, имеющая в со-

[1] Впрочем, как мы увидим ниже, построчная репликация, появившаяся в версии MySQL 5.1, может порождать гораздо больший сетевой трафик, чем традиционная покомандная репликация.

ставе реплицированные подчиненные серверы, способна существенно сократить время простоя. Мы рассмотрим эту тему в главе 9.

Тестирование новых версий MySQL

Очень часто на подчиненный сервер устанавливают новую версию MySQL и перед тем как ставить ее на промышленные серверы, проверяют, что все запросы работают нормально.

Как работает репликация

Перед тем как вплотную заняться настройкой репликации, посмотрим, как же на самом деле MySQL реплицирует данные. На самом верхнем уровне репликацию можно описать в виде процедуры, состоящей из трех частей.

1. Главный сервер записывает изменения данных в двоичный журнал. Эти записи называются *событиями двоичного журнала*.
2. Подчиненный сервер копирует события двоичного журнала в свой журнал ретрансляции (relay log).
3. Подчиненный сервер воспроизводит события из журнала ретрансляции, применяя изменения к собственным данным.

Это лишь общая картина, в реальности каждый шаг весьма сложен. На рис. 8.1 схема процедуры репликации изображена более подробно.

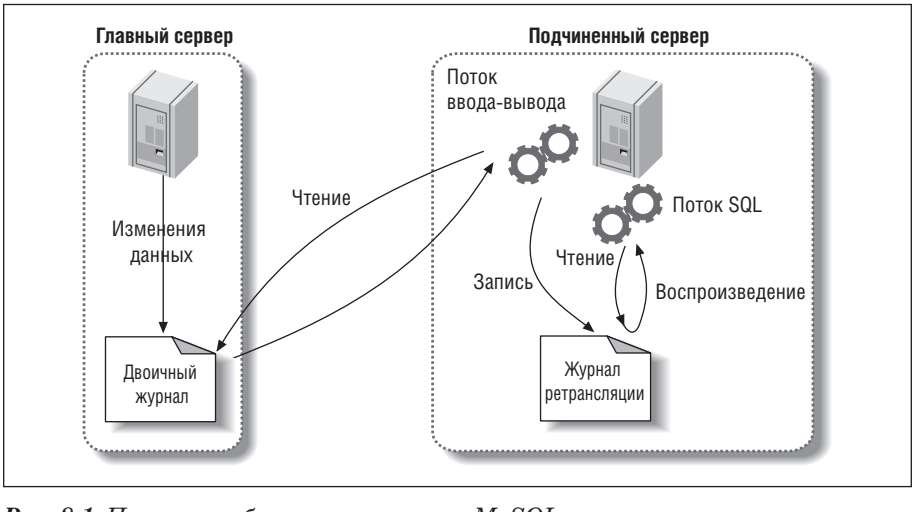
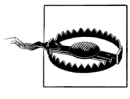


Рис. 8.1. *Принцип работы репликации в MySQL*

Первый этап данного процесса – запись в двоичный журнал на главном сервере (как ее настроить, мы объясним чуть позже). Непосредственно перед тем, как завершить транзакцию, обновляющую данные, главный сервер заносит изменения в свой двоичный журнал. MySQL записывает транзакции последовательно, даже если во время выполнения пере-

межаются команды из разных транзакций. Записав события в двоичный журнал, главный сервер просит подсистему хранения зафиксировать транзакцию.

На следующем этапе подчиненный сервер копирует двоичный журнал главного сервера на свой жесткий диск, в так называемый *журнал ретрансляции*. Первым делом он запускает *поток ввода/вывода*. Этот поток открывает обычное клиентское соединение с главным сервером, а затем запускает специальный процесс *дампа двоичного журнала* (*binlog dump*) (соответствующей команды в языке SQL не существует). Этот процесс читает события из двоичного журнала главного сервера. Он не опрашивает события активно. Обнаружив конец журнала, процесс выгрузки дампа засыпает и ждет, пока главный сервер не просигнализирует о появлении новых событий. Прочитанные события поток ввода/вывода записывает в журнал ретрансляции на подчиненном сервере.



До выхода версии MySQL 4.0 репликация во многих отношениях работала по-другому. Например, первоначально никакого журнала ретрансляции не было, поэтому для репликации использовалось два, а не три потока. Но сейчас, как правило, эксплуатируются более поздние версии сервера, поэтому рассказывать об уже неактуальных деталях мы не будем.

На последнем этапе в дело вступает *поток SQL*. Он читает и воспроизводит события из журнала ретрансляции, приводя данные на подчиненном сервере в соответствие с главным сервером. При условии, что поток SQL успевает за потоком ввода/вывода, журнал ретрансляции обычно остается в кэше операционной системы, так что накладные расходы на работу с этим журналом очень низкие. События, исполняемые потоком SQL, могут также записываться в собственный двоичный журнал подчиненного сервера, что бывает полезно в некоторых сценариях, которые мы рассмотрим ниже в этой главе.

На рис. 8.1 показано только два потока репликации на подчиненном сервере, но есть и еще один поток на главном сервере – тот, что ассоциирован с соединением, которое открыл подчиненный сервер.

Такая архитектура репликации позволяет развязать процессы выборки и воспроизведения событий на подчиненном сервере и сделать их асинхронными. Иными словами, поток ввода/вывода может работать независимо от потока SQL. Кроме того, она налагает определенные ограничения на процедуру репликации, из которых важнее всего то, что *репликация сериализуется на подчиненном сервере*. Это означает, что обновления, производившиеся на главном сервере, возможно, параллельно (в разных потоках), на подчиненном сервере распараллелены быть не могут. Как мы увидим ниже, при некоторых характеристиках рабочей нагрузки это способно стать узким местом.

Настройка репликации

В MySQL настройка репликации не вызывает особых сложностей, но у основных шагов есть много вариаций, зависящих от конкретного сценария. Самый простой случай – когда главный и подчиненный серверы только что установлены и еще не введены в эксплуатацию. На верхнем уровне процедура выглядит следующим образом:

1. Завести учетные записи репликации на каждом сервере.
2. Сконфигурировать главный и подчиненный сервера.
3. Сказать подчиненному серверу, чтобы он соединился с главным и начал реплицировать данные с него.

Здесь подразумевается, что многие принимаемые по умолчанию параметры удовлетворительны, и это действительно так, если главный и подчиненный серверы только что установлены и обладают в точности одними и теми же данными (стандартной базой `mysql`). Мы последовательно опишем действия на каждом шаге, предполагая, что серверы называются `server1` (IP-адрес 192.168.0.1) и `server2` (IP-адрес 192.168.0.2). Затем мы объясним, как инициализировать подчиненный сервер с помощью уже эксплуатируемого, и подробно рассмотрим рекомендуемую конфигурацию репликации.

Создание учетных записей репликации

В MySQL предусмотрено несколько специальных привилегий, необходимых для запуска репликации. Поток ввода/вывода, работающий на подчиненном сервере, устанавливает TCP/IP-соединение с главным сервером. Это означает, что на главном сервере должна существовать учетная запись, наделенная соответствующими привилегиями, чтобы поток ввода/вывода мог соединиться от ее имени и читать двоичный журнал главного сервера. Ниже показано, как создать такую учетную запись, – мы назвали ее *repl*:

```
mysql> GRANT REPLICATION SLAVE, REPLICATION CLIENT ON *.*
    -> TO repl@'192.168.0.%' IDENTIFIED BY 'p4ssword';
```

Такая запись создается как на главном, так и на подчиненном сервере. Отметим, что мы разрешили пользователю устанавливать соединение только из локальной сети, поскольку учетная запись репликации не защищена (дополнительную информацию о безопасности см. в главе 12).

Учетной записи репликации на самом деле нужна только привилегия `REPLICATION SLAVE` на главном сервере, а `REPLICATION CLIENT` не нужна ни на главном, ни на подчиненном сервере. Так зачем же мы их дали на обоих серверах? Тому есть две причины.

- Учетной записи, которая применяется для мониторинга и управления репликацией, нужна привилегия `REPLICATION CLIENT`, и луч-

ше не усложнять себе жизнь, а использовать одну и ту же запись для обеих целей.

- Если вы заведете учетную запись на главном сервере, а затем клонируете его для настройки подчиненного сервера, то подчиненный сервер уже будет подготовлен для роли главного на случай, если вы захотите поменять серверы ролями.

Конфигурирование главного и подчиненного серверов

Следующий шаг – настроить несколько параметров на главном сервере, в качестве которого у нас будет выступать `server1`. Необходимо включить двоичный журнал и задать идентификатор сервера. Введите (или убедитесь в наличии) такие строчки в файл *my.cnf* на главном сервере:

```
log_bin = mysql-bin
server_id = 10
```

Конкретные значения выберите по своему усмотрению. Мы пошли по простейшему пути, но вам, возможно, захочется сделать что-то похитрее.

Необходимо явно назначить серверу уникальный идентификатор. Мы задали 10, а не 1, так как значение 1 сервер обычно выбирает по умолчанию, если не задано никакое другое (это зависит от версии, некоторые версии MySQL в этом случае вообще не работают). Поэтому выбор 1 легко может привести к конфликтам между серверами, которым идентификатор явно не назначен. Часто в качестве идентификатора выбирают последний октет IP-адреса сервера, предполагая, что он уникален и в будущем не изменится (то есть все серверы находятся в одной подсети).

Если двоичный журнал ранее не был включен на главном сервере, то MySQL придется перезапустить. Чтобы проверить, создан ли на главном сервере двоичный журнал, выполните команду `SHOW MASTER STATUS` и сравните ее результат с приведенным ниже (MySQL добавляет к имени файла несколько цифр, поэтому истинное имя будет отличаться от заданного вами):

```
mysql> SHOW MASTER STATUS;
+------------------+----------+--------------+------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+------------------+----------+--------------+------------------+
| mysql-bin.000001 |       98 |              |                  |
+------------------+----------+--------------+------------------+
1 row in set (0.00 sec)
```

Файл *my.cnf* на подчиненном сервере выглядит примерно так же, как на главном, но с некоторыми дополнениями; подчиненный сервер также необходимо перезапустить:

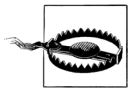
```
log_bin     = mysql-bin
server_id   = 2
```

```
relay_log         = mysql-relay-bin
log_slave_updates = 1
read_only         = 1
```

Некоторые из этих параметров, строго говоря, необязательны, а для других мы явно задали значения, совпадающие со значениями по умолчанию. На самом деле, на подчиненном сервере обязательным является только параметр `server_id`, но мы включили также `log_bin` и присвоили файлу журнала явное имя. По умолчанию имя этого файла совпадает с именем хоста, но при такой конфигурации могут возникнуть проблемы, если в будущем имя хоста изменится. Кроме того, мы хотим, чтобы журналы на обоих серверах назывались одинаково на случай, если возникнет желание превратить подчиненный сервер в главный. Исходя из этого мы не только завели на обоих серверах одноименные учетные записи, но и остальные параметры задали одинаково.

Еще мы добавили два необязательных параметра: `relay_log` (определяет имя и местоположение журнала ретрансляции) и `log_slave_updates` (чтобы подчиненный сервер записывал реплицированные события в собственный двоичный журнал). Последнее добавляет подчиненному серверу работы, но, как мы вскоре убедимся, имеются обоснованные причины для того, чтобы задавать эти параметры на всех подчиненных серверах.

Некоторые предпочитают включать только двоичный журнал, не задавая параметр `log_slave_updates`, с целью сразу же увидеть, изменяются ли какие-нибудь данные на подчиненном сервере (например, из-за неправильно сконфигурированного приложения). Если возможно, то лучше использовать параметр `read_only`, который не дает модифицировать данные никому, кроме потоков со специальными привилегиями (не выдавайте пользователям больше привилегий, чем реально необходимо для работы!) Однако часто параметр `read_only` оказывается непрактичным, особенно если некое приложение должно иметь возможность создавать таблицы на подчиненных серверах.



Не помещайте такие конфигурационные параметры, как `master_host` и `master_port`, в файл *my.cnf* на подчиненном сервере. Это устаревший способ конфигурирования подчиненного сервера. Он может привести к неприятностям и не дает никаких преимуществ.

Запуск подчиненного сервера

Следующий шаг – сообщить подчиненному серверу о том, как соединиться с главным и начать воспроизведение двоичных журналов. Для этой цели используется не файл *my.cnf*, а команда `CHANGE MASTER TO`. Она полностью заменяет соответствующие настройки в файле *my.cnf*. Кроме того, она позволяет впоследствии указать подчиненному серверу другой главный без перезапуска. Ниже приведена простейшая форма команды, необходимой для запуска репликации на подчиненном сервере:

```
mysql> CHANGE MASTER TO MASTER_HOST='server1',
    -> MASTER_USER='repl',
    -> MASTER_PASSWORD='p4ssword',
    -> MASTER_LOG_FILE='mysql-bin.000001',
    -> MASTER_LOG_POS=0;
```

Параметр MASTER_LOG_POS устанавливается в 0, потому что это начало журнала. После того как эта команда отработает, выполните команду SHOW SLAVE STATUS и проверьте, что параметры подчиненного сервера установлены правильно:

```
mysql> SHOW SLAVE STATUS\G
*************************** 1. row ***************************
       Slave_IO_State:
          Master_Host: server1
          Master_User: repl
          Master_Port: 3306
        Connect_Retry: 60
      Master_Log_File: mysql-bin.000001
  Read_Master_Log_Pos: 4
       Relay_Log_File: mysql-relay-bin.000001
        Relay_Log_Pos: 4
Relay_Master_Log_File: mysql-bin.000001
     Slave_IO_Running: No
    Slave_SQL_Running: No
          ...опущено...
Seconds_Behind_Master: NULL
```

Столбцы Slave_IO_State, Slave_IO_Running и Slave_SQL_Running показывают, что процессы репликации на подчиненном сервере не запущены. Внимательный читатель заметит также, что позиция указателя журнала равна 4, а не 0. Это объясняется тем, что 0 – это не столько истинное значение указателя, сколько признак «в начале файла журнала». MySQL знает, что данные первого события начинаются в позиции 4[1].

Чтобы запустить репликацию, выполните следующую команду:

```
mysql> START SLAVE;
```

Она не должна выводить никаких сообщений. Снова проверьте состояние подчиненного сервера:

```
mysql> SHOW SLAVE STATUS\G
*************************** 1. row ***************************
       Slave_IO_State: Waiting for master to send event
          Master_Host: server1
          Master_User: repl
```

[1] В действительности, из результата приведенной выше команды SHOW MASTER STATUS следует, что указатель находится в позиции 98. Подчиненный сервер разберется в этом вопросе, когда установит соединение с главным, чего пока не произошло.

```
          Master_Port: 3306
        Connect_Retry: 60
      Master_Log_File: mysql-bin.000001
  Read_Master_Log_Pos: 164
       Relay_Log_File: mysql-relay-bin.000001
        Relay_Log_Pos: 164
Relay_Master_Log_File: mysql-bin.000001
     Slave_IO_Running: Yes
    Slave_SQL_Running: Yes
       ...пропущено...
Seconds_Behind_Master: 0
```

Теперь на подчиненном сервере работают потоки ввода/вывода и SQL, а переменная состояния `Seconds_Behind_Master` отлична от `NULL` (что она означает, мы расскажем ниже). Поток ввода/вывода ожидает события от главного сервера, то есть в настоящий момент он прочитал из двоичного журнала все записи, которые там были. Позиции указателей в обоих журналах сместились, иными словами, какие-то события были прочитаны и обработаны (на вашем сервере картина может быть иной). Если сейчас произвести какое-нибудь изменение на главном сервере, то указатели позиций на подчиненном увеличатся. Кроме того, изменения будут применены к подчиненному серверу!

Потоки репликации должны быть видны в списках процессов на главном и подчиненном серверах. На главном вы увидите соединение, созданное потоком ввода/вывода, который работает на подчиненном сервере:

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 55
   User: repl
   Host: slave1.webcluster_1:54813
     db: NULL
Command: Binlog Dump
   Time: 610237
  State: Has sent all binlog to slave; waiting for binlog to be updated
   Info: NULL
```

На подчиненном сервере должно быть два потока: ввода/вывода и SQL:

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 1
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 611116
  State: Waiting for master to send event
   Info: NULL
*************************** 2. row ***************************
     Id: 2
   User: system user
```

```
     Host:
       db: NULL
  Command: Connect
     Time: 33
    State: Has read all relay log; waiting for the slave I/O thread to update it
     Info: NULL
```

Приведенные нами примеры распечаток получены на серверах, которые проработали достаточно долго, поэтому в столбце `Time` для потока ввода/вывода на главном и подчиненном серверах выводится большое значение. SQL-поток на подчиненном сервере простаивает 33 секунды, то есть в течение этого времени не было воспроизведено ни одного события.

Эти процессы всегда работают от имени учетной записи «system user», но значения в остальных столбцах могут отличаться от показанных. Например, когда поток SQL воспроизводит событие на подчиненном сервере, в столбце `Info` отображается исполняемый запрос.

Если вы хотите просто поэкспериментировать с репликацией, попробуйте сценарий MySQL Sandbox, написанный Джузеппе Максиа (Giuseppe Maxia) (*http://sourceforge.net/projects/mysql-sandbox/)*. Он позволяет быстро создать из дистрибутивного tgz-файла новую инсталляцию MySQL, которую потом можно будет безболезненно удалить. Чтобы получить работающие главный и два подчиненных сервера, достаточно нескольких нажатий клавиш и 15 секунд.

```
$ ./set_replication.pl ~/mysql-5.0.45-linux-x86_64-glibc23.tar.gz
```

Инициализация подчиненного сервера на основе существующего

Выше мы предполагали, что главный и подчиненный серверы только что установлены, поэтому данные на них практически одинаковы и позиция указателя в файле двоичного журнала известна. Но на практике так обычно не бывает. Как правило, уже существует главный сервер, который проработал какое-то время, и требуется синхронизировать с ним новый подчиненный сервер, на котором еще нет копии данных с главного.

Существует несколько способов инициализировать, или «клонировать», подчиненный сервер из имеющегося: копирование данных с главного, клонирование другого подчиненного сервера и загрузка на подчиненный сервер данных из свежей резервной копии. Чтобы синхронизировать подчиненный сервер с главным, необходимы три вещи.

- Мгновенный снимок данных главного сервера в некоторый момент времени.
- Текущий файл журнала главного сервера и смещение от начала этого файла в точности на тот момент времени, когда был сделан мгно-

венный снимок. Вместе они называются *координатами репликации*, так как однозначно идентифицируют позицию в двоичном журнале. Найти координаты репликации вам поможет команда `SHOW MASTER STATUS`.

- Файлы двоичных журналов главного сервера с момента мгновенного снимка до текущего момента.

Существует несколько способов клонировать подчиненный сервер с помощью другого сервера.

Холодная копия

Самый простой способ запустить подчиненный сервер состоит в том, чтобы остановить сервер, который впоследствии станет главным, и скопировать файлы с него на подчиненный сервер (об эффективных способах копирования файлов см. приложение А). После перезапуска главный сервер откроет новый двоичный журнал и можно будет воспользоваться командой `CHANGE MASTER TO`, указав на подчиненном сервере начало файла в качестве позиции в двоичном журнале. Недостаток такого решения очевиден: в течение всего времени копирования главный сервер должен быть остановлен.

Горячая копия

Если все таблицы имеют тип MyISAM, то можно воспользоваться командой *mysqlhotcopy*, которая копирует файлы с работающего сервера. Подробную информацию см. в главе 11.

Использование mysqldump

Если все таблицы имеют тип InnoDB, то чтобы выгрузить данные с главного сервера в дамп, загрузить их на подчиненный и изменить координаты репликации на подчиненном сервере в соответствии с позицией в двоичном журнале главного сервера, можно воспользоваться такой командой:

```
$ mysqldump --single-transaction --all-databases
--master-data=1 --host=server1 | mysql --host=server2
```

Флаг *--single-transaction* говорит, что при выгрузке нужно читать те данные, которые существовали на момент начала транзакции. Возможно, он работает и для других транзакционных систем хранения, но мы этого не проверяли. Если имеются нетранзакционные таблицы, то для получения согласованного дампа всех таблиц следует задать флаг *--lock-all-tables*.

С помощью мгновенного снимка LVM или резервной копии

Если известны координаты в нужном двоичном журнале, то для инициализации подчиненного сервера можно воспользоваться мгновенным снимком LVM или резервной копией (в последнем случае необходимо иметь все двоичные журналы главного сервера с момента снятия этой копии). Восстановите данные из мгновенного снимка или копии на подчиненном сервере, а затем задайте координаты ре-

пликации с помощью команды CHANGE MASTER TO. Дополнительную информацию об этом способе см. в главе 11.

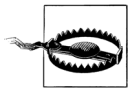
Технология InnoDB Hot Backup, которая также рассматривается в главе 11, – еще один удобный способ инициализировать подчиненный сервер в ситуации, когда все таблицы имеют тип InnoDB.

На основе другого подчиненного сервера

Любой из вышеупомянутых методов годится для клонирования одного подчиненного сервера из другого. Однако флаг *--master-data* в команде *mysqldump* работать не будет.

Еще отметим, что вместо того, чтобы получать координаты репликации в двоичном журнале главного сервера командой SHOW MASTER STATUS, следует воспользоваться командой SHOW SLAVE STATUS для отыскания позиции в двоичном журнале главного сервера, на которой подчиненный сервер остановился в момент снятия мгновенного снимка.

Серьезный недостаток клонирования другого подчиненного сервера состоит в том, что если подчиненный сервер рассинхронизирован с главным, то вы будет клонировать неактуальные данные.



Не пользуйтесь командами LOAD DATA FROM MASTER и LOAD TABLE FROM MASTER! Они устарели, работают медленно и крайне опасны. К тому же они применимы только к таблицам типа MyISAM.

На каком бы методе вы ни остановились, потратьте время на то, чтобы освоиться с ним, и документируйте свои действия или напишите сценарий. Не исключено, что эту процедуру придется проделать не раз, и вы не должны растеряться, если что-то пойдет не так.

Рекомендуемая конфигурация репликации

Параметров репликации много, и большинство из них так или иначе влияют на безопасность данных и производительность. Ниже мы расскажем о том, какие правила можно нарушать и когда. А в этом разделе приведем рекомендуемую «безопасную» конфигурацию, которая сводит к минимуму шансы нарваться на неприятность.

На главном сервере самым важным параметром для двоичного журналирования является sync_binlog :

```
sync_binlog=1
```

При таком значении MySQL сбрасывает двоичный журнал на диск в момент фиксации транзакции, поэтому события журнала не потеряются в случае сбоя. Если отключить этот режим, то работы у сервера станет меньше, но при возникновении сбоя записи в журнале могут оказаться поврежденными или вообще отсутствовать. На подчиненном сервере, который не выступает в роли главного, этот режим приводит к излишним накладным расходам. Он применяется только к двоичному журналу, а не к журналу ретрансляции.

Если повреждение таблиц после сбоя неприемлемо, то мы рекомендуем работать с InnoDB. Подсистема хранения MyISAM хороша, если с поврежденной таблицей можно примириться, но имейте в виду, что после сбоя подчиненного сервера таблицы типа MyISAM могут оказаться в несогласованном состоянии. Есть вероятность, что некая команда будет применена к одной или нескольким таблицам не полностью, поэтому данные останутся несогласованными даже после исправления таблиц.

При использовании InnoDB мы настоятельно рекомендуем задавать на главном сервере следующие параметры:

```
innodb_flush_logs_at_trx_commit=1 # Сброс после каждой записи в журнал
innodb_support_xa=1               # Только в версии MySQL 5.0 и более поздних
innodb_safe_binlog                # только в версии MySQL 4.1, примерный
                                  # эквивалент innodb_support_xa
```

Эти значения подразумеваются по умолчанию в версии MySQL 5.0. На подчиненном сервере мы рекомендуем включить следующие параметры:

```
skip_slave_start
read_only
```

Параметр `skip_slave_start` предотвращает автоматический перезапуск подчиненного сервера после сбоя, это оставляет вам возможность восстановить сервер при наличии проблем. Если же подчиненный сервер перезапускается автоматически после некорректного завершения, а база данных находится в несогласованном состоянии, то повреждение может дойти до такой степени, что все данные придется выбросить и начать все сначала. Даже если вы установите все параметры так, как мы предлагаем, подчиненный сервер все равно может выйти из строя после сбоя, поскольку журналы ретрансляции и файл *master.info* не защищены от повреждений. Они даже не сбрасываются принудительно на диск, и не существует никакого параметра, который управлял бы этим поведением. Разработанная компанией Google заплата, о которой мы поговорим ниже, решает эту проблему.

Параметр `read_only` не дает большинству пользователей изменять какие-либо таблицы, кроме временных. Исключение составляют поток SQL и потоки, работающие с привилегией `SUPER`. Это единственная причина, по которой обычной учетной записи имеет смысл давать привилегию `SUPER` (о привилегиях см. главу 12).

Если подчиненный сервер очень сильно отстает от главного, то поток ввода/вывода может создать множество журналов ретрансляции. Поток SQL удаляет их сразу после воспроизведения (это поведение можно изменить с помощью параметра `relay_log_purge`), но если отставание велико, то поток ввода/вывода вполне может заполнить весь диск. Справиться с этой проблемой поможет конфигурационный параметр `relay_log_space_limit`. Если совокупный размер всех журналов ретрансляции больше значения этого параметра, то поток ввода/вывода приостанавливается и ждет, пока поток SQL освободит место на диске.

На первый взгляд, все хорошо, но здесь таится одна проблема. Если подчиненный сервер не скопировал в журнал ретрансляции все события с главного сервера, то в случае сбоя последнего они могут быть навсегда потеряны. Если места на диске достаточно, то лучше дать возможность подчиненному серверу создавать журналы ретрансляции без ограничений. Именно поэтому мы не включили параметр `relay_log_space_limit` в рекомендуемую конфигурацию.

Взгляд на репликацию изнутри

Теперь, когда мы познакомились с основами репликации, можно копнуть и поглубже. Мы рассмотрим, как на самом деле работает механизм репликации, познакомимся с его сильными и слабыми сторонами и изучим некоторые дополнительные конфигурационные параметры.

Покомандная репликация

Версии MySQL 5.0 и более ранние поддерживали только *покомандную репликацию* (она также называется *логической*). В мире СУБД это необычно. Принцип работы такого механизма заключается в том, что протоколируются все выполненные главным сервером команды изменения данных. Когда подчиненный сервер читает из своего журнала ретрансляции событие и воспроизводит его, на самом деле он отрабатывает в точности ту же команду, которая была ранее выполнена на главном сервере. У такого решения есть свои плюсы и минусы.

Очевидный плюс – относительная легкость реализации. Простое журналирование и воспроизведение всех предложений, изменяющих данные, теоретически поддерживает синхронизацию подчиненного сервера с главным. Еще одно достоинство покомандной репликации состоит в том, что события в двоичном журнале представлены компактно. Иначе говоря, покомандная репликация потребляет не слишком большую часть пропускной способности сети – запрос, обновляющий гигабайты данных, занимает всего-то несколько десятков байтов в двоичном журнале. Кроме того, уже упоминавшийся инструмент *mysqlbinlog* удобнее использовать именно для покомандной репликации.

На практике, однако, покомандная репликация не так проста, как кажется, поскольку многие изменения на главном сервере могут зависеть от факторов, не выражаемых в тексте запроса. Например, моменты выполнения команд на главном и подчиненном сервере могут слегка – или даже сильно – различаться. Поэтому в двоичном журнале MySQL присутствует не только текст запроса, но и кое-какие метаданные, такие как временная метка. Но все равно некоторые команды невозможно реплицировать корректно, в частности, запросы, в которых встречается функция `CURRENT_USER()`. Проблемы возникают также с хранимыми процедурами и триггерами.

С покомандной репликацией связана еще одна неприятность – модификации должны быть сериализуемы. Для этого приходится обрабатывать в коде сервера различные особые случаи, вводить специальные параметры и неоправданные функции. Например, в этой связи можно упомянуть блокировку следующего ключа в InnoDB и блокировку при выполнении автоинкремента. Не все подсистемы хранения корректно поддерживают покомандную репликацию, хотя подсистемы, включенные в официальный дистрибутив MySQL вплоть до версии 5.1, с этим справляются.

Полный перечень недостатков покомандной репликации можно найти в соответствующей главе руководства по MySQL.

Построчная репликация

В версии MySQL 5.1 добавилась поддержка *построчной репликации*, при которой в двоичный журнал записываются фактические изменения данных, как это делается в большинстве других СУБД. У такой схемы есть свои плюсы и минусы. Самое существенное достоинство заключается в том, что теперь MySQL может корректно реплицировать любую команду, причем в некоторых случаях это происходит гораздо более эффективно. Основной недостаток – это то, что двоичный журнал стал намного больше и из него непонятно, какие команды привели к обновлению данных, так что использовать его для аудита с помощью программы *mysqlbinlog* уже невозможно.

Построчная репликация не является обратно совместимой. Утилита *mysqlbinlog*, входящая в дистрибутив MySQL 5.1, может читать двоичные журналы, содержащие события в формате построчной репликации (он не предназначен для чтения человеком). Однако версии *mysqlbinlog* из предыдущих версий MySQL такой журнал не распознают и при попытке прочитать его завершаются с ошибкой.

Некоторые изменения, хранящиеся в формате построчной репликации, MySQL воспроизводит более эффективно, так как ему не приходится повторять запросы, выполненные на главном сервере. А ведь воспроизведение определенных запросов обходится весьма дорого. Например, следующий запрос агрегирует данные из большой таблицы, помещая результаты в таблицу, меньшую по размерам:

```
mysql> INSERT INTO summary_table(col1, col2, sum_col3)
    -> SELECT col1, col2, sum(col3)
    -> FROM enormous_table
    -> GROUP BY col1, col2;
```

Предположим, что в таблице `enormous_table` есть всего три уникальных комбинации столбцов `col1` и `col2`. В этом случае запрос просматривает очень много строк в исходной таблице, но результатом является вставка лишь трех строк в конечную таблицу. Если это событие реплицировать

как команду, то подчиненный сервер должен будет повторить всю работу для того, чтобы вычислить эти три строки, тогда как построчная репликация обходится до смешного дешево. В данном случае построчная репликация многократно эффективнее.

С другой стороны, следующее событие гораздо дешевле обрабатывается методом покомандной репликации:

```
mysql> UPDATE enormous_table SET col1 = 0;
```

Применение построчной репликации для данного запроса обходится очень дорого, так как изменяется каждая строка, следовательно, каждую строку нужно будет записать в двоичный журнал, который вырастет до неимоверных размеров. В результате нагрузка на главный сервер резко возрастает как на этапе сохранения данных в журнале, так и на этапе репликации, а из-за медленной записи в журнал может пострадать конкурентность.

Поскольку ни тот, ни другой формат не идеален, MySQL 5.1 динамически переключается с одного на другой по мере необходимости. По умолчанию применяется покомандная репликация, но если обнаруживается событие, которое невозможно корректно реплицировать командой, то сервер переходит на построчную репликацию. Разрешается также явно управлять форматом с помощью сеансовой переменной `binlog_format`.

Если двоичный журнал представлен в формате построчной репликации, то восстановить данные на конкретный момент времени в прошлом становится более сложно, но все-таки возможно. В этом может помочь сервер журнала, но подробнее об этом мы поговорим ниже.

Теоретически построчная репликация решает некоторые из вышеупомянутых проблем. На практике же многие наши знакомые, работающие с MySQL 5.1, по-прежнему предпочитают покомандную репликацию на промышленных серверах. Поэтому пока о построчной репликации трудно сказать что-нибудь определенное.

Файлы репликации

Теперь рассмотрим некоторые файлы, участвующие в процессе репликации. О двоичном журнале и журнале ретрансляции вы уже знаете, но есть и другие файловые объекты. Конкретное место их хранения зависит в основном от конфигурационных параметров MySQL. В разных версиях СУБД умолчания различны. Чаще всего их можно найти в каталоге данных или в каталоге, где находится *pid*-файл сервера (в UNIX-системах это обычно каталог */var/run/mysqld/*). Перечислим их.

mysql-bin.index

Если запись в двоичный журнал включена, то сервер создает файл с таким же именем, как у двоичных журналов, но с расширением

.index. В нем регистрируются все файлы двоичных журналов, имеющиеся на диске. Это не индекс в том смысле, в каком мы говорим об индексах по таблицам; он просто состоит из текстовых строк, в каждой из которых указано имя одного файла двоичного журнала.

Вероятно, у вас возникла мысль, что этот файл лишний и может быть удален (в конце концов, MySQL может просто найти все файлы на диске). Не делайте этого! MySQL игнорирует двоичные журналы, не указанные в индексном файле.

mysql-relay-bin.index

Этот файл играет ту же роль для журналов ретрансляции, что рассмотренный выше файл для двоичных журналов.

master.info

В этом файле хранится информация, необходимая подчиненному серверу для соединения с главным. Формат текстовый (по одному значению в строке) и изменяется в зависимости от версии MySQL. Не удаляйте его, иначе после перезапуска подчиненный сервер не будет знать, как подключиться к главному. Поскольку в этом файле может храниться пароль пользователя в открытом виде, будет разумно максимально ограничить права доступа к нему.

relay-log.info

В этом файле на подчиненном сервере хранятся имя его текущего двоичного журнала и координаты репликации (то есть место в двоичном журнале главного сервера, до которого дошел подчиненный). Не удаляйте этот файл, иначе после перезапуска подчиненный сервер не будет знать, с какого места продолжить репликацию, и попытается воспроизвести уже выполненные команды.

Совокупность этих файлов представляет собой довольно прямолинейный способ сохранить состояние репликации и журналов. К сожалению, запись в них производится не синхронно, поэтому если произойдет сбой питания в момент, когда файлы не были сброшены на диск, то после перезапуска данные в них окажутся некорректными.

По умолчанию имя двоичного журнала образуется из имени хоста, к которому добавляется цифровой суффикс, но лучше задать базовое имя явно в файле *my.cnf*:

```
log_bin             # Не делайте этого, если не хотите,
                    # чтобы имя образовывалось из имени хоста
log_bin = mysql-bin # Так безопасно
```

Это существенно, поскольку в случае изменения имени хоста репликация может перестать работать. Мы также не рекомендуем выбирать в качестве базового имени имя хоста; иными словами, не делайте умолчание явным решением. Лучше выберите какое-то одно имя для двоичных журналов и используйте его на всех серверах. Тогда будет гораздо

проще перемещать файлы сервера с одной машины на другую и автоматизировать переключение в случае сбоя (failover).

Следует также явно выбирать имена для журналов ретрансляции (которые тоже по умолчанию образуются из имени хоста) и соответствующих *index*-файлов. Вот как мы рекомендуем задавать все эти параметры в файле *my.cnf*:

```
log_bin         = mysql-bin
log_bin_index   = mysql-bin.index
relay_log       = mysql-relay-bin
relay_log_index = mysql-relay-bin.index
```

Вообще-то *index*-файлы по умолчанию наследуют имена от соответствующих файлов журналов, но не будет никакого вреда, если задать их явно.

На *index*-файлы влияет также параметр `expire_logs_days`, определяющий, сколько времени MySQL должен сохранять уже закрытые двоичные журналы. Если в файле *mysql-bin.index* упоминаются файлы, отсутствующие на диске, то автоматическое удаление работать не будет; даже команда `PURGE MASTER LOGS` ничего не даст. В общем случае для решения этой проблемы нужно поручить управление двоичными журналами самому серверу MySQL, уж он-то не запутается.

Необходимо выработать стратегию удаления старых журналов – с помощью параметра `expire_logs_days` или иными средствами, – иначе рано или поздно MySQL заполнит двоичными журналами весь диск. При этом следует учитывать и принятую в организации политику резервного копирования. Дополнительную информацию о двоичном журнале см. в разделе «Формат двоичного журнала» на стр. 601.

Отправка событий репликации другим подчиненным серверам

Параметр `log_slave_updates` позволяет использовать подчиненный сервер в роли главного для других подчиненных. Он заставляет сервер MySQL записывать события, выполняемые потоком SQL, в собственный двоичный журнал, доступный подчиненным ему серверам. Эта схема изображена на рис. 8.2.

В данном случае любое изменение на главном сервере приводит к записи события в его двоичный журнал. Первый подчиненный сервер извлекает и исполняет это событие. Обычно на этом жизнь события и завершилась бы, но, поскольку включен режим `log_slave_updates`, подчиненный сервер записывает его в свой двоичный журнал. Теперь второй подчиненный сервер может извлечь это событие и поместить в свой журнал ретрансляции. Такая конфигурация означает, что изменения, произведенные на главном сервере, распространяются по цепочке подчиненных серверов, не подключенных напрямую к главному. Мы предпо-

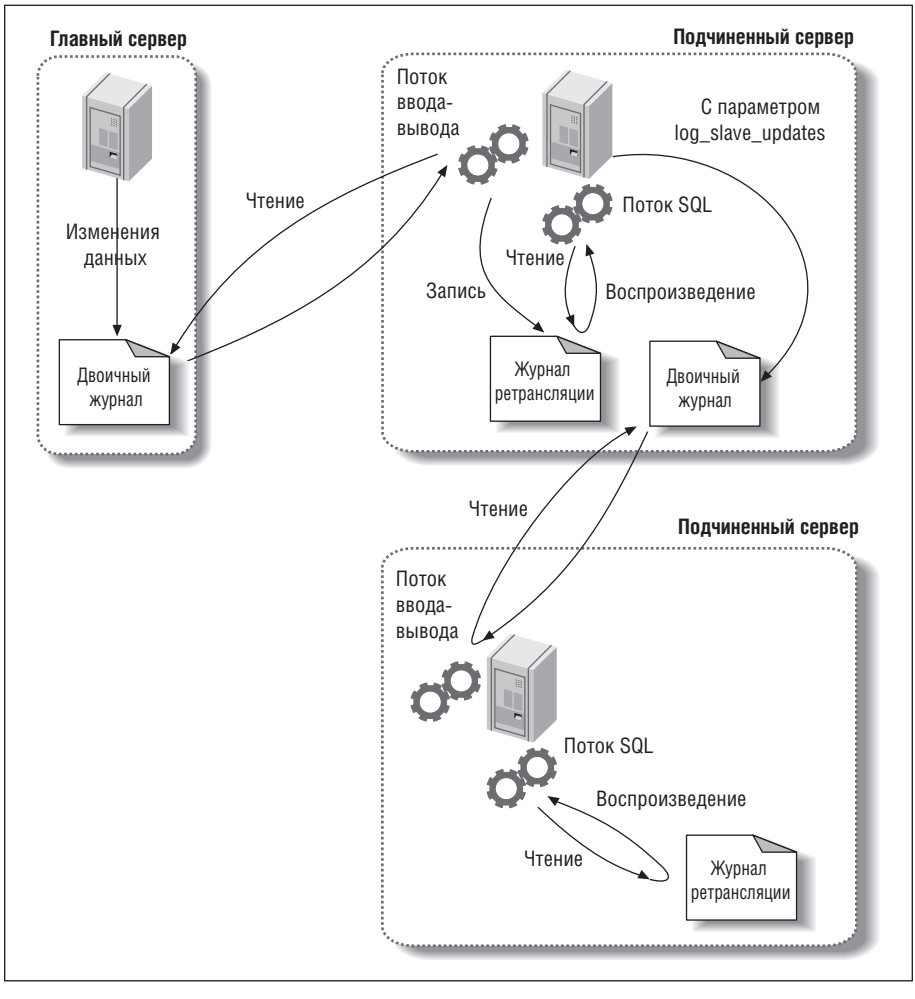


Рис. 8.2. *Передача события репликации по цепочке подчиненных серверов*

читаем по умолчанию оставлять режим `log_slave_updates` включенным, так как это позволяет подключать подчиненный сервер без перезапуска сервера.

Когда первый подчиненный сервер переписывает событие из двоичного журнала главного сервера в свой двоичный журнал, его позиция в журнале почти наверняка изменится, то есть она может оказаться либо в другом файле, либо по другому смещению. Поэтому нельзя предполагать, что все серверы, логически находящиеся в одной и той же точке репликации, имеют одинаковые координаты репликации. Позже мы увидим, что это существенно осложняет решение некоторых задач,

например подключение подчиненного сервера к другому главному или преобразование подчиненного сервера в главный.

Если не позаботиться о том, чтобы у каждого сервера был уникальный идентификатор, то подобное конфигурирование подчиненного сервера может послужить причиной трудноуловимых ошибок и даже привести к полной остановке репликации. Часто спрашивают, зачем нужно присваивать серверу идентификатор. Разве MySQL не может реплицировать команды, не зная их происхождения? Почему MySQL хочет, чтобы идентификатор сервера был глобально уникальным? А все дело в том, чтобы предотвратить бесконечные циклы в процедуре репликации. Когда поток SQL на подчиненном сервере читает журнал ретрансляции, он отбрасывает все события, в которых идентификатор сервера совпадает с его собственным. Тем самым бесконечный цикл разрывается. Предотвращение бесконечных циклов важно в таких важных топологиях репликации, как главный–главный (master–master).

При возникновении затруднений с настройкой репликации первым делом обратите внимание на идентификатор сервера. Недостаточно просто проверить переменную `@@server_id`. У нее всегда есть какое-то значение по умолчанию, но репликация не будет работать, если значение не задано явно – в файле *my.cnf* или командой `SET`. Если вы пользуетесь командой `SET`, не забудьте также обновить конфигурационный файл, иначе измененные настройки пропадут после перезапуска сервера.

Фильтры репликации

Параметры фильтрации позволяют реплицировать только часть данных, хранящихся на сервере. Есть два вида фильтров репликации: одни применяются при записи событий в двоичный журнал на главном сервере, а другие – при чтении событий из журнала ретрансляции на подчиненном сервере. Те и другие изображены на рис. 8.3.

Фильтрацией двоичного журнала управляют параметры `binlog_do_db` и `binlog_ignore_db`. Но, как мы скоро поясним, их как раз использовать *не стоит*.

Параметры `replicate_*` управляют фильтрацией событий, считываемых потоком SQL из журнала ретрансляции на подчиненном сервере. Можно реплицировать или игнорировать одну или несколько баз данных, переписывать одну базу данных в другую и реплицировать или игнорировать определенные таблицы, задаваемые с помощью предиката `LIKE`.

Очень важно понимать, что параметры `*_do_db` и `*_ignore_db` – как на главном, так и на подчиненном сервере, – работают не так, как можно было бы ожидать. Естественно полагать, что фильтрация производится по имени базы данных объекта, но на самом деле анализируется имя

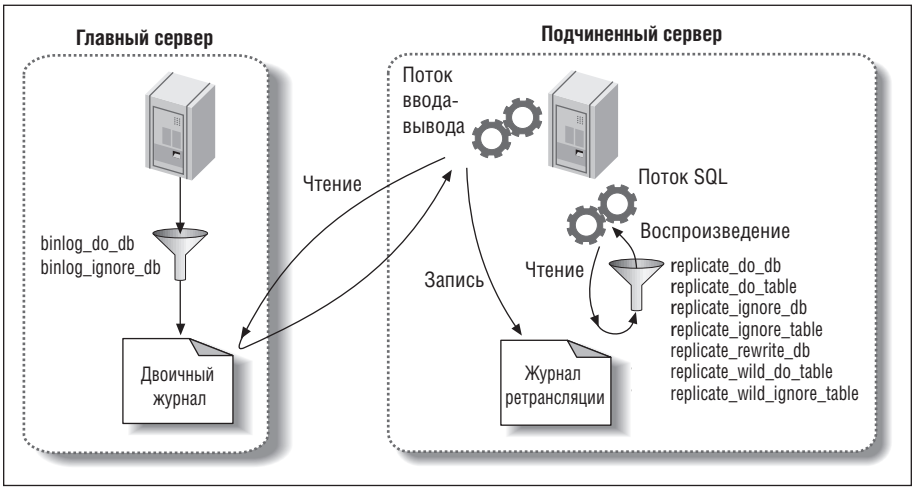
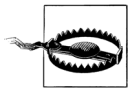


***Рис. 8.3.** Фильтры репликации*

текущей базы данных по умолчанию. Иными словами, если выполнить на главном сервере такие команды:

```
mysql> USE test;
mysql> DELETE FROM sakila.film;
```

то параметры `*_do_db` и `*_ignore_db` будут отфильтровывать команду `DELETE` в базе `test`, а не в базе `sakila`. Обычно это совсем не то, что требуется, а в результате будут реплицироваться или игнорироваться не те команды. Хотя у параметров `*_do_db` и `*_ignore_db` есть полезные применения, нужны они бывают редко, так что будьте осторожны. Если вы ими воспользуетесь, то реплики очень легко могут оказаться рассогласованными.



Параметры `binlog_do_db` и `binlog_ignore_db` могут не только нарушить репликацию, но и делают невозможным восстановление данных на конкретный момент времени в прошлом. Поэтому старайтесь не применять их. Ниже в этой главе мы покажем безопасные способы фильтровать события репликации с помощью таблиц-«черных дыр».

Типичное применение фильтрации – это предотвращение репликации команд `GRANT` и `REVOKE` на подчиненные серверы[1]. Часто администратор выдает какому-нибудь пользователю привилегию на запись с помощью команды `GRANT` на главном сервере, а затем обнаруживает, что та же при-

[1] Предпочтительный способ ограничить привилегии на подчиненных серверах состоит в использовании параметра `read_only` и поддержании одинаковых привилегий на главном и подчиненных серверах.

вилегия распространилась и на подчиненный сервер, где этому пользователю запрещено изменять данные. Для предотвращения такого развития событий следует задать параметры репликации следующим образом:

```
replicate_ignore_table=mysql.columns_priv
replicate_ignore_table=mysql.db
replicate_ignore_table=mysql.host
replicate_ignore_table=mysql.procs_priv
replicate_ignore_table=mysql.tables_priv
replicate_ignore_table=mysql.user
```

Иногда советуют просто отфильтровать все таблицы в базе данных `mysql`, например с помощью такого правила:

```
replicate_wild_ignore_table=mysql.%
```

Конечно, команды `GRANT` вы подобным образом отфильтруете, но заодно не будут реплицироваться события и подпрограммы. Вот из-за таких непредвиденных последствий мы и призываем быть очень аккуратными при работе с фильтрами. Возможно, стоит вместо этого воспрепятствовать репликации конкретных команд, обычно с помощью команды `SET SQL_LOG_BIN=0`, но с такой практикой сопряжены другие опасности. В общем случае к фильтрам репликации следует подходить с особой осторожностью и применять их только в случае острой необходимости, потому что нарушить покомандную репликацию с их помощью ничего не стоит. Некоторые из описанных проблем теоретически может разрешить построчная репликация, но на практике это пока стопроцентно не доказано.

Параметры фильтрации хорошо документированы в руководство по MySQL, поэтому мы не станем повторяться.

Топологии репликации

MySQL позволяет настроить репликацию чуть ли не для любой конфигурации главных и подчиненных серверов с одним ограничением: у каждого подчиненного сервера может быть только один главный. Возможны различные сложные топологии, но даже совсем простые обладают немалой гибкостью. У одной и той же топологии существует много различных применений. Имейте это в виду, читая последующие разделы, поскольку мы описываем только наиболее простые применения. Разнообразия способов использования репликации хватило бы на целую книгу.

Мы уже видели, как настроить главный сервер с одним подчиненным. В этом разделе мы рассмотрим другие распространенные топологии

и обсудим их сильные стороны и ограничения. По ходу изложения не забывайте о нескольких базовых правилах:

- У каждого подчиненного сервера MySQL может быть только один главный.
- У каждого подчиненного сервера должен быть уникальный идентификатор.
- Один главный сервер может иметь много подчиненных (иными словами, у подчиненного сервера может быть много «братьев»).
- Подчиненный сервер может распространять полученные от главного изменения далее, то есть выступать в роли главного сервера для своих подчиненных; для этого следует включить режим `log_slave_updates`.

Один главный сервер с несколькими подчиненными

Если исключить рассмотренную выше конфигурацию с двумя серверами главный–подчиненный, то эта топология самая простая. На самом деле она ничуть не сложнее базовой: подчиненные сервера никак не взаимодействуют между собой; все подчиненные сервера подключены к главному. Такая схема изображена на рис. 8.4.

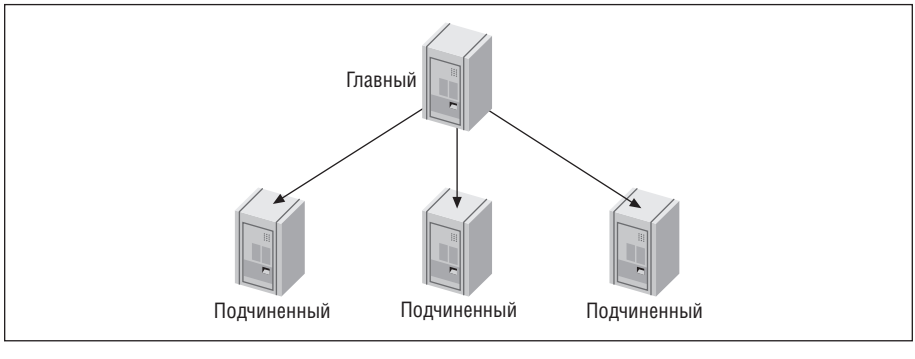


***Рис. 8.4.** Один главный сервер с несколькими подчиненными*

Эта конфигурация наиболее полезна, когда операций записи мало, а операций чтения много. Для чтения можно выделить сколько угодно подчиненных серверов при условии, что репликация данных на них не занимает слишком большую долю полосы пропускания сети и не «съедает» слишком много ресурсов у главного сервера. Подчиненные серверы можно включить все сразу или добавлять постепенно, как было описано выше в настоящей главе.

Хотя указанная топология очень проста, ее гибкости достаточно для решения различных задач. Приведем несколько идей:

- Задействовать подчиненные серверы для разных ролей (например, можно построить другие индексы или использовать другие подсистемы хранения).
- Настроить один из подчиненных серверов как резервный на случай выхода главного из строя; никаких других действий, кроме репликации, на нем не производится.
- Поставить один подчиненный сервер в удаленный центр обработки данных для восстановления в случае катастроф (например, пожар).
- Настроить задержку применения изменений на подчиненном сервере для восстановления данных.
- Использовать один из подчиненных серверов для резервного копирования, обучения, разработки или предварительной подготовки данных.

Одна из причин популярности данной топологии – это то, что она позволяет избежать сложностей, присущих другим конфигурациям. Например, легко сравнить один подчиненный сервер с другим исходя из позиции в двоичном журнале на главном сервере, поскольку все они должны быть одинаковы. Иными словами, если остановить все подчиненные серверы в одной и той же логической точке репликации, то окажется, что все они читают из одного физического места в журнале главного сервера. Это весьма полезное свойство упрощает ряд административных задач, в частности, преобразование подчиненного сервера в главный.

Упомянутым свойством характеризуются только подчиненные «сервера-братья». Сравнивать позиции в журнале для серверов, не связанных прямым отношением главный-подчиненный и не являющихся братьями, гораздо сложнее. Для многих топологий, рассматриваемых ниже, например, древовидной репликации или конфигурации с главным сервером-распределителем, трудно понять, в какой точке логической последовательности событий находится подчиненный сервер.

Главный–главный в режиме активный–активный

Репликация типа главный–главный (или репликация с двумя главными, или двунаправленная репликация) подразумевает наличие двух серверов, каждый из которых сконфигурирован одновременно как главный и подчиненный по отношению к другому (рис. 8.5).

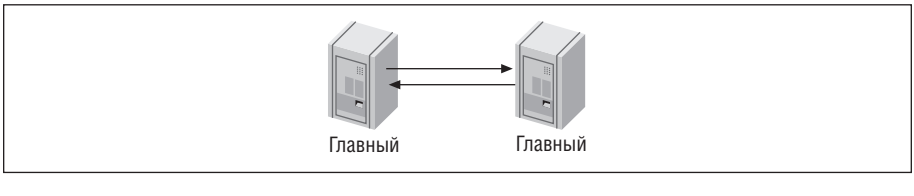


Рис. 8.5. *Репликация главный–главный*

MySQL не поддерживает репликацию с несколькими главными серверами

Мы используем термин репликацию с несколькими главными серверами (multimaster replication) для описания ситуации, когда один подчиненный сервер связан с несколькими главными. Чтобы вам ни говорили по этому поводу, MySQL (в отличие от некоторых других СУБД) пока не поддерживает конфигурацию, изображенную на рис. 8.6. Однако ниже мы все же покажем, как можно эмулировать такую конфигурацию.

К сожалению, этот термин часто используют небрежно, имея в виду любую топологию, где имеется более одного главного сервера, например древовидную, которая будет описана ниже. А иногда так называют репликацию типа главный–главный, когда каждый из двух серверов является для другого и главным и подчиненным.

Такого рода терминологический разнобой вызывает путаницу и даже споры, поэтому мы призываем строже подходить к названиям. Представьте, как трудно станет находить общий язык, когда в MySQL будет введена поддержка одного подчиненного сервера с несколькими главными! Как ее называть, если не зарезервировать словосочетание «репликация с несколькими главными серверами» специально для этой цели?

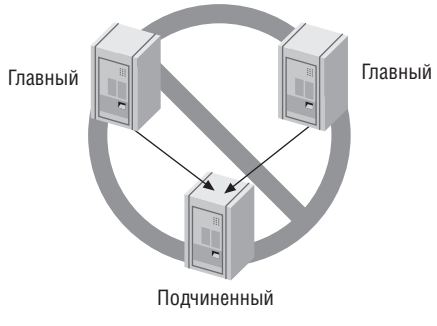


Рис. 8.6. *MySQL не поддерживает репликацию с несколькими главными серверами*

У репликации типа главный–главный в режиме активный–активный есть применения, но обычно узкоспециализированные. Например, ее можно использовать в географически разнесенных отделениях, в каждом из которых необходимо изменять данные.

Основная возникающая в этом случае проблема – как обрабатывать конфликтующие изменения. Но перечень проблем, связанных с наличием двух равноправных главных серверов, этим далеко не исчерпывается. Обычно сложности появляются, когда одна и та же строка одновременно изменяется на обоих серверах или в один и тот же момент времени производится вставка в таблицу с автоинкрементным столбцом.

В версии MySQL 5.0 были добавлены средства, сделавшие этот вид репликации немного безопаснее: параметры `auto_increment_increment` и `auto_increment_offset`. Они позволяют серверам автоматически генерировать неконфликтующие значения в запросах `INSERT`. Но все равно разрешать запись на обоих главных серверах опасно. Если на двух машинах обновления производятся в разном порядке, то данные могут незаметно рассинхронизироваться. Пусть, например, имеется таблица с одной строкой и одним столбцом, в которой находится значение `1`. И пусть одновременно выполняются такие две команды:

- На первом сервере:

```
mysql> UPDATE tbl SET col=col + 1;
```

- На втором сервере:

```
mysql> UPDATE tbl SET col=col * 2;
```

Что мы получим? На одном сервере значение `4`, а на другом – `3`. А главное, ни о каких ошибках подсистема репликации не сообщит.

Рассинхронизация данных – это еще цветочки. Что если репликация по какой-то причине остановится, а приложения на каждом сервере продолжат обновлять данные? В этом случае ни один из серверов нельзя взять в качестве основы для клонирования для синхронизации, так как на каждом есть изменения, отсутствующие на другом. Выпутаться из такой ситуации крайне сложно.

Отчасти эти проблемы можно обойти, если тщательно продумать схему секционирования данных и распределение привилегий[1]. Но это трудно и обычно существуют другие способы достичь желаемого результата.

В общем случае разрешение записи на обоих серверах создает больше проблем, чем решает. Однако в режиме активный–пассивный эта конфигурация может быть весьма полезной, что мы и продемонстрируем в следующем разделе.

Главный–главный в режиме активный–пассивный

У репликации типа главный–главный есть вариант, который позволяет обойти все подводные камни, рассмотренные выше. Более того, это дей-

[1] Но только отчасти – мы можем взять на себя роль адвоката дьявола и указать на изъяны почти в любой мыслимой конфигурации.

ственный способ конструирования отказоустойчивых и высокодоступных систем. Основное отличие состоит в том, что один из серверов «пассивен», то есть может только читать данные (рис. 8.7).

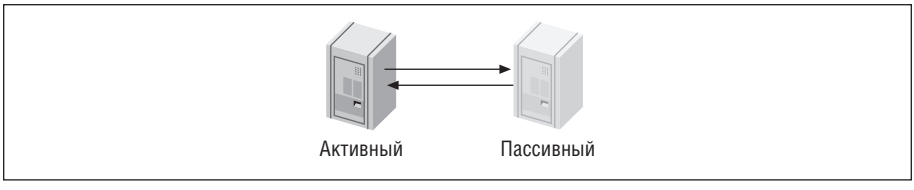


Рис. 8.7. *Репликация главный–главный в режиме активный–пассивный*

Такая схема позволяет без труда менять активный и пассивный серверы ролями, поскольку конфигурации серверов симметричны. А это, в свою очередь, дает возможность без проблем аварийно переключиться на резервный (failover) и вернуться на основной (failback) сервер. Кроме того, вы можете выполнять обслуживание базы, оптимизировать таблицы, переходить на новую версию операционной системы (приложения, оборудования) и решать другие задачи, не останавливая работу.

Например, команда `ALTER TABLE` блокирует таблицу полностью, запрещая для нее чтение и запись. Это может занять много времени, в течение которого обслуживание будет прервано. Однако имея конфигурацию с несколькими главными серверами, вы можете остановить потоки репликации на активном сервере, так что он не будет обрабатывать обновления от пассивного сервера, изменить на пассивном сервере таблицу, поменять сервера ролями и запустить потоки репликации на сервере, который раньше был активным[1]. Теперь этот сервер прочтет свой журнал ретрансляции и выполнит ту же самую команду `ALTER TABLE`. Это, конечно, займет столь же много времени, но какая разница, если теперь серверу не нужно обслуживать запросы?

Конфигурация главный–главный в режиме активный–пассивный позволяет обойти и многие другие проблемы и ограничения MySQL. Для настройки и управления такой системой имеется специальная утилита MySQL Master-Master Replication Manager (*http://code.google.com/p/mysql-master-master/*). Она автоматизирует различные нетривиальные задачи, например восстановление и ресинхронизацию репликации после ошибок, добавление новых подчиненных серверов и т. д.

Разберемся, как конфигурируется пара главный–главный. Описанные ниже действия нужно выполнить на *обоих* серверах, чтобы итоговые конфигурации были симметричны.

---

[1] Можно не останавливать репликацию, а на время отключить запись в двоичный журнал командой `SET SQL_LOG_BIN=0`. Кроме того, некоторые команды, к примеру, `OPTIMIZE TABLE`, поддерживают режимы `LOCAL` или `NO_WRITE_TO_BINLOG`, подавляющие запись в двоичный журнал.

1. Включить запись в двоичный журнал, назначить серверам уникальные идентификаторы и добавить учетные записи для репликации.
2. Включить режим журналирования обновлений подчиненного сервера. Как мы вскоре увидим, это очень важно для аварийного переключения на резервный сервер и обратно.
3. Необязательно – сконфигурировать пассивный сервер в режиме чтения во избежание изменений, которые могли бы конфликтовать с изменениями на активном сервере.
4. Убедиться, что на обоих серверах находятся в точности одинаковые данные.
5. Запустить MySQL на обоих серверах.
6. Сконфигурировать каждый сервер, так чтобы он был подчиненным для другого, начав с пустого двоичного журнала.

Теперь проследим за тем, что происходит, когда на активном сервере производится какое-либо изменение. Оно записывается в его двоичный журнал и реплицируется в журнал ретрансляции пассивного сервера. Пассивный сервер выполняет запрос и записывает событие в собственный двоичный журнал, поскольку режим `log_slave_updates` включен. Затем активный сервер извлекает то же самое изменение в свой журнал ретрансляции, но игнорирует его, так как идентификатор сервера совпадает с его собственным.

Подробнее о том, как поменять роли, см. раздел «Смена главного сервера» на стр. 474.

В некотором смысле топологию главный–главный в режиме активный–пассивный можно считать способом горячего резервирования с тем, однако, отличием, что «резервный» сервер иногда используется для повышения производительности. С него можно читать данные, выполнять на нем резервное копирование, обслуживание в автономном режиме, устанавливать новые версии программного обеспечения и т. д. Ничего этого на настоящем сервере «горячего» резерва делать нельзя. Однако такая конфигурация не дает возможности повысить производительность записи по сравнению с одиночным сервером (мы еще скажем несколько слов поэтом поводу ниже).

По мере рассмотрения других сценариев и способов применения репликации мы будем возвращаться к этой конфигурации. Она широко распространена и очень полезна.

Главный–главный с подчиненными

Рассмотренную только что конфигурацию можно обобщить, добавив к каждому главному серверу один или несколько подчиненных, как показано на рис. 8.8.

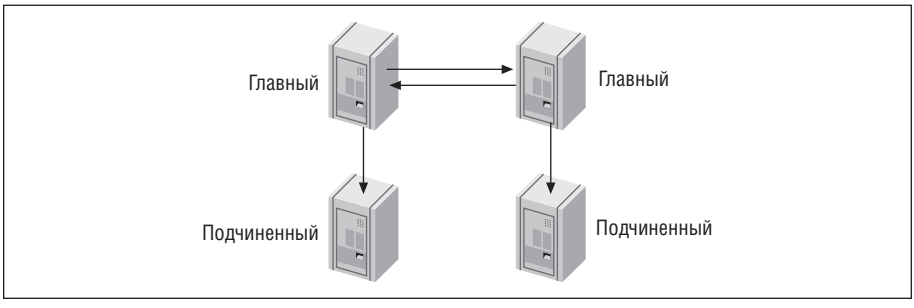


Рис. 8.8. *Репликация главный–главный с подчиненными серверами*

Достоинством такой конфигурации является дополнительная избыточность. В случае репликации между географически удаленными центрами она позволяет устранить единственную точку отказа в каждом центре. Кроме того, как обычно, на подчиненных серверах можно выполнять запросы, читающие много данных.

Но даже если вы собираетесь применять топологию главный–главный только для переключения на резервный сервер при отказе, эта конфигурация все равно полезна. Один из подчиненных серверов может взять на себя роль отказавшего главного, хотя реализовать это несколько труднее. Можно также переподчинить один из серверов другому главному. Впрочем, нужно принимать во внимание повышенную сложность.

Кольцо

Конфигурация с двумя главными серверами – на самом деле, лишь частный случай[1] топологии «кольцо», показанной на рис. 8.9. В кольце есть три или более главных серверов. Каждый сервер выступает в роли подчиненного для предшествующего ему сервера и в роли главного – для последующего. Такая топология называется также *круговой репликацией* (*circular replication*).

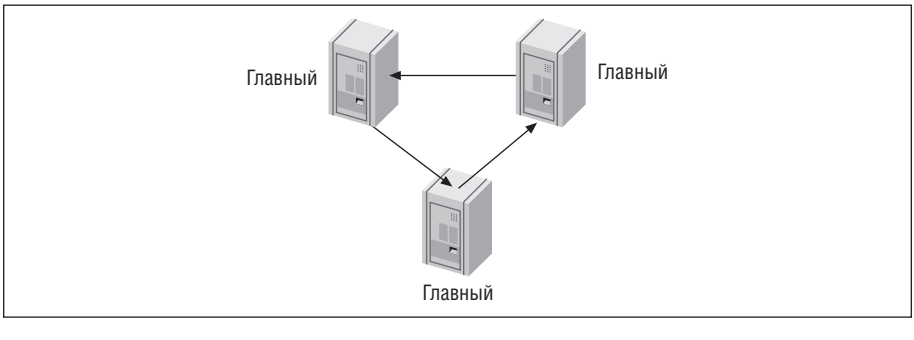


Рис. 8.9. *Топология репликации «кольцо»*

[1] Чуть более осмысленный частный случай, могли бы мы добавить.

Кольцо не обладает некоторыми существенными достоинствами топологии главный–главный, например симметричностью конфигурации и простотой аварийного переключения. Кроме того, для работы такой системы необходимо, чтобы все входящие в кольцо сервера были доступны, а это существенно увеличивает вероятность отказа всей системы. Если удалить из кольца один узел, то события, порожденные этим узлом, могут курсировать по кольцу бесконечно, так как отфильтровать событие может лишь создавший его сервер. Таким образом, кольца по природе своей хрупки и лучше к ним не прибегать.

В какой-то мере снизить риски, присущие круговой репликации, можно, добавив в каждом узле подчиненные серверы, как показано на рис. 8.10. Но это лишь страховка на случай отказа сервера. Выключение питания или иные проблемы, приводящие к пропаданию соединения между узлами, все равно разрушают кольцо.

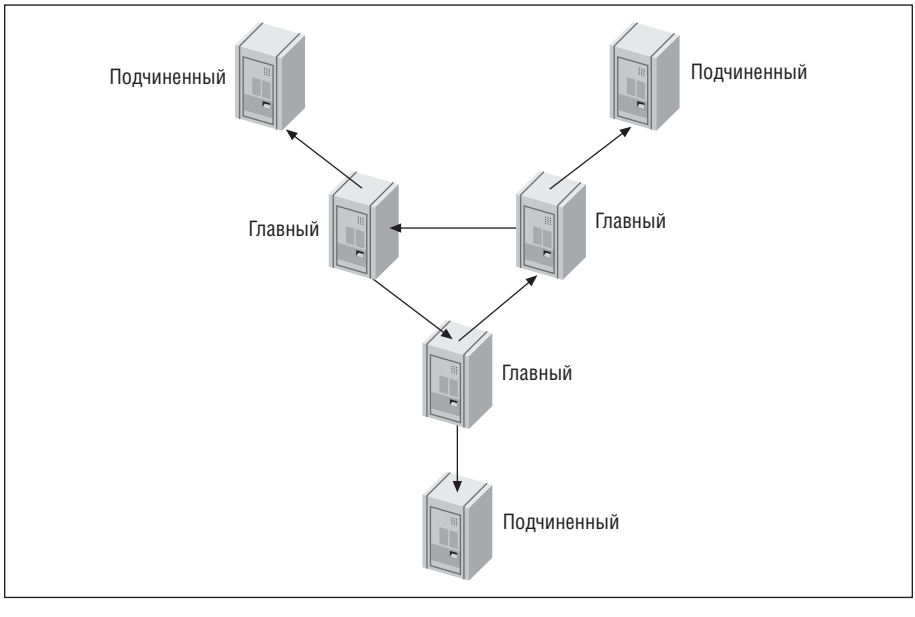


Рис. 8.10. *Круговая репликация с подчиненными серверами в каждом узле*

Главный, главный–распространитель и подчиненные

Мы уже отмечали, что при наличии большого количества подчиненных серверов нагрузка на главный может оказаться чрезмерной. Каждый подчиненный сервер создает на главном отдельный поток, который выполняет специальную команду *binlog dump*. Эта команда читает данные из двоичного журнала и посылает их подчиненному серверу. Для каждого подчиненного потока работа дублируется; никакого совместного использования ресурсов, необходимых команде дампа, не предусмотрено.

Если подчиненных серверов много и в двоичном журнале встретится какое-то особо громоздкое событие, например `LOAD DATA INFILE` для очень длинного файла, то нагрузка на главный сервер может резко возрасти. У главного сервера может даже закончиться память, поскольку все подчиненные запрашивают это огромное событие одновременно. Итог – аварийное завершение. С другой стороны, если все подчиненные сервера извлекают из двоичного журнала *разные* события, которые уже вытеснены из кэша файловой системы, то возрастает количество операций поиска на диске, что тоже негативно влияет на производительность главного сервера.

По этой причине, если необходимо много подчиненных серверов, то часто имеет смысл разгрузить главный сервер и ввести в схему так называемый *главный сервер-распространитель* (*distribution master*). Он выступает в роли подчиненного и выполняет одну-единственную функцию – читать двоичные журналы с главного сервера и передавать их далее. К серверу-распространителю можно подключить много подчиненных, сняв тем самым нагрузку с основного сервера. А чтобы не тратить ресурсы распространителя на выполнение запросов, нужно задать на нем для всех таблиц подсистему хранения Blackhole, как показано на рис. 8.11.

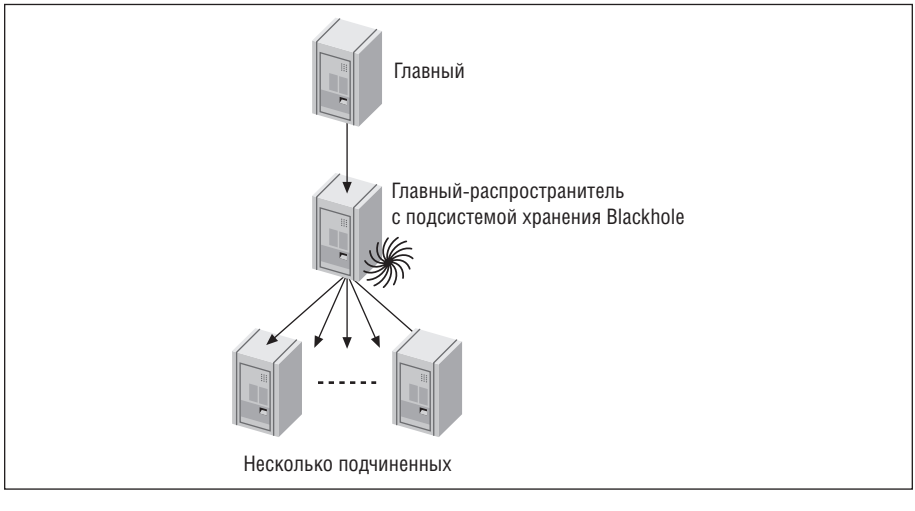


Рис. 8.11. *Главный, главный-распространитель и несколько подчиненных*

Трудно определенно сказать, сколько подчиненных серверов может обслужить главный без распространителя. В качестве ориентира можно считать, что если пропускная способность главного сервера используется полностью, то присоединять к нему стоит не больше 10 подчиненных. Если количество операций записи очень мало или реплицируется лишь несколько таблиц, то главный сервер, наверное, сможет обслужить и больше подчиненных. Кроме того, необязательно ограничивать-

ся только одним распространителем. Если количество подчиненных серверов очень велико, то можно организовать и несколько распространителей и даже создать из них пирамиду.

Серверы-распространители можно задействовать и для других целей, например задав на них правила фильтрации и перезаписи событий в двоичном журнале. Это гораздо эффективнее, чем повторять запись в журнал, перезапись и фильтрацию на каждом подчиненном сервере.

Если на сервере-распространителе используется подсистема хранения Blackhole, то количество обслуживаемых им подчиненных серверов можно увеличить. Распространитель, конечно, обязан выполнять запросы, но это обходится очень дешево, так как в таблицах типа Blackhole никакие данные не хранятся.

Часто задают вопрос: как гарантировать, что все таблицы на сервере-распространителе имеют тип Blackhole? Что если кто-то создаст на главном сервере новую таблицу, указав другую подсистему хранения? Вообще, эта проблема возникает всякий раз, как подсистемы хранения на главном и подчиненном сервере желательно сделать разными. Обычно для ее решения на сервере задают следующий параметр:

```
storage_engine = blackhole
```

Он распространяется только на команды `CREATE TABLE`, в которых подсистема хранения не указана явно. Если вы пользуетесь существующим неподконтрольным вам приложением, то такая топология может оказаться хрупкой. Можно с помощью параметра `skip_innodb` запретить использование InnoDB, тем самым заменив подсистему хранения на MyISAM, но отключить подсистемы MyISAM и Memory не получится.

Еще один существенный недостаток – сложность подмены главного сервера одним из настоящих подчиненных. Сделать подчиненный сервер главным трудно, так как из-за наличия промежуточного сервера координаты в двоичном журнале почти наверняка будут не такими, как на исходном главном сервере.

Дерево или пирамида

Если главный сервер реплицируется на очень много подчиненных – неважно, с целью распределения данных по географическому признаку или просто для повышения пропускной способности чтения, – может оказаться удобной пирамидальная топология, изображенная на рис. 8.12.

Преимущество такой конфигурации состоит в том, что разгружается главный сервер, – как и в рассмотренном выше случае применения распределителя. Недостаток же в том, что отказ на промежуточном уровне распространяется сразу на несколько серверов, чего не произошло бы, будь все они подключены к главному напрямую. К тому же, чем больше промежуточных уровней, тем сложнее восстановление после сбоев.

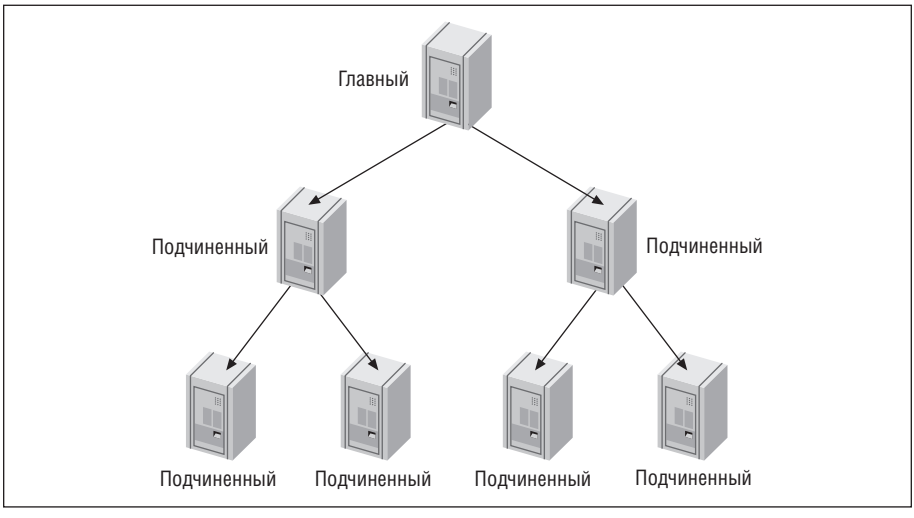


Рис. 8.12. *Топология репликации «пирамида»*

Специальные схемы репликации

Механизм репликации в MySQL достаточно гибок для того, чтобы можно было создавать специальные схемы под нужды конкретного приложения. Как правило, применяется комбинация фильтрации, распространения и репликации с использованием различных подсистем хранения. Можно также прибегать к различным «трюкам», например задавать для таблиц на исходном или конечном сервере подсистему хранения Blackhole (как описано в разделе «Главный, главный-распространитель и подчиненные» на стр. 457). Схема может быть сколь угодно изощренной. Надо лишь стараться, чтобы сложность мониторинга и администрирования оставалась в разумных пределах, и учитывать имеющиеся ресурсы (пропускную способность сети, мощность процессора и т. д.).

Избирательная репликация

Чтобы в полной мере воспользоваться локальностью ссылок и уместить рабочее множество для запросов на чтение в памяти, можно реплицировать небольшие порции данных на множество подчиненных серверов. Если на каждом подчиненном сервере находится лишь малая часть информации, то, разослав запросы на чтение всем таким серверам, можно добиться гораздо более эффективного использования памяти на каждом подчиненном. Кроме того, на каждый подчиненный сервер будет приходиться лишь часть общей нагрузки, порождаемой операциями записи на главном сервере, так что главный сервер можно будет сделать более мощным, не опасаясь, что подчиненные отстанут.

Эта схема напоминает горизонтальное секционирование данных, о котором мы будем подробно говорить в следующей главе, но ее достоинство в том, что все же имеется один сервер – главный, – на котором хранятся все данные. Это означает, что любая операция записи производится только на одном сервере, а если для некоторой операции чтения не существует единственного подчиненного сервера, содержащего все необходимые данные, то всегда можно выполнить ее на главном сервере. Даже если все операции чтения невозможно выполнить на подчиненных серверах, то за счет переноса большей их части все-таки можно разгрузить главный.

Простейший способ реализовать эту схему заключается в том, чтобы поместить информацию в разные базы на главном сервере, а потом реплицировать каждую базу на разные подчиненные сервера. Например, можно разнести по подчиненным серверам данные, относящиеся к различным подразделениям компании: `sales`, `marketing`, `procurement` и т. д. Тогда на каждом подчиненном сервере нужно будет задать конфигурационный параметр `replicate_wild_do_table`, который ограничивает воспринимаемые им данные одной базой. Вот как это может выглядеть для базы данных `sales`:

```
replicate_wild_do_table = sales.%
```

Полезна также фильтрация на сервере-распространителе. Например, если требуется реплицировать какую-то часть сильно загруженного сервера по медленной или очень дорогой сети, то можно организовать локальный распространитель с таблицами типа Blackhole и правилами фильтрации. Фильтры на этом сервере будут удалять ненужные записи из журналов. Это позволит одновременно избежать задания небезопасных параметров протоколирования на главном сервере и передачи всех журналов по сети удаленным подчиненным.

Разграничение функций

Многие приложения представляют собой смесь оперативной обработки транзакций (OLTP) и оперативной аналитической обработки (OLAP). OLTP-запросы обычно бывают короткими и транзакционными. Напротив, OLAP-запросы большие, выполняются медленно, зато не требуют стопроцентно актуальных данных. Запросы разных видов предъявляют к серверу различные требования. Поэтому лучше выполнять их на разных серверах с разными конфигурациями и, возможно, даже разными подсистемами хранения и оборудованием.

Часто эту проблему решают путем репликации данных с OLTP-сервера на подчиненные серверы, специально настроенные для работы в условиях OLAP-нагрузки. На них может быть установлено другое оборудование, построены другие индексы и применены иные подсистемы хранения. Если для OLAP-запросов выделяется отдельный подчиненный сервер, то, возможно, придется смириться с более значительным отста-

ванием репликации или каким-то другим снижением качества обслуживания на этом сервере. Зато этот сервер можно использовать для задач, которые на невыделенном подчиненном сервере работали бы недопустимо долго, например для выполнения очень долгих запросов.

Никакой специальной схемы репликации для этого придумывать не надо, хотя можно не реплицировать часть данных с главного сервера, если это позволит добиться ощутимой экономии на подчиненном сервере. Напомним, что игнорирование даже небольшой части данных за счет фильтров репликации на журнале ретрансляции может заметно снизить объем ввода/вывода и активность кэша.

Архивирование данных

На подчиненном сервере можно хранить архив данных, удаленных с главного. Для этого нужно выполнить на главном сервере команду удаления так, чтобы она не была повторена на подчиненном. Это можно сделать двумя способами: выборочно отключить запись в двоичный журнал на главном сервере или сконфигурировать правила `replicate_ignore_db` на подчиненном.

В первом случае нужно сначала выполнить команду `SET SQL_LOG_BIN=0` в процессе, который уничтожает данные на главном сервере, а уже потом приступать к удалению. Плюс заключается в том, что не требуется как-то специально конфигурировать репликацию на подчиненном сервере, а, поскольку команды удаления даже не записываются в двоичный журнал, то и за счет этого эффективность хоть немного да повышается. Основной же недостаток кроется в том, что двоичный журнал главного сервера невозможно использовать для аудита или восстановления на конкретный момент в прошлом, поскольку часть изменений, произведенных на сервере, в нем отсутствует. Кроме того, этот способ требует привилегии `SUPER`.

При использовании второго подхода нужно выполнить на главном сервере команду `USE`, то есть сделать текущей какую-то базу данных перед тем, как начинать удаление. Например, можно создать специальную базу с именем `purge`, указать в файле *my.cnf* на подчиненном сервере параметр `replicate_ignore_db=purge` и потом перезапустить его. В результате подчиненный сервер будет игнорировать все команды, выполненные в контексте текущей базы `purge`. У этого способа нет минусов, присущих предыдущему, но есть другой (не очень серьезный) недостаток: подчиненный сервер вынужден выбирать из двоичного журнала все события, даже те, в которых не заинтересован. Кроме того, есть опасность, что кто-то по ошибке выполнит в контексте базы данных `purge` запрос, не связанный с удалением архивных данных, и тогда этот запрос не будет воспроизведен на подчиненном сервере.

Программа *mk-archiver*, входящая в комплект инструментов Maatkit, поддерживает оба метода.

Есть и третий способ — воспользоваться параметром `binlog_ignore_db` для фильтрации событий репликации, но выше мы отмечали, что считаем этот вариант слишком опасным.

Использование подчиненных серверов для полнотекстового поиска

Во многих приложениях необходимо сочетание транзакций с полнотекстовым поиском. Однако возможность полнотекстового поиска существует лишь для таблиц типа MyISAM, а они не поддерживают транзакций. Типичное обходное решение заключается в том, чтобы сконфигурировать для полнотекстового поиска подчиненный сервер, поменяв на нем для некоторых таблиц подсистему хранения. Затем на подчиненном сервере можно построить полнотекстовые индексы. Тем самым удается избежать потенциальных проблем, связанных с использованием в одном запросе транзакционных и нетранзакционных подсистем хранения, а заодно снять с главного сервера заботу о поддержании полнотекстовых индексов.

Подчиненные серверы только для чтения

Во многих организациях предпочитают использовать подчиненные серверы только для чтения, чтобы непреднамеренные изменения не нарушили репликацию. Этого можно добиться с помощью параметра `read_only`, который запрещает большинство операций записи; исключение составляют только потоки репликации, пользователи с привилегий `SUPER` и запись во временные таблицы. Эта тактика отлично работает при условии, что обычным пользователям не выдается привилегия `SUPER`, чего в любом случае не следует делать.

Эмуляция репликации с несколькими главными серверами

В настоящее время MySQL не поддерживает репликацию с несколькими главными серверами (то есть, наличие нескольких главных серверов для одного подчиненного). Однако такую топологию можно эмулировать, поочередно изменяя главный сервер в распределенной системе. Например, сначала вы назначаете в качестве главного сервер A, некоторое время работаете в такой конфигурации, потом делаете главным сервер B, снова какое-то время работаете, после чего опять возвращаетесь к серверу A. Насколько хороша такая схема, зависит от данных и от того, какой объем работы возлагают на подчиненный сервер оба главных. Если главные серверы не слишком загружены и выполняемые ими обновления не конфликтуют между собой, то все будет работать нормально.

Но придется позаботиться об отслеживании координат репликации обоих главных серверов. Кроме того, желательно, чтобы поток ввода/

вывода на подчиненном сервере не извлекал больше данных, чем вы намереваетесь обрабатывать в каждом цикле, иначе можно значительно увеличить сетевой трафик, загружая и тут же выбрасывая большой объем информации на каждой итерации цикла.

Готовый сценарий для этих целей можно найти по адресу *http://code.google.com/p/mysql-mmre/*.

Репликацию с несколькими главными серверами можно эмулировать также с помощью топологии главный-главный (или кольцо) с использованием подсистемы хранения Blackhole на подчиненном сервере, как показано на рис. 8.13.

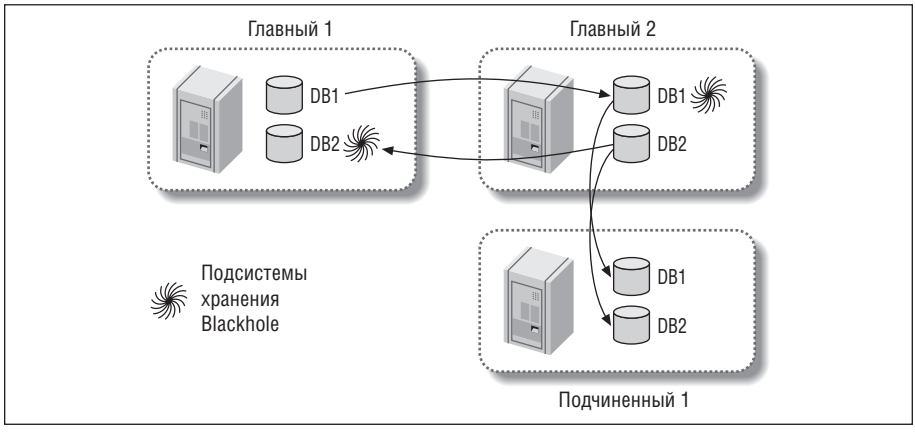


***Рис. 8.13.** Эмуляция репликации с несколькими главными серверами с применением двух главных серверов и подсистемы хранения Blackhole*

В этой конфигурации оба главных сервера содержат свои собственные данные. Дополнительно на каждом созданы такие же таблицы, как на другом главном, но, поскольку они имеют тип Blackhole, то реальные данные в них не хранятся. Подчиненный сервер подключен к одному из главных – не важно, к какому именно. Но на подчиненном сервере таблиц типа Blackhole нет, поэтому фактически он подчинен обоим главным.

На самом деле, для достижения желаемого эффекта топология главный-главный даже не нужна. Можно просто реплицировать с `server1` на `server2` и далее на подчиненный. Если на `server2` для таблиц, реплицированных с `server1`, определена подсистема хранения Blackhole, то он не будет хранить данные, поступающие с `server1`. Такая схема изображена на рис. 8.14.

Каждой из этих конфигураций свойственны общие проблемы, например конфликтующие обновления и команды `CREATE TABLE` с явно заданной подсистемой хранения.

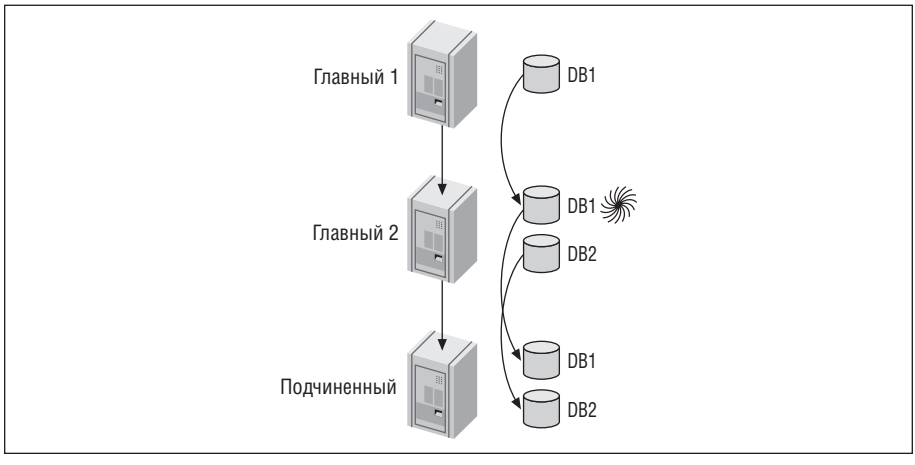


Рис. 8.14. *Еще один способ эмулировать репликацию с несколькими главными серверами*

Создание сервера журналов

Среди прочего механизм репликации в MySQL позволяет создать «сервер журналов», единственное назначение которого – упростить повтор и/или фильтрацию событий двоичного журнала (никаких данных там не хранится). Ниже в этой главе мы увидим, что это весьма полезно для возобновления репликации после сбоя, а также для восстановления данных на конкретный момент времени в прошлом (см. главу 11).

Представим, что имеется набор двоичных журналов или журналов ретрансляции – быть может, из резервной копии, а, быть может, с «упавшего» сервера, – и вы хотите воспроизвести хранящиеся в них события. Можно было бы извлечь события с помощью утилиты *mysqlbinlog*, но удобнее и более эффективно просто настроить экземпляр MySQL вообще без данных и заставить его поверить, что это его собственные журналы. Чтобы создать временный сервер журналов, который потом будет уничтожен, можно воспользоваться сценарием MySQL Sandbox по адресу *http://sourceforge.net/projects/mysql-sandbox/*. Серверу журналов *не нужны никакие данные*, потому что он будет не исполнять журналы, а только транслировать их другим серверам (однако учетная запись для репликации должна существовать).

Давайте посмотрим, как работает эта техника (примеры ее применения мы приведем ниже). Предположим, что журналы называются *somelog-bin.000001*, *somelog-bin.000002* и т. д. Поместим эти файлы в каталог двоичных журналов на сервере журналов. Пусть это будет */var/log/mysql*. Затем перед запуском сервера журналов изменим файл *my.cnf* следующим образом:

```
log_bin = /var/log/mysql/somelog-bin
log_bin_index = /var/log/mysql/somelog-bin.index
```

Автоматически сервер не найдет файлы журналов, поэтому нужно также изменить индексный файл журналов сервера. В UNIX-системах это делается с помощью такой команды[1]:

```
# /bin/ls -1 /var/log/mysql/somelog-bin.[0-9]* > /var/log/mysql/somelog-bin.index
```

Убедитесь, что пользователь, от имени которого работает MySQL, может читать и писать в индексный файл. Теперь можно запустить сервер и с помощью команды `SHOW MASTER LOGS` проверить, что он видит файлы журналов.

Почему сервер журналов предпочтительнее утилиты *mysqlbinlog* для восстановления? По нескольким причинам:

- Он быстрее, так как нет необходимости извлекать команды из журнала и по конвейеру передавать их процессу *mysql*
- Легко наблюдать за ходом работы
- Легко обходить ошибки. Например, можно пропускать команды, которые не удается реплицировать
- Легко отфильтровывать события репликации
- Иногда *mysqlbinlog* не сможет прочитать двоичный журнал из-за изменившегося формата

Репликация и планирование пропускной способности

Узким местом репликации являются операции записи, причем масштабированию они поддаются плохо. Планируя, какую долю общей пропускной способности системы отнимет добавление подчиненных серверов, нужно тщательно производить расчеты. Когда дело касается репликации, очень легко просчитаться.

Пусть, например, рабочая нагрузка состоит на 20% из операций записи и на 80% из операций чтения. Чтобы облегчить расчет, примем следующие, очень приблизительные, упрощающие предположения:

- Трудоемкость запросов на чтение и на запись одинакова
- Все серверы абсолютно идентичны между собой и способны обработать ровно 1000 запросов в секунду
- Характеристики производительности главных и подчиненных серверов одинаковы

[1] Мы употребляем полное имя */bin/ls*, чтобы не вызывалась программа с тем же именем, которая добавляет управляющие символы для цветового выделения на экране.

- Все запросы на чтение можно переместить на подчиненные серверы

Если в данный момент имеется один сервер, обрабатывающий 1000 запросов в секунду, то сколько подчиненных серверов придется добавить, чтобы можно было удвоить нагрузку и переместить на подчиненные сервера все запросы на чтение?

На первый взгляд кажется, что можно добавить два подчиненных сервера и распределить между ними 1600 запросов на чтение. Однако не будем забывать, что к рабочей нагрузке добавилось 400 запросов на запись в секунду и их-то распределить между главным и подчиненными серверами не получится. Каждый подчиненный сервер должен выполнить 400 операций записи в секунду. Это означает, что каждый такой сервер на 40% занят записью и может обслужить лишь 600 запросов на чтение. Следовательно, для удвоения трафика нужно добавить не два, а *три* подчиненных сервера.

А что если трафик снова удвоится? Теперь каждую секунду производится 800 операций записи, так что главный сервер пока справляется. Но каждый из подчиненных занят записью на 80%, поэтому для обработки 3200 операций чтения потребуется уже 16 подчиненных серверов. И если трафик еще чуть-чуть возрастет, то главный сервер уже может не справиться.

Как видите, масштабируемость далека от линейной: при увеличении количества запросов в 4 раза количество серверов возрастает в 17 раз. Это показывает, как быстро уменьшается отдача при добавлении серверов, подчиненных одному главному. И это еще при наших нереалистичных предположениях, в которых не учитывается, например, тот факт, что при однопоточной покомандной репликации подчиненные серверы обычно имеют меньшую пропускную способность, чем главный. Поэтому в реальных условиях репликация, скорее всего, будет вести себя даже хуже, чем в теории.

Почему репликация не помогает масштабированию записи

Фундаментальная проблема, из-за которой отношение количества серверов к пропускной способности так быстро возрастает, состоит в том, что операции записи нельзя распределить между несколькими машинами так же равномерно, как операции чтения. Другими словами, репликация пригодна для масштабирования чтения, но не записи.

Возникает вопрос, можно ли как-то применить репликацию с целью повышения пропускной способности записи? Ответ: нет, никак нельзя. Единственный способ масштабирования записи – секционирование данных, и мы обсудим его в следующей главе.

Возможно, у некоторых читателей возникла мысль использовать топологию главный–главный (см. раздел «Главный–главный в режиме ак-

тивный–активный» на стр. 453) и писать на оба главных сервера. При такой конфигурации можно слегка увеличить количество операций записи по сравнению с топологией главный-подчиненный, так как накладные расходы на сериализацию делятся пополам между обоими серверами. Если на каждом сервере выполняется 50% операций записи, то сериализовать придется только те 50%, которые обусловлены репликацией другого сервера. Теоретически это лучше, чем выполнять все 100% записи параллельно на одной машине (главном сервере) и 100% записи последовательно на другой машине (подчиненном сервере).

Идея может показаться заманчивой. Однако такая конфигурация не в состоянии обработать столько же операций записи, сколько один сервер. Сервер, для которого сериализуются 50% операций записи, работает медленнее, чем сервер, выполняющий все операции записи параллельно.

Поэтому такая тактика не обеспечивает масштабирования записи. Это лишь способ разделить неэффективные сериализованные операции на оба сервера, так что «самое слабое звено» оказывается не таким уж слабым. Мы получаем лишь сравнительно небольшое улучшение по сравнению с конфигурацией активный–пассивный, увеличивая риски и почти ничего не получая взамен. Более того, как будет показано в следующем разделе, в общем случае отсутствует даже минимальный выигрыш.

Запланированная недогрузка

Намеренная недогрузка серверов может оказаться наиболее правильным и экономичным способом построения крупномасштабных приложений, особенно если применяется репликация. Сервер, располагающий резервной пропускной способностью, лучше выдерживает всплески нагрузки, обладает ресурсами для выполнения медленных запросов и операций обслуживания (например, команд `OPTIMIZE TABLE`) и лучше справляется с репликацией.

Попытка немного уменьшить издержки репликации за счет записи на оба узла в топологии главный–главный, как правило, оказывается иллюзорной экономией. Обычно каждый сервер в паре нельзя загружать чтением более чем на 50%, так как иначе пропускной способности не хватит в случае сбоя одного из них. Если оба сервера способны справиться с нагрузкой самостоятельно, то, наверное, вообще нет смысла беспокоиться об издержках однопоточной репликации.

Кроме того, резервирование избыточной пропускной способности – один из лучших способов обеспечить высокую доступность, хотя имеются и другие, например выполнение приложения в «неполноценном» режиме в случае отказа. В следующей главе эта тема будет рассмотрена более подробно.

Администрирование и обслуживание репликации

Вряд ли вам придется заниматься настройкой репликации постоянно, если только количество серверов не слишком велико. Но, коль скоро репликация включена, то ее мониторинг и администрирование становятся регулярной задачей, сколько бы серверов ни было задействовано.

Эту работу следует максимально автоматизировать. Но писать собственные приложения для этой цели, возможно, и не понадобится; в главе 14 мы рассмотрим несколько инструментов повышения продуктивности для MySQL, и во многие из них средства мониторинга репликации уже встроены или реализованы в виде подключаемых модулей. Наиболее богатую функциональность предлагают программы Nagios, MySQL Enterprise Monitor и MonYOG.

Мониторинг репликации

При использовании репликации сложность мониторинга MySQL возрастает. Хотя она производится как на главном, так и на подчиненном сервере, большую часть работы выполняет последний, и именно здесь чаще возникают проблемы. Все ли подчиненные серверы реплицируют данные? Были ли на каком-нибудь из них ошибки? Насколько отстал самый медленный из подчиненных серверов? MySQL предоставляет большую часть информации, необходимой для ответа на эти вопросы, но автоматизация процедуры мониторинга и обеспечение надежной репликации возлагается на вас.

На главном сервере можно воспользоваться командой `SHOW MASTER STATUS`, которая покажет текущую позицию в двоичном журнале главного сервера и его конфигурацию (см. раздел «Конфигурирование главного и подчиненного сервера» на стр. 433). Можно также узнать, какие двоичные журналы есть на диске:

```
mysql> SHOW MASTER LOGS;
+------------------+-----------+
| Log_name         | File_size |
+------------------+-----------+
| mysql-bin.000220 |    425605 |
| mysql-bin.000221 |   1134128 |
| mysql-bin.000222 |     13653 |
| mysql-bin.000223 |     13634 |
+------------------+-----------+
```

Эти сведения востребованы при определении параметров команды `PURGE MASTER LOGS`. Для просмотра событий репликации в двоичном журнале полезна команда `SHOW BINLOG EVENTS`. Например, после выполнения предыдущей директивы мы создали таблицу на неиспользуемом сервере. Поскольку нам точно известно, что это единственная команда, из-

меняющая данные, то мы знаем, что ее смещение в двоичном журнале равно 13634, поэтому можно посмотреть, что туда было записано:

```
mysql> SHOW BINLOG EVENTS IN 'mysql-bin.000223' FROM 13634\G
*************************** 1. row ***************************
Log_name: mysql-bin.000223
Pos: 13634
Event_type: Query
Server_id: 1
End_log_pos: 13723
Info: use `test`; CREATE TABLE test.t(a int)
```

Измерение отставания подчиненного сервера

Очень часто возникает необходимость следить за тем, насколько главный сервер опережает подчиненный. Хотя столбец `Seconds_behind_master` в таблице, которую печатает команда `SHOW SLAVE STATUS`, теоретически показывает отставание подчиненного сервера, фактически он не всегда точен, по разным причинам:

- Подчиненный сервер вычисляет `Seconds_behind_master`, сравнивая текущую временную метку (timestamp) сервера с временной меткой, записанной в событии двоичного журнала, поэтому подчиненный сервер не может даже сообщить о своем отставании, если не обрабатывает в данный момент какой-нибудь запрос.
- Обычно подчиненный сервер возвращает `NULL`, если на нем не работают процессы репликации.
- Некоторые ошибки (например, несоответствие конфигурационных параметров `max_allowed_packet` на главном и подчиненном сервере или неустойчивая сеть) могут нарушить репликацию и/или остановить потоки репликации на подчиненном сервере, но в столбце `Seconds_behind_master` при этом будет 0, а не сообщение об ошибке.
- Иногда подчиненный сервер не в состоянии вычислить свое отставание, хотя процессы репликации работают. В этом случае он может вывести как 0, так и `NULL`.
- Очень длительная транзакция может вызвать флуктуации отображаемого отставания. Например, если транзакция, обновляющая данные, остается открытой в течение часа, а затем фиксируется, то запись об обновлении попадет в двоичный журнал спустя час после того, как оно фактически состоялось. Когда подчиненный сервер приступит к обработке этой команды, он сообщит, что отставание от главного составляет час, а сразу вслед за этим – что никакого отставания нет.
- Если главный сервер-распространитель отстает, а ему подчинены какие-то сервера, то эти сервера будут говорить, что отставание составляет 0 секунд, если успевают за распространителем, хотя отставание от настоящего главного сервера ненулевое.

Все эти проблемы можно решить, если игнорировать величину `Seconds_behind_master` и судить об отставании подчиненного сервера с помощью чего-то, что можно наблюдать и измерять непосредственно. Неплохое решение дает *heartbeat record* (сердцебиение), которое представляет собой временную метку, обновляемую на главном сервере один раз в секунду. Чтобы вычислить отставание, достаточно вычесть heartbeat из текущей временной метки на подчиненном сервере. Этот метод лишен всех вышеупомянутых недостатков и обладает дополнительным преимуществом: наличие удобной временной метки, показывающей, на какой момент времени актуальны данные на подчиненном сервере. Сценарий *mk-heartbeat*, входящий в состав комплекта инструментов Maatkit, – одна из реализаций репликации heartbeat.

Ни одна из рассмотренных метрик отставания не позволяет судить о том, сколько времени понадобится подчиненному серверу, чтобы догнать главный. Это зависит от целого ряда факторов, например, от мощности подчиненного сервера и количества запросов, продолжающих поступать главному.

Как узнать, согласованы ли подчиненные серверы с главным

В идеальном мире подчиненный сервер всегда являлся бы точной копией главного. Но грубая реальность такова, что иногда из-за ошибок репликации данные на подчиненном сервере могут рассинхронизироваться с главным. Даже если явных ошибок нет, рассинхронизация все равно возможна, поскольку некоторые операции MySQL реплицируются неправильно, из-за ошибок в коде MySQL, из-за искажений в сети, сбоев, некорректных остановов и т. д[1].

Опыт показывает, что это правило, а не исключение, а значит, проверка согласованности подчиненных серверов с главными должна стать рутинной задачей. Особенно важно это, когда репликация применяется для резервного копирования, поскольку вряд ли вам захочется снимать резервную копию подчиненного сервера с испортившимися данными.

В первом издании настоящей книги был приведен сценарий, который сравнивал количество строк в таблицах на главном и подчиненном серверах. Конечно, кое-какие различия таким способом выловить можно, но все же счетчик строк нельзя считать строгим доказательством идентичности данных. Нужен какой-то метод, позволяющий сравнивать само содержимое таблиц.

В MySQL нет встроенного способа определить, совпадают ли данные на двух серверах. Зато имеются некоторые заготовки для вычисления контрольной суммы таблиц и данных, например `CHECKSUM TABLE`. Однако

[1] Если вы пользуетесь нетранзакционной подсистемой хранения, то останов сервера без предварительной команды `STOP SLAVE` считается некорректным.

сравнение подчиненного сервера с главным в момент, когда репликация работает, – задача нетривиальная.

В комплекте Maatkit есть инструмент *mk-table-checksum*, который решает эту и ряд других задач. Среди прочего он позволяет выполнять быстрое параллельное сравнение сразу нескольких серверов, но основная его особенность – умение проверять синхронизированность данных на главном и подчиненном серверах. Принцип работы основан на выполнении запросов `INSERT ... SELECT` на главном сервере.

В этих запросах вычисляется контрольная сумма данных, и результаты вставляются в таблицу. Затем указанные команды реплицируются и повторно выполняются на подчиненном сервере. Далее можно сравнить результаты на обоих серверах и узнать, различаются ли данные. Поскольку вся процедура основана на репликации, то для получения правильных результатов не нужно одновременно блокировать таблицы на обоих серверах.

Типичный способ использования этого инструмента состоит в том, чтобы запустить его на главном сервере примерно с такими параметрами:

```
$ mk-table-checksum --replicate=test.checksum
--chunksize 100000 --sleep-coef=2 <master_host>
```

Данная команда вычисляет контрольные суммы всех таблиц, обрабатывая их кусками примерно по 100 000 строк, и вставляет результаты в таблицу `test.checksum`. Между соседними кусками делается пауза продолжительностью в два раза больше того времени, которое потребовалось для вычисления контрольной суммы последнего куска. Смысл этого в том, чтобы запросы не мешали нормальной работе базы данных.

После того как эти запросы были реплицированы на подчиненные серверы, можно выполнить простой запрос, который проверит, есть ли отличия от главного. Сценарий *mk-table-checksum* автоматически находит все серверы, подчиненные данному, запускает на каждом из них запрос и выводит результаты. Следующая команда спускается по иерархии подчиненных серверов на глубину 10, начиная с одного и того же главного сервера, и выводит перечень различающихся таблиц:

```
$ mk-table-checksum --replicate=test.checksum --replcheck 10 <master_host>
```

Компания MySQL AB планирует реализовать аналогичный механизм в самом сервере. Возможно, он будет лучше внешнего сценария, но пока *mk-table-checksum* – единственный инструмент, позволяющий легко и надежно сравнивать данные на главном и подчиненном серверах.

Восстановление синхронизации подчиненного сервера с главным

Очень может быть, что вам придется неоднократно столкнуться с рассинхронизацией подчиненного сервера. Возможно, расхождения выя-

вились в результате подсчета контрольных сумм, а, быть может, вы заранее знаете, что некий запрос был пропущен подчиненным сервером или что кто-то изменил данные непосредственно на нем.

Традиционно для восстановления синхронизации рекомендуют остановить подчиненный сервер и заново клонировать главный. Если несогласованность состояния критична, то, вероятно, стоит вывести подчиненный сервер из промышленной эксплуатации немедленно после обнаружения проблемы. Затем его можно заново клонировать или восстановить из резервной копии.

Недостаток этого подхода – неудобство, особенно когда объем хранимой информации велик. Если вы можете определить, какие данные различаются, то есть более эффективные способы, чем клонирование всего сервера. А если обнаруженная рассогласованность не критична, то можно оставить подчиненный сервер в оперативном режиме и восстановить синхронизацию только затронутых данных.

Простейшая мера – выгрузить и снова загрузить рассогласованные объекты с помощью утилиты *mysqldump*. Этот подход вполне приемлем, если на протяжении данного процесса содержимое баз не изменяется. Тогда достаточно заблокировать таблицу на главном сервере, выгрузить ее, дождаться, пока подчиненный сервер догонит главный, и импортировать ее на подчиненном сервере (подождать нужно для того, чтобы не внести рассогласование в другие таблицы, например обновлявшиеся по результатам соединения с рассинхронизированной).

Хотя во многих случаях такая тактика работает, к серверу, который постоянно занят, она не применима. Есть у нее и другой недостаток: данные на подчиненном сервере изменяются в обход репликации. Обычно безопаснее всего модифицировать информацию на подчиненном сервере посредством репликации (производя изменения на главном), поскольку при этом не возникает неприятных «гонок» (race condition) и других сюрпризов. Кроме того, если объем данных очень велик или пропускная способность сети ограничена, то выгрузка обходится недопустимо дорого. Что если различается только каждая тысячная строка в таблице, содержащей миллион строк? В этой ситуации выгрузка и загрузка всей таблицы – чрезмерное расточительство.

В комплекте Maatkit имеется инструмент *mk-table-sync*, решающий некоторые из описанных выше проблем. Он умеет эффективно находить и устранять различия между таблицами. К тому же его работа основана на механизме репликации, так что синхронизация подчиненного сервера достигается путем выполнения запросов на главном, что позволяет избежать гонок (no race conditioons). Впрочем, он пригоден далеко не всегда; для корректной работы необходимо, чтобы репликация функционировала, поэтому в случае ошибки репликации рассчитывать на этот сценарий не приходится. Инструмент *mk-table-sync* проектировался с учетом эффективности, но с очень большими объемами данных он может не справиться. Сравнение терабайта данных на главном и подчи-

ненном серверах неизбежно создает дополнительную нагрузку на оба. И все-таки в тех случаях, когда сценарий работает, он может сэкономить немало времени и сил.

Смена главного сервера

Рано или поздно наступает момент, когда подчиненному серверу требуется другой главный. Возможно, серверы ротируются для перехода на новую версию или произошел сбой и бывший подчиненный сервер необходимо сделать главным, или просто требуется перераспределить пропускную способность. Какова бы ни была причина, подчиненному серверу нужно сообщить, что с этого момента назначенный ему главный сервер изменился.

Любую процедуру лучше спланировать заранее (во всяком случае, это проще, чем действовать в условиях стресса). Все, что требуется, – это выполнить команду `CHANGE MASTER` на подчиненном сервере, указав правильные параметры. Большинство параметров необязательны; можно задавать лишь те, которые нужно изменить. Подчиненный сервер отбросит текущую конфигурацию и журналы ретрансляции, после чего начнет реплицировать с вновь назначенного главного. Кроме того, новые параметры будут записаны в файл *master.info*, чтобы изменения не потерялись после перезапуска.

Самая трудная часть процедуры – определить позицию в двоичном журнале нового главного сервера, чтобы подчиненный продолжил репликацию с той же логической точки, на которой остановился в момент внесения изменений.

Преобразования подчиненного сервера в главный несколько сложнее. Существует два основных сценария замены главного сервера одним из подчиненных ему. Первый – запланированное «повышение», второй – незапланированное.

Запланированное повышение

Концептуально преобразование подчиненного сервера в главный сделать несложно. Ниже перечислены соответствующие шаги.

1. Прекратить всякую запись на старом главном сервере.
2. Дать возможность подчиненным серверам догнать главный (это необязательно, но упрощает последующие шаги).
3. Сконфигурировать подчиненный сервер, так чтобы он стал новым главным.
4. Сообщить другим подчиненным серверам о новом главном и перенаправить на него операции записи.

Но дьявол, как всегда, кроется в деталях. Существуют различные ситуации, в зависимости от топологии репликации. Так, описанные шаги

выглядят несколько по-разному в конфигурациях главный–главный и главный–подчиненный.

Распишем более подробно действия, которые, по всей вероятности, придется предпринять в большинстве конфигураций.

1. Прекратить всякую запись на текущем главном сервере. Если возможно, следует принудительно остановить все клиентские программы (кроме соединений, связанных с репликацией). Очень неплохо, если в клиентские приложения встроен флаг «не работать», которым вы можете управлять. Если используются виртуальные IP-адреса, то можно попросту отключить виртуальный адрес, а затем прервать все клиентские соединения, чтобы закрыть открытые транзакции.
2. Запретить дальнейшие операции записи на главном сервере, выполнив команду `FLUSH TABLES WITH READ LOCK` (это необязательный шаг). Можно также перевести главный сервер в режим чтения, установив параметр `read_only`. Начиная с этого момента, никакие операции записи на сервере, который вскоре будет заменен, невозможны. Ведь раз он больше не главный, то любая запись на нем означала бы потерю данных!
3. Выбрать подчиненный сервер, которому суждено стать главным, и убедиться, что он полностью синхронизирован (то есть выполнены все события в журналах ретрансляции, полученных от старого главного сервера).
4. Необязательно – проверить, что новый главный сервер содержит в точности те же данные, что и старый.
5. Выполнить на новом главном сервере команду `STOP SLAVE`.
6. Выполнить на новом главном сервере команду `CHANGE MASTER TO MASTER_HOST=''`, а затем `RESET SLAVE`, чтобы он отключился от старого главного и исключил информацию об этом соединении из своего файла *master.info*. Это не будет работать правильно, если информация о соединении задана в файле *my.cnf*, в частности поэтому мы рекомендуем не помещать ее туда.
7. Получить координаты репликации нового главного сервера, выполнив команду `SHOW MASTER STATUS`.
8. Дождаться, пока все прочие подчиненные серверы догонят главный.
9. Остановить старый главный сервер.
10. В версии MySQL 5.1 и более поздних активировать события на новом главном сервере, если это необходимо.
11. Позволить клиентам соединяться с новым главным сервером.
12. Выполнить на каждом подчиненном сервере команду `CHANGE MASTER`, указав на новый главный сервер. Задать координаты репликации полученные на шаге 7.

При повышении подчиненного сервера до уровня главного не забудьте удалить на нем все специфичные для подчиненных серверов базы данных, таблицы и привилегии. Кроме того, необходимо изменить все относящиеся к подчиненным серверам конфигурационные параметры, например, ослабленный режим `innodb_flush_log_at_trx_commit`.

Если главные и подчиненные серверы сконфигурированы одинаково, то менять ничего не нужно.

Незапланированное повышение

Если произошел аварийный отказ главного сервера и его необходимо заменить подчиненным, то процедура может осложниться. В случае, когда вы располагаете всего одним подчиненным сервером, выбирать не из чего. Но если таковых несколько, придется выполнить ряд дополнительных шагов.

Здесь добавляется проблема потенциальной потери событий репликации. Может случиться, что некоторые обновления, произведенные на главном сервере, еще не дошли ни до одного подчиненного. Возможно даже, что команда была выполнена на главном сервере, а затем отменена, но откат еще не реплицирован на подчиненный, – таким образом, подчиненный сервер может опередить главный с точки зрения логической позиции репликации[1]. Если можно восстановить данные главного сервера на какой-то момент, то, быть может, вы сумеете найти потерянные команды и применить их вручную.

Во всех последующих шагах следует использовать при вычислениях значения `Master_Log_File` и `Read_Master_Log_Pos`. Ниже описана процедура повышения подчиненного сервера в случае топологии «главный и подчиненные».

1. Определить, какой из подчиненных серверов содержит наиболее актуальные данные. Выполнить на каждом из них команду `SHOW SLAVE STATUS` и выбрать тот, для которого координаты репликации `Master_Log_File/Read_Master_Log_Pos` самые свежие.
2. Дать всем подчиненным серверам закончить обработку журналов ретрансляции, полученных со старого главного сервера до момента его сбоя. Если сменить главный сервер до того, как подчиненный просмотрел весь журнал ретрансляции, оставшиеся в нем события будут отброшены, и вы никогда не узнаете, на чем подчиненный сервер остановился.
3. Выполнить шаги 5–7 из списка в предыдущем разделе.

[1] Это действительно возможно, хотя MySQL не записывает в журнал события до момента фиксации транзакции. Подробнее см. в разделе «Смешивание транзакционных и нетранзакционных таблиц» на стр. 485.

4. Сравнить координаты `Master_Log_File/Read_Master_Log_Pos` на каждом подчиненном сервере с соответствующими координатами на новом главном.
5. Выполнить шаги 10–12 из списка в предыдущем разделе.

Мы предполагаем, что на всех подчиненных серверах включены параметры `log_bin` и `log_slave_updates`, как было рекомендовано в начале главы. В этом случае вы сможете восстановить все подчиненные серверы на один и тот же момент времени, что по-другому надежно сделать невозможно.

Нахождение нужных позиций в журнале

Если какой-нибудь подчиненный сервер находится не в той же позиции, что главный, то вам предстоит найти в двоичных журналах нового главного сервера позицию, соответствующую последнему событию, реплицированному на этот подчиненный сервер, и указать ее в команде `CHANGE MASTER TO`. Утилита *mysqlbinlog* позволяет узнать последний выполненный подчиненным сервером запрос и найти его в двоичном журнале нового главного сервера. Возможно, придется проделать кое-какие вычисления.

Для иллюстрации предположим, что номера событий журнала возрастают и что самый актуальный подчиненный сервер – новый главный – успел выбрать событие 100 как раз перед сбоем старого главного. Предположим также, что есть еще два подчиненных сервера, `slave2` и `slave3`, причем `slave2` извлек событие 99, а `slave3` – событие 98. Если для обоих подчиненных серверов указать текущую позицию в двоичном журнале нового главного, то они начнут репликацию с события 101, то есть произойдет рассинхронизация. Однако, если двоичный журнал нового главного сервера велся в режиме `log_slave_updates`, то вы сможете найти в нем события 99 и 100 и тем самым восстановить согласованное состояние подчиненных серверов.

Вследствие перезапуска сервера, различий в конфигурации, ротации журналов или выполнения команды `FLUSH LOGS` смещение одного и того же события от начала журнала на разных серверах может отличаться. Поиск событий иногда утомителен и занимает много времени, но обычно ничего сложного в нем нет. Достаточно просмотреть последнее событие, выполненное на каждом сервере, запустив утилиту *mysqlbinlog* для двоичного журнала или журнала ретрансляции подчиненного сервера. Затем нужно отыскать тот же запрос в двоичном журнале нового главного сервера, опять же с помощью *mysqlbinlog*; в результате мы получим смещение запроса в байтах, которое можно подставить в команду `CHANGE MASTER TO`.

Эту процедуру можно ускорить, вычислив разность смещений событий, на которых остановились новый главный и подчиненный сервер. Если затем вычесть эту величину из текущей позиции в двоичном журнале нового главного сервера, то искомый запрос, скорее всего, будет нахо-

диться в получившейся позиции. Только не забудьте убедиться в этом, и если все так, то вы нашли позицию, с которой нужно запускать подчиненный сервер.

Рассмотрим конкретный пример. Пусть `server1` – главный сервер для `server2` и `server3`, причем `server1` вышел из строя. Исходя из значений `Master_Log_File/Read_Master_Log_Pos`, показанных командой `SHOW SLAVE STATUS`, делаем вывод, что `server2` реплицировал все события из двоичного журнала `server1`, но данные на `server3` еще неактуальны. Эта картина изображена на рис. 8.15 (события в журнале и байтовые смещения приведены только для иллюстрации).

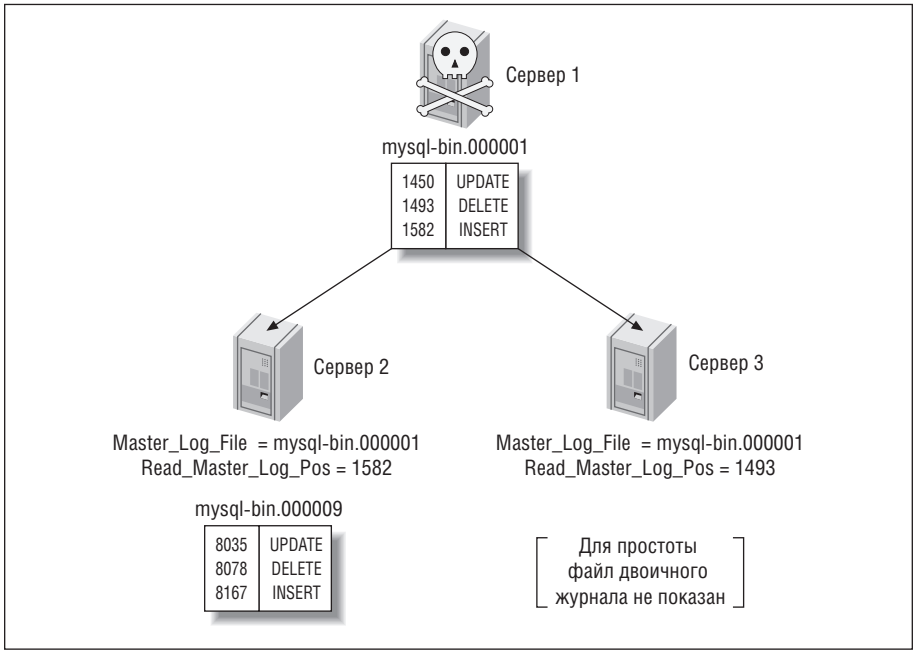


***Рис. 8.15.** В момент сбоя server1 сервер server2 успел обработать все события, а server3 отстал*

Мы можем быть уверены, что `server2` на рис. 8.15 реплицировал все события из двоичного журнала главного сервера, так как значения `Master_Log_File` и `Read_Master_Log_Pos` на нем совпадают с последней позицией на сервере `server1`. Поэтому `server2` можно сделать новым главным, а `server3` – его подчиненным.

Но какие параметры нужно указать в команде `CHANGE MASTER TO` на сервере `server3`? Тут придется немножко посчитать и подумать. Сервер `server3` остановился на смещении 1493, которое на 89 байт отстает от смещения 1582, соответствующего последней команде, выполненной

сервером server2. В данный момент указатель записи в двоичном журнале на server2 находится в позиции 8167. 8167 – 89 = 8078, поэтому теоретически мы должны указать серверу server3 именно эту позицию в двоичном журнале server2. Но все же следует посмотреть, какие события есть в окрестности этой позиции, и убедиться, что там действительно находится то, что нам нужно. Не исключено, что в этом месте оказалась другая команда, например из-за обновления данных, которые выполнялись только на server2.

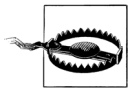
В предположении, что события совпадают, сделать server3 подчиненным для server2 можно следующей командой:

```
server2> CHANGE MASTER TO MASTER_HOST="server2", MASTER_LOG_FILE=
"mysql-bin.000009", MASTER_LOG_POS=8078;
```

А что если к моменту сбоя server1 уже закончил выполнение и протоколирование еще одного события со смещением, большим 1582? Поскольку server2 успел прочитать и выполнить события только до смещения 1582, то одно событие можно потерять навсегда. Однако если диск старого главного сервера не поврежден, то отсутствующее событие все-таки удастся восстановить из его двоичного журнала с помощью утилиты *mysqlbinlog* или сервера журналов.

Если возникает необходимость восстановить отсутствующие события со старого главного сервера, то мы рекомендуем делать это *после* назначения нового главного, но *до* подключения к нему клиентов. В этом случае не придется вручную воспроизводить потерянные события на каждом подчиненном сервере – об этом позаботится механизм репликации. Но если вышедший из строя главный сервер вообще недоступен, то придется подождать и проделать эту работу позже.

Вариацией на ту же тему является надежное хранение двоичных журналов главного сервера, например в SAN-сети или на распределенном реплицируемом блочном устройстве (distributed replicated block device – DRBD). Даже если главный сервер полностью вышел из строя, его двоичные журналы никуда не денутся. Можно настроить сервер журналов, сделать его главным для всех подчиненных серверов и дать им возможность дойти то той точки, где главный сервер вышел из строя. После этого повышение любого из подчиненных серверов до главного достаточно тривиально – по существу, оно ничем не отличается от запланированного повышения. В следующей главе мы рассмотрим различные системы внешней памяти.



При повышении подчиненного сервера до уровня главного не изменяйте его идентификатор. Сделав это, вы не сможете применить сервер журналов для воспроизведения событий старого главного сервера. Это одна из многих причин, по которым идентификаторы серверов следует рассматривать как неизменяемые атрибуты.

Смена ролей в конфигурации главный–главный

Одно из достоинств топологии репликации главный–главный состоит в том, что можно без труда поменять активный и пассивный серверы ролями, так как эта конфигурация симметрична. В настоящем разделе мы рассмотрим, как выполняется такое переключение.

Важнее всего гарантировать, что в любой момент времени запись производится только на один из двух серверов. Если запись ведется на оба сервера, то могут иметь место конфликты. Иными словами, после смены ролей пассивный сервер не должен получать событий из двоичного журнала активного. Чтобы гарантировать такое положение вещей, вы должны подождать, пока поток репликации на подчиненном сервере «подберет» все события активного сервера, и только потом разрешать запись на нем.

Ниже описана последовательность действий, при которой смена ролей не грозит конфликтами в результате обновлений.

1. Прекратить все операции записи на активном сервере.
2. Выполнить на активном сервере команду `SET @@global.read_only := 1` и установить параметр `read_only` в конфигурационном файле, чтобы изменение не потерялось после перезапуска. Напомним, что это не помешает обновлять данные пользователям с привилегией `SUPER`. Если вы хотите воспрепятствовать и им тоже, выполните команду `FLUSH TABLES WITH READ LOCK`. В противном случае следует разорвать все соединения с клиентами, чтобы гарантированно не осталось долго работающих команд и незафиксированных транзакций.
3. Выполнить на активном сервере команду `SHOW MASTER STATUS` и записать координаты репликации.
4. Выполнить на пассивном сервере команду `SELECT MASTER_POS_WAIT()`, указав в ней координаты репликации активного сервера. Эта команда блокирует работу до тех пор, пока все подчиненные процессы не догонят активный сервер.
5. Выполнить на пассивном сервере команду `SET @@global.read_only := 0`, тем самым превратив его в активный.
6. Реконфигурировать приложения так, чтобы они писали на новый активный сервер.

В зависимости от особенностей приложений, возможно, придется произвести и другие действия, например изменить IP-адреса обоих серверов. Мы поговорим об этом в следующей главе.

Проблемы репликации и их решение

Нарушить репликацию в MySQL совсем нетрудно. Простота реализации, позволяющая так легко настраивать механизм, означает одно-

временно, что его можно с той же легкостью остановить, запутать или иным способом вывести из строя. В этом разделе мы расскажем о типичных проблемах – как они проявляются и как можно их разрешить или даже предотвратить.

Ошибки, вызванные повреждением или утратой данных

По разным причинам репликация в MySQL не очень устойчива к сбоям, пропаданию электропитания и повреждению данных, вызванному ошибками диска, памяти или сети. Почти наверняка из-за таких ошибок вам рано или поздно придется перезапускать репликацию.

Причина многих проблем, появляющихся после неожиданного останова сервера, коренится в том, что один серверов не сбросил что-то на диск. Ниже описаны неприятности, с которыми можно столкнуться после неожиданного останова.

Неожиданный останов главного сервера

Если главный сервер не сконфигурирован в режиме `sync_binlog`, то последние несколько событий перед сбоем могли быть не сброшены в двоичный журнал. Поэтому поток ввода/вывода на подчиненном сервере мог находиться в процессе чтения события, которое так никогда и не попало на диск. После перезапуска главного сервера подчиненный повторно соединится с ним и попробует снова прочитать то событие, на котором остановился, однако получит ответ, что события с таким смещением нет. Передача данных из двоичного журнала главного сервера на подчиненный происходит практически мгновенно, поэтому подобную ситуацию нельзя назвать из ряда вон выходящей.

Решение проблемы состоит в том, чтобы заставить подчиненный сервер начать чтение с начала следующего двоичного журнала. А чтобы события журнала не терялись, надо задать на главном сервере режим `sync_binlog`.

Но даже при установке режима `sync_binlog` данные в таблицах типа MyISAM могут быть повреждены при сбое. То же относится и к таблицам InnoDB, если параметр `innodb_flush_logs_at_trx_commit` не равен `1`.

Неожиданный останов подчиненного сервера

Когда подчиненный сервер перезапускается после незапланированного останова, он читает свой файл *master.info*, чтобы понять, на чем прервалась репликация. К сожалению, этот файл не синхронизирован с диском, поэтому информация в нем может не соответствовать действительности. Подчиненный сервер, скорее всего, попытается повторно выполнить несколько событий в двоичном журнале. Это чревато ошибками в связи с дублированием ключа в уникальном индексе. Если вы не можете определить, где на самом деле остановил-

ся подчиненный сервер (а это маловероятно), то не остается другого выбора, как проигнорировать такие ошибки. В этом может помочь инструмент *mk-slaverestart*, входящий в комплект Maatkit.

Если вы работаете только с таблицами InnoDB, то можете заглянуть в журнал ошибок MySQL после перезапуска подчиненного сервера. Процесс восстановления InnoDB выводит координаты репликации вплоть до точки восстановления. Их можно использовать для указания подчиненному серверу позиции в двоичном журнале главного.

Помимо потери данных из-за некорректного останова MySQL не столь уж и редко случается повреждение двоичных журналов или журналов ретрансляции на диске. Перечислим наиболее типичные случаи.

Повреждение двоичных журналов на главном сервере

Если повреждены двоичные журналы на главном сервере, то единственный выход – попробовать обойти испорченный фрагмент. Можно выполнить на главном сервере команду `FLUSH LOGS`, чтобы он начал новый журнал, и указать подчиненному на начало этого журнала. А можно попытаться найти конец поврежденной области. Иногда полезна команда `SET GLOBAL SQL_SLAVE_SKIP_COUNTER = 1`, позволяющая пропустить одно поврежденное событие. Если таких событий несколько, попробуйте повторять эту команду, пока не пропустите их все.

Повреждение журналов ретрансляции на подчиненном сервере

Если двоичные журналы главного сервера не повреждены, то можно воспользоваться командой `CHANGE MASTER TO`, чтобы отбросить поврежденные журналы ретрансляции и попытаться получить их заново. Только нужно указать подчиненному серверу ту же позицию, на которой остановилась репликация (`Relay_Master_Log_File/ Exec_Master_Log_Pos`). Это заставит его отбросить все имеющиеся на диске журналы ретрансляции.

Двоичный журнал рассинхронизирован с журналом транзакций InnoDB

В случае сбоя главного сервера InnoDB может пометить транзакцию как зафиксированную, хотя она не попала в двоичный журнал на диске. Отсутствующую транзакцию восстановить невозможно, если только она не попала в журнал ретрансляции подчиненного сервера. Предотвратить такую ситуацию поможет параметр `sync_binlog` в MySQL 5.0 или пара параметров `sync_binlog` и `safe_binlog` в MySQL 4.1.

Если двоичный журнал поврежден, то количество данных, которое из него удастся восстановить, зависит от характера повреждения. Наиболее распространены следующие варианты.

Байты изменены, но событие представляет собой допустимую SQL-команду

К сожалению, MySQL не способна даже обнаружить такое повреждение. Поэтому имеет смысл время от времени проверять корректность данных на подчиненных серверах.

Байты изменены, и событие представляет собой недопустимую SQL-команду

Возможно, удастся извлечь такое событие с помощью *mysqlbinlog* и увидеть искаженные данные, например:

```
UPDATE tbl SET col????????????????
```

Попробуйте найти начало следующего события, для чего нужно сложить смещение и длину, и напечатайте его. Быть может, удастся просто пропустить поврежденный фрагмент.

Байты пропали и/или длина события неверна

В этом случае *mysqlbinlog* иногда завершается с ошибкой или даже аварийно, поскольку не может ни прочитать событие, ни найти начало следующего.

Несколько событий повреждены, затерты либо произошло смещение данных в журнале, так что смещение следующего события указано неверно

И в этом случае *mysqlbinlog* мало чем поможет.

Если повреждение настолько серьезно, что *mysqlbinlog* не может прочитать события в журнале, то для поиска границ между событиями остается прибегнуть к шестнадцатеричному редактору или иному столь же трудоемкому методу. Но обычно особых трудностей не возникает, так как события разделены четко опознаваемыми маркерами.

Рассмотрим пример. Сначала давайте взглянем, какие смещения событий в воображаемом журнале показывает *mysqlbinlog*:

```
$ mysqlbinlog mysql-bin.000113 | egrep '^# at '
# at 4
# at 98
# at 185
# at 277
# at 369
# at 447
```

Найти смещения в журнале можно, сравнив их с результатом следующей команды *strings*:

```
$ strings -n 2 -t d mysql-bin.000113
1 binpC'G
25 5.0.38-Ubuntu_0ubuntu1.1-log
99 C'G
146 std
156 test
161 create table test(a int)
186 C'G
233 std
243 test
248 insert into test(a) values(1)
```

```
278 C'G
325 std
335 test
340 insert into test(a) values(2)
370 C'G
417 std
427 test
432 drop table test
448 D'G
474 mysql-bin.000114
```

Существует легко распознаваемый образец, позволяющий отыскивать начала событий. Отметим, что строки, кончающиеся буквой 'G', отстоят от начала события на один байт. Они являются частью заголовка события, имеющего фиксированную длину.

Точное значение зависит от исследуемого сервера. После недолгого разбирательства вы сможете обнаружить этот образец в своем двоичном журнале и определить смещение следующего неповрежденного события. После этого можно попытаться пропустить плохие события, задав в командной строке *mysqlbinlog* аргумент *--start-position* или указав параметр `MASTER_LOG_POS` в команде `CHANGE MASTER TO`.

Использование нетранзакционных таблиц

Пока все идет нормально, покомандная репликация как правило отлично работает с нетранзакционными таблицами. Однако если при обновлении нетранзакционной таблицы произошла ошибка, например команда была прервана до того, как завершилась, то на главном и подчиненном сервере могут оказаться разные данные.

Предположим, например, что вы обновляете таблицу типа MyISAM, содержащую 100 строк. Что произойдет, если 50 строк обновилось, а потом команда была прервана? Половина строк изменится, половина – нет. В результате реплика окажется рассинхронизирована, так как команда будет повторена на подчиненном сервере, где обновит все 100 строк (MySQL обнаружит, что команда завершилась с ошибкой на главном сервере, но не на подчиненном, и репликация остановится с сообщением об ошибке).

Если вы работаете с таблицами типа MyISAM, не забывайте выполнять команду `STOP SLAVE` перед остановом сервера, иначе в ходе останова будут прерваны все выполняющиеся запросы (в том числе и незавершенные команды обновления). У транзакционных подсистем хранения такой проблемы не возникает. Незавершенное обновление транзакционной таблицы будет откачено на главном сервере и не попадет в двоичный журнал.

Смешивание транзакционных и нетранзакционных таблиц

При использовании транзакционной подсистемы хранения MySQL не записывает команды в двоичный журнал, пока транзакция не зафиксирована. Следовательно, в случае отката транзакции соответствующие команды не протоколируются и не повторяются на подчиненном сервере. Однако если команда затрагивает одновременно транзакционные и нетранзакционные таблицы, и произошел откат, то MySQL способна откатить лишь изменения в транзакционных таблицах, а изменения в нетранзакционных – сохранятся. Пока не возникает ошибок, например прерванного обновления, никакой проблемы это не составляет: вместо того, чтобы вообще не протоколировать выполненные команды, MySQL заносит в журнал сначала их, а потом команду `ROLLBACK`. В результате на подчиненном сервере выполняются те же самые команды, что на главном, и все довольны. Конечно, это не слишком эффективно, так как подчиненный сервер должен проделать некую работу только для того, чтобы отбросить ее результат, но хотя бы теоретически синхронизация между подчиненным и главным сервером сохраняется.

Но все хорошо только до поры. Проблема появляется, когда на подчиненном сервере возникает взаимоблокировка, которой не было на главном. Изменения в таблицах транзакционного типа на подчиненном сервере откатываются, а в нетранзакционных остаются. В итоге данные на подчиненном и на главном серверах различаются.

Единственный способ предотвратить такую ситуацию – не смешивать транзакционные и нетранзакционные таблицы в одной транзакции. Столкнувшись с подобной проблемой, вы можете сделать только одну вещь – игнорировать ошибку на подчиненном сервере и заново синхронизировать рассогласованные таблицы.

В принципе, построчная репликация не должна быть подвержена таким сложностям. При ее использовании протоколируются изменения строк, а не SQL-команды. Если команда изменяет какие-то строки в MyISAM-таблице и в InnoDB-таблице, а затем на главном сервере возникает взаимоблокировка и изменения в InnoDB-таблице откатываются, то изменения в MyISAM-таблице все равно записываются в двоичный журнал и воспроизводятся на подчиненном сервере. Мы проверили это на простых тестовых примерах и убедились, что все работает правильно; однако на момент написания этой книги у нас не было достаточного опыта работы с построчной репликацией, чтобы можно было с уверенностью сказать, что она полностью решает проблему смешивания транзакционных и нетранзакционных таблиц.

Недетерминированные команды

Любая команда, изменяющая данные недетерминированным образом, может привести к расхождению информации, хранящейся на главном

и подчиненном сервере. Например, команда UPDATE с фразой LIMIT зависит от порядка выборки строк из таблицы. Если не гарантируется, что этот порядок на главном и подчиненном сервере совпадает, – скажем, если строки упорядочены по первичному ключу, – то команда может обновить на двух серверах разные строки. Такого рода труднонаходимые ошибки непросто выявить, поэтому некоторые пользователи берут за правило не использовать фразу LIMIT в командах, которые изменяют данные.

Обращайте также внимание на команды, в которых участвуют таблицы из схемы INFORMATION_SCHEMA. Они вполне могут быть различны на главном и подчиненном сервере, что опять-таки приведет к расхождению. Наконец, имейте в виду, что большинство серверных переменных, например @@server_id и @@hostname, в версиях до MySQL 5.1 реплицируются некорректно.

У построчной репликации таких ограничений нет.

Использование различных подсистем хранения на главном и подчиненном серверах

Мы уже упоминали выше, что на подчиненном сервере часто бывает удобно использовать другую подсистему хранения, что на главном. Однако в некоторых случаях это приводит к тому, что покомандная репликация порождает на подчиненном сервере иные данные. Например, шансы на то, что недетерминированные команды (в частности, упомянутые в предыдущем разделе) приведут к рассогласованию, растут.

Если вы обнаружите, что некоторые таблицы рассогласованы, то проверьте, какие подсистемы хранения указаны для них на обоих серверах, а также изучите все запросы на обновление, в которых участвуют эти таблицы.

Изменение данных на подчиненном сервере

Покомандная репликация полагается на то, что данные на главном и подчиненном сервере одинаковы, поэтому не следует вносить изменения на подчиненном сервере (это легко гарантировать с помощью конфигурационной переменной read_only). Рассмотрим следующую команду:

```
mysql> INSERT INTO table1 SELECT * FROM table2;
```

Если таблица table2 на подчиненном сервере содержит не те же данные, что и на главном, то после выполнения этой команды и таблица table1 начнет отличаться. Другими словами, рассогласование одних таблиц имеет тенденцию перекидываться на другие таблицы. Данное утверждение относится ко всем типам запросов, а не только к INSERT ... SELECT. Кончиться это может двояко: либо вы получите на подчиненном сервере сообщение о нарушении ограничения уникальности, либо не получите никакого сообщения. Оповещение об ошибке – это удача, так как

вы хотя бы знаете, что произошла рассинхронизация подчиненного сервера с главным. Если же ошибка не сопровождается никаким сообщением, то хаос неминуем.

Единственный способ устранить последствия ошибки – заново синхронизировать данные с главным сервером.

Неуникальный идентификатор сервера

Это одна из самых трудноуловимых ошибок, с которыми можно столкнуться при репликации. Если случайно присвоить двум подчиненным серверам один и тот же идентификатор, то может показаться, что все работает нормально. Однако стоит внимательно приглядеться к журналам ошибок или понаблюдать за главным сервером с помощью утилиты *innotop*, как обнаружатся странные вещи.

Видно, что в каждый момент времени к главному серверу подключен только один из двух подчиненных (обычно подчиненные сервера подключены одновременно и все время занимаются репликацией). В журнале ошибок на подчиненном сервере вы увидите частые сообщения о разрыве и восстановлении соединения, но не найдете никакого упоминания о неправильном сконфигурированном идентификаторе.

В зависимости от версии MySQL подчиненные сервера могут реплицировать правильно, но медленно, или вовсе неправильно – любой из них может пропускать события в двоичном журнале или, наоборот, дважды обрабатывать одно и то же событие, что способно привести к нарушению ограничения уникальности (или к искажению данных безо всяких сообщений). Возможен даже сбой или повреждение данных на главном сервере из-за возрастания нагрузки вследствие борьбы подчиненных серверов между собой. А если эта борьба становится особенно ожесточенной, то за очень короткое время журналы ошибок могут вырасти до неимоверных размеров.

Единственное решение этой проблемы – проявлять осторожность при настройке подчиненных серверов. Возможно, стоит вести список идентификаторов, присвоенных подчиненным серверам, чтобы не забыть, какой идентификатор кому из них назначен[1]. Если все подчиненные серверы находятся в одной подсети, то в качестве идентификатора можно выбрть последний октет IP-адреса.

Неопределенный идентификатор сервера

Если в файле *my.cnf* не определен идентификатор сервера, то MySQL позволит настроить репликацию командой `CHANGE MASTER TO`, но не даст запустить подчиненный сервер:

[1] Быть может, вам захочется сохранить его в таблице базы данных? Это шутка лишь наполовину – вполне можно добавить уникальный индекс по столбцу ID.

```
mysql> START SLAVE;
ERROR 1200 (HY000): The server is not configured as slave;
fix in config file or with CHANGE MASTER TO
```

Эта ошибка вызывает особенно сильное недоумение, если вы только что выполнили команду `CHANGE MASTER TO` и проверили настройки командой `SHOW SLAVE STATUS`. Команда `SELECT @@server_id` вернет некоторое значение, но это всего лишь умолчание. Вы должны задать идентификатор явно.

Зависимости от нереплицируемых данных

Если на главном сервере имеются базы данных или таблицы, отсутствующие на подчиненном, то по неосторожности можно легко нарушить репликацию. Предположим, что главный сервер хранит таблицу `scratch`, не существующую на подчиненном. Если на главном сервере будет выполнена любая команда обновления с участием этой таблицы, то репликация нарушится, так как подчиненный сервер не сможет эту команду повторить.

У данной проблемы нет обходного решения. Избежать ее можно только одним способом: не создавая на главном сервере таблиц, которых нет на подчиненном.

Как вообще появляются такие таблицы? Вариантов много, и некоторые предотвратить довольно трудно. Допустим к примеру, что база данных `scratch` изначально была создана только на подчиненном сервере, а затем по какой-то причине главный и подчиненный серверы поменялись ролями. Ну а потом вы, конечно, забыли удалить базу `scratch` и ее привилегии. Далее кто-то подключается к новому главному серверу и запускает в этой базе данных запрос, или периодическое задание обнаруживает в ней таблицы и для каждой из них выполняет команду `OPTIMIZE TABLE`.

Об этом надо помнить, когда вы преобразуете подчиненный сервер в главный или думаете о том, как сконфигурировать подчиненные сервера. Любое отличие подчиненного сервера от главного – источник потенциальных проблем в будущем.

Построчная репликация, по идее, должна устранить некоторые проблемы такого рода, но полной уверенности в этом пока нет.

Отсутствующие временные таблицы

Временные таблицы очень удобны для ряда задач, но, к сожалению, несовместимы с покомандной репликацией. В случае сбоя или останова подчиненного сервера все временные таблицы, которые использовались в потоке репликации, внезапно исчезают. После перезапуска сервера последующие команды, ссылающиеся на отсутствующие временные таблицы, завершаются ошибкой.

Не существует безопасного способа использования временных таблиц на главном сервере в сочетании с покомандной репликацией. Многие на-

столько нежно относятся к временным таблицам, что убедить их в этом почти невозможно, но, увы, это так. Пусть даже временная таблица существует очень короткое время, она все равно потенциально может воспрепятствовать останову и запуску подчиненных серверов и восстановлению после сбоя. Это верно даже в случае, если такая таблица используется только в пределах одной транзакции. Применение временных таблиц на подчиненном сервере, где они могут оказаться удобны, чуть менее проблематично, но только если подчиненный сервер одновременно не является главным.

Если репликация останавливается, потому что подчиненный сервер не может найти временную таблицу после перезапуска, у вас есть всего два выхода. Можно игнорировать ошибку или вручную создать таблицу с таким же именем и структурой, как у пропавшей временной. В любом случае данные на подчиненном и главном сервере, скорее всего, окажутся различными, если существовал хотя бы один запрос на запись с участием временной таблицы.

Исключить временные таблицы не так трудно, как кажется. У них есть два особенно полезных свойства:

- Временная таблица видна только в контексте того соединения, где была создана, поэтому не конфликтует с одноименными таблицами, которые были созданы в других соединениях.
- Временная таблица исчезает при закрытии соединения, поэтому ее не нужно удалять явно.

Эти свойства можно эмулировать, зарезервировав специальную базу данных для псевдовременных таблиц, которые на самом деле являются постоянными. Нужно лишь выбирать для таблиц уникальные имена. К счастью, сделать это довольно просто: достаточно приписать к имени таблицы идентификатор соединения. Например, вместо команды `CREATE TEMPORARY TABLE top_users(...)` можно написать `CREATE TABLE temp.top_users_1234(...)`, где **1234** – значение, возвращенное функцией `CONNECTION_ID()`. После того как приложение закончило работать с временной таблицей, ее можно удалить явно или поручить это специальному процессу очистки. Наличие идентификатора соединения в имени позволяет легко определить, какие таблицы больше не используются, – можно получить список активных соединений с помощью команды `SHOW PROCESSLIST` и сравнить их с идентификаторами в именах таблиц[1].

У использования реальных таблиц вместо временных есть и другие преимущества. Например, это упрощает отладку приложения, так как данные, которыми манипулирует приложение, видны из другого соединения. При использовании временных таблиц эта возможность была бы недоступна.

[1] Программа `mk_find`, также из комплекта Maatkit, позволяет легко удалять псевдовременные таблицы, если задать аргумент *--pid* или *--sid*.

Однако реальные таблицы привносят некоторые издержки, отсутствующие у временных: на их создание уходит больше времени, поскольку соответствующий FRM-файл необходимо сбрасывать на диск. Для ускорения этой процедуры можно отключить режим `sync_frm`, но это опасно.

Если вы используете временные таблицы, то перед остановом подчиненного сервера не забывайте проверять, что переменная состояния `Slave_open_temp_tables` равна 0. Если вы перезапустите подчиненный сервер при ненулевом значении, то в будущем скорее всего получите проблемы. Правильная последовательность действий такова: выполнить команду `STOP SLAVE`, проверить переменную, и только потом останавливать сервер. Проверка переменной до остановки процессов репликации чревата гонкой (race conditions).

Репликация не всех обновлений

Если вы неправильно используете команду `SET SQL_LOG_BIN=0` или не понимаете правил фильтрации репликации, то подчиненный сервер может не выполнить некоторые обновления, произведенные на главном. Иногда именно такая цель и ставится (для архивирования), но чаще это происходит непреднамеренно и приводит к плачевным последствиям.

Предположим к примеру, что имеется правило `replicate_do_db`, которое разрешает реплицировать только базу данных `sakila` на один из подчиненных серверов. Если выполнить на главном сервере следующие команды:

```
mysql> USE test;
mysql> UPDATE sakila.actor ...
```

то данные на подчиненном и на главном сервере рассинхронизируются. Другие команды могут даже привести к ошибке репликации из-за нереплицируемых зависимостей.

Конкуренция, вызванная блокировками при выполнении SELECT в InnoDB

Обычно в InnoDB команды `SELECT` не приводят к блокировке, но в некоторых случаях блокировку все же требуется установить. В частности, команда `INSERT ... SELECT` по умолчанию блокирует все строки, которые читает из исходной таблицы. MySQL вынужден ставить блокировки для того, чтобы эта команда давала тот же самый результат при ее повторении на подчиненном сервере. По существу, блокировка сериализует команду на главном сервере, что совпадает со способом выполнения ее на подчиненном.

Такое проектное решение может приводить к конкуренции за блокировки и таймаутам в ожидании блокировки. Чтобы несколько смягчить остроту проблемы, не держите транзакцию на главном сервере от-

крытой дольше, чем необходимо; это уменьшит время удержания блокировок.

Кроме того, рекомендуется разбивать длинные команды на более короткие. Это эффективный способ уменьшить конкуренцию за блокировки, и даже если сделать это трудно, попытаться все равно стоит.

Еще один обходной путь – заменить на главном сервере команды `INSERT ... SELECT` последовательностью команд `SELECT INTO OUTFILE` и `LOAD DATA INFILE`. Работает такой способ быстро и не требует блокирования. Мы согласны, что это «грязный трюк», но иногда он бывает полезен. Самое трудное здесь – выбрать уникальное имя выходного файла и стереть его по завершении работы. Для подбора уникального имени можно воспользоваться описанной в разделе «Отсутствующие временные таблицы» (стр. 488) техникой работы с функцией `CONNECTION_ID()` и периодически запускать задание, которое будет удалять файлы после того, как завершатся соединения, в которых они были созданы (с помощью *crontab* в UNIX-системах или планировщика задач в Windows).

Не поддавайтесь искушению отключить блокировки вовсе вместо использования описанных выше обходных путей. Это можно сделать, но в большинстве случаев это безрассудно, так как в таком случае подчиненный сервер может рассинхронизироваться с главным без каких бы то ни было сообщений. Если вы все же решите, что риск оправдан, то установите следующий конфигурационный параметр:

```
innodb_locks_unsafe_for_binlog = 1
```

Теперь результат выполнения команды может зависеть от данных, при чтении которых не ставились блокировки. Если какая-то другая команда модифицирует эти данные и завершится раньше первой, то при повторе тех же команд на подчиненном сервере могут получиться иные результаты. Это справедливо как для репликации, так и для восстановления на конкретный момент времени в прошлом.

Чтобы понять, как блокировка предотвращает подобный хаос, представьте, что есть две таблицы: одна пустая, а другая с единственной строкой, в которой хранится значение `99`. Указанные данные обновляются в двух транзакциях. Транзакция 1 вставляет содержимое второй таблицы в первую, а транзакция 2 обновляет вторую (исходную) таблицу, как показано на рис. 8.16.

В этой последовательности очень важен шаг 2. На нем транзакция 2 пытается обновить исходную таблицу, для чего должна поставить монопольную блокировку (записи) на строки, которые собирается обновить. Монопольная блокировка не совместима ни с какой другой, в том числе и с разделяемой блокировкой, поставленной на строку транзакцией 1, поэтому транзакция 2 должна дождаться фиксации транзакции 1. Транзакции записываются в двоичный журнал в порядке фиксации, поэтому их повтор на подчиненном сервере даст точно такие же результаты, как на главном.

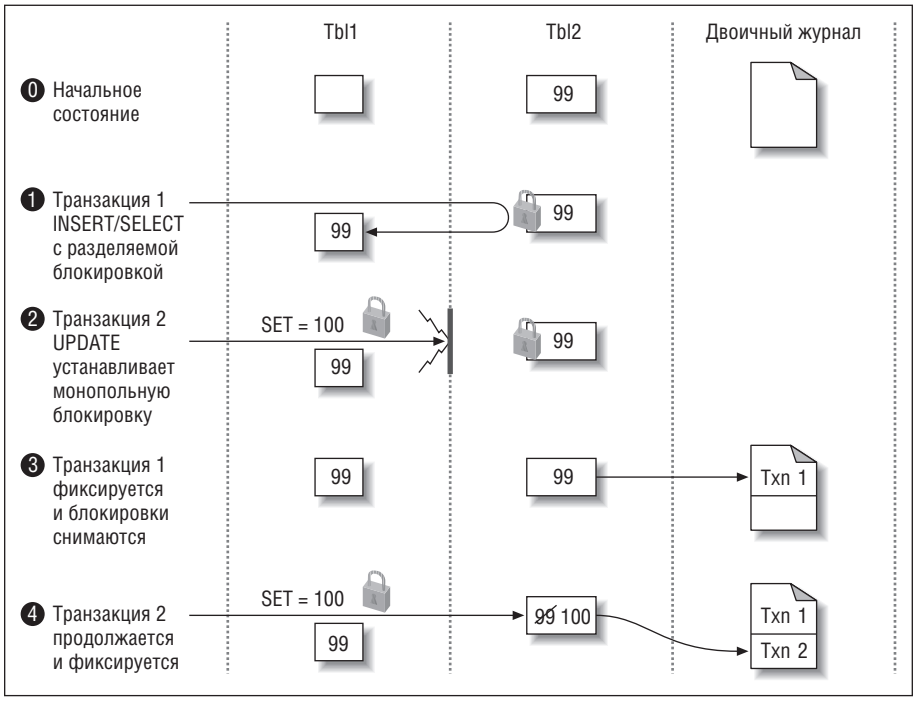


Рис. 8.16. *Две транзакции обновляют данные, устанавливая разделяемые блокировки, чтобы сериализовать обновления*

С другой стороны, если транзакция 1 не ставит разделяемую блокировку на строки, читаемые для команды `INSERT`, то никаких гарантий не дается. Взгляните на рис. 8.17, где показана возможная последовательность событий в случае, когда блокировки не ставятся.

Отсутствие блокировок позволяет записать транзакции в двоичный журнал в таком порядке, при котором повторное воспроизведение даст иные результаты, что и видно из рисунка. Сначала MySQL протоколирует транзакцию 2, поэтому она повлияет на результаты выполнения транзакции 1 на подчиненном сервере. На главном ничего подобного не было. В итоге данные на главном и подчиненном сервере различаются.

Мы настоятельно рекомендуем в большинстве случаев оставлять параметр `innodb_locks_unsafe_for_binlog` равным `0`.

Запись на обоих главных серверах в конфигурации главный–главный

В общем случае писать на обоих главных серверах – не слишком удачная мысль. Некоторые возникающие при этом проблемы можно разрешить безопасным образом, другие – нет.

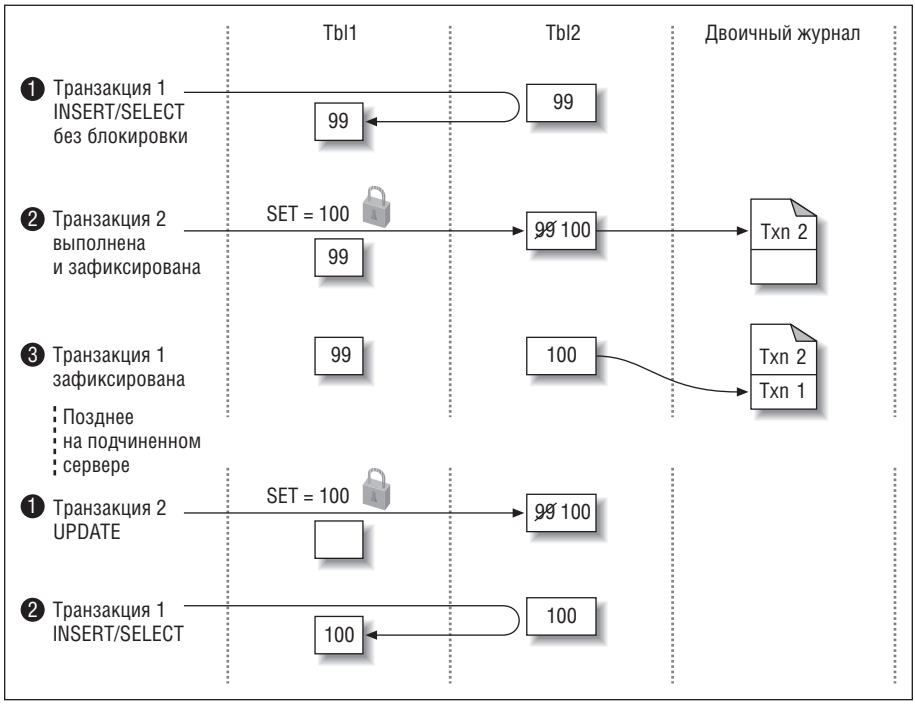


Рис. 8.17. *Две транзакции обновляют данные, не устанавливая разделяемые блокировки для сериализации обновлений*

В версии MySQL 5.0 появилось два конфигурационных параметра, которые помогают разрешить конфликт между автоинкрементными первичными ключами (с модификатором `AUTO_INCREMENT`). Они называются `auto_increment_increment` и `auto_increment_offset`. Смысл в том, чтобы значения, генерируемые на разных серверах, чередовались, а не повторялись.

Однако так можно решить только проблему автоинкремента, а это лишь малая толика тех трудностей, с которыми придется столкнуться, если вы захотите писать на оба сервера. Более того, это решение на самом деле порождает ряд новых проблем:

- Усложняется смена ролей серверов, участвующих в репликации.
- Из-за промежутков (gaps) в нумерации напрасно расходуется пространство ключей.
- Оно работает только в том случае, когда для всех таблиц определены автоинкрементные первичные ключи, а заводить такие ключи не всегда имеет смысл.

Вы можете самостоятельно генерировать неконфликтующие первичные ключи. Один из способов – создать первичный ключ по нескольким столбцам и в первом столбце хранить идентификатор сервера. Такой

метод работает, но увеличивается длина первичного ключа, а в InnoDB это отражается еще и на внешних ключах.

В качестве альтернативы можно строить первичный ключ по одному столбцу, а идентификатор сервера хранить в старших битах целочисленного значения ключа. Реализуется это простыми операциями поразрядного сдвига влево и сложения. Например, если 8 старших битов беззнакового столбца типа `BIGINT` (64-разрядного) отведено под идентификатор сервера, то вставить значение 11 на сервере 15 можно так:

```
mysql> INSERT INTO test(pk_col, ...) VALUES( (15 << 56) + 11, ...);
```

Понять, что при этом происходит, проще всего, если перейти к основанию 2 и дополнить результат нулями до длины 64:

```
mysql> SELECT LPAD(CONV(pk_col, 10, 2), 64, '0') FROM test;
+------------------------------------------------------------------+
| LPAD(CONV(pk_col, 10, 2), 64, '0') |
+------------------------------------------------------------------+
| 0000111100000000000000000000000000000000000000000000000000001011 |
+------------------------------------------------------------------+
```

Но при таком подходе вам придется генерировать значения ключей самостоятельно, поскольку режим автоинкремента отключен. Не используйте `@@server_id` вместо константы 15 в команде `INSERT`, поскольку тогда на подчиненном сервере получатся другие результаты.

Можно также прибегнуть к псевдослучайным значениям, используя такие функции, как `MD5()` или `UUID()`, но это негативно скажется на производительности, так как ключи получаются длинными и случайными, что плохо, в частности, для InnoDB. Не пользуйтесь функцией `UUID()`, если только не генерируете значения в коде приложения, поскольку эта функция некорректно реплицируется в случае покомандной репликации.

В общем, эта задача с трудом поддается решению, поэтому обычно мы рекомендуем перепроектировать приложение так, чтобы запись производилась только на один главный сервер.

Слишком большое отставание репликации

Отставание репликации – распространенная проблема. Вне зависимости от причины рекомендуется проектировать приложение таким образом, чтобы оно сохраняло работоспособность при небольшом отставании подчиненных серверов. Если система не может работать в подобных условиях, то, по-видимому, ее архитектура не должна быть основана на репликации. Однако можно предпринять кое-какие шаги, чтобы подчиненные серверы не отставали от главного слишком далеко.

Однопоточная природа репликации в MySQL означает, что на подчиненном сервере она принципиально менее эффективна. Даже очень

быстрый подчиненный сервер с большим количеством дисков, процессоров и кучей памяти легко может отстать от главного, поскольку единственный поток обычно задействует лишь один процессор и диск. Но в любом случае подчиненный сервер должен быть не менее мощным, чем основной.

Блокировка на подчиненных серверах также представляет проблему. Прочие запросы, работающие на подчиненном сервере, могут заблокировать поток репликации. Поскольку репликация однопоточная, то пока этот поток ждет, он не может выполнять никакой полезной работы.

Отставание репликации может проявляться двояко: в виде кратковременных всплесков, после которых подчиненный сервер догоняет главный, и постоянного отставания. В первом случае запаздывание обычно вызвано одиночным долго работающим запросом, тогда как во втором никаких особенно длительных запросов может и не быть, а отставание все равно накапливается.

К несчастью, в настоящее время понять, насколько подчиненный сервер близок к исчерпанию своей пропускной способности, можно лишь путем изучения косвенных эмпирических данных. Если нагрузка строго равномерна во времени, то подчиненный сервер будет одинаково хорошо работать вне зависимости от того, какая доля пропускной способности задействована, – 10 или 99 процентов, но при достижении 100% он внезапно начнет отставать. На практике нагрузка редко бывает равномерной, поэтому по мере приближения к пределу пропускной способности в пиковые моменты вы, вероятно, будете наблюдать, как отставание растет.

Это плохой знак! Вероятно, вы опасно приблизились к той точке, после которой подчиненные серверы уже не смогут догнать главный даже при временном снижении нагрузки. Чтобы грубо оценить, насколько близок потолок, можно остановить на время поток SQL на подчиненном сервере, затем снова запустить его и посмотреть, сколько времени уйдет на то, чтобы догнать главный сервер.

Компания Google выпустила ряд заплат к MySQL (см. раздел «Синхронная репликация в MySQL», стр. 558), среди которых есть команда `SHOW-USER STATISTICS`, печатающая столбец `Busy_time` – процентную долю времени, потраченного потоком репликации на обработку запросов. Это еще один способ оценить запас прочности, имеющийся у потока SQL.

Протоколирование запросов на подчиненном сервере и использование какого-нибудь инструмента анализа журналов, который может показать самые медленные запросы, – едва ли не лучшее, что можно сделать, когда подчиненный сервер перестает справляться. Не доверяйте интуитивным представлениям о том, что может работать медленно, и не спешите с выводами, основанными на сведениях, как те или иные запросы выполняются на главном сервере, поскольку характеристики производительности на главном и подчиненном сервере силь-

но различаются. Самый лучший способ проделать такой анализ – временно включить журнал медленных запросов на подчиненном сервере. В стандартный журнал медленных запросов MySQL не включаются запросы, выполняемые потоком репликации, поэтому он не позволяет понять, какие из реплицируемых запросов «тормозят». Эта проблема решена в заплате, реализующей протоколирование медленных запросов с микросекундной точностью (см. раздел «Профилирование MySQL» в главе 2, стр. 96).

Если подчиненный сервер не справляется с репликацией, то его настройка мало чего позволит добиться, положительного эффекта можно достигнуть, разве что установив более быстрые диски и процессоры. Большинство рекомендаций связано с отключением чего-нибудь в попытке слегка уменьшить нагрузку на подчиненный сервер. Одно из простых решений – уменьшить частоту сброса на диск в подсистеме InnoDB, вследствие чего транзакции станут фиксироваться быстрее. Для этого присвойте параметру `innodb_flush_log_at_trx_commit` значение `2`. Можно также отключить на подчиненном сервере запись в двоичный журнал. Для этого в случае InnoDB нужно задать параметр `innodb_locks_unsafe_for_binlog` равным `1`, а в случае MyISAM – присвоить параметру `delay_key_write` значение `ALL`. Но при этом за быстродействие приходится расплачиваться надежностью. Повышая подчиненный сервер до уровня главного, не забудьте восстановить безопасные значения.

Не дублируйте дорогостоящие операции записи

Часто самый лучший способ помочь подчиненным серверам справиться с нагрузкой заключается в том, чтобы перепроектировать приложение или оптимизировать запросы. Попробуйте свести к минимуму объем работы, дублируемой в разных частях системы. Любая дорогостоящая операция записи на главном сервере должна быть повторена на каждом подчиненном. Если можно переместить часть работы с главного сервера на подчиненный, то выполнять ее придется только одному из подчиненных серверов. Затем результаты работы можно загрузить на главный сервер, например командой `LOAD DATA INFILE`.

Приведем пример. Предположим, что имеется очень большая таблица, которую необходимо агрегировать и свести к меньшей по размерам:

```
mysql> REPLACE INTO main_db.summary_table (col1, col2, ...)
    -> SELECT col1, sum(col2, ...)
    -> FROM main_db.enormous_table GROUP BY col1;
```

Если выполнять эту операцию на главном сервере, то каждый подчиненный вынужден будет повторить гигантский запрос с фразой `GROUP BY`. Когда подобных запросов станет много, подчиненные сервера начнут отставать. Поможет перенос запроса на какой-нибудь подчиненный сервер. На нем можно завести специальную базу, чтобы избежать конфликтов при обновлении данных, реплицируемых с главного сервера, и выполнить такой запрос:

```
mysql> REPLACE INTO summary_db.summary_table (col1, col2, ...)
    -> SELECT col1, sum(col2, ...)
    -> FROM main_db.enormous_table GROUP BY col1;
```

Теперь можно выполнить команду SELECT INTO OUTFILE, а вслед за ней команду LOAD DATA INFILE на главном сервере, чтобы загрузить в его базу данных результаты. Вуаля – дублирование работы свелось к выполнению простейшей директивы LOAD DATA INFILE. Если имеется N подчиненных серверов, то мы сэкономили $N - 1$ длительных запросов с фразой GROUP BY.

Впрочем, у этой стратегии есть и недостаток – неактуальные данные. Иногда бывает трудно получить согласованные результаты, выполняя чтение таблиц на подчиненном сервере и запись – на главном (эту проблему мы будем обсуждать в следующей главе). Если чтение на подчиненном сервере вызывает сложности, то тогда и задачу можно упростить, сэкономив подчиненным серверам массу работы. Если отделить части REPLACE и SELECT друг от друга, то можно произвести выборку результатов в приложении, а затем вставить эти результаты обратно в базу данных на главном сервере. Сначала выполним на главном сервере следующий запрос:

```
mysql> SELECT col1, sum(col2, ...) FROM main_db.enormous_table GROUP BY col1;
```

Затем вставим полученные результаты в сводную таблицу, повторив для каждой строки результирующего набора такой запрос:

```
mysql> REPLACE INTO main_db.summary_table (col1, col2, ...) VALUES (?, ?, ...);
```

Как и раньше, нам удалось избавить подчиненные серверы от выполнения трудоемкой группировки; разделение SELECT и REPLACE означает, что часть команд SELECT, которая отвечает за агрегацию, не повторяется на каждом подчиненном сервере.

Эта общая стратегия – избавление подчиненных серверов от дорогостоящей части операции записи – помогает во многих ситуациях, когда результаты запроса сложно получить, но после получения легко обработать.

Выполняйте параллельную запись без репликации

Еще один прием, помогающий сократить отставание подчиненных серверов, – обойти репликацию вовсе. Любая операция записи, выполненная на главном сервере, должна быть сериализована на подчиненном, поэтому «сериализованную запись» можно рассматривать как дефицитный ресурс. Задайтесь вопросом, все ли операции записи необходимо пропускать через репликацию от главного к подчиненному? Как можно сделать так, чтобы ограниченное количество ресурса «сериализованная запись» расходовалось только на те процедуры, которые действительно должны проходить через репликацию?

Такая постановка вопроса помогает приоритезировать операции записи. В частности, если удается выявить действия, легко выполняемые вне репликации, то их можно распараллелить и сэкономить драгоценный ресурс подчиненного сервера.

Замечательный пример дает архивирование, уже обсуждавшееся выше в этой главе. В OLTP-системах данные часто архивируются по строке за раз. Если просто переместить ненужные строки из одной таблицы в другую, то причина для репликации таких операций записи на подчиненных серверах может просто отпасть. Вместо этого можно отключить запись в двоичный журнал для команд архивирования, а затем запустить отдельные идентичные процессы архивирования на главном и всех подчиненных серверах.

Идея о том, чтобы копировать данные на другой сервер самостоятельно вместо того, чтобы полагаться на репликацию, может показаться безумной, но в некоторых программах она имеет смысл. В особенности это относится к случаю, когда приложение является единственным источником обновлений в некотором наборе таблиц. Часто узким местом является небольшое подмножество всех таблиц, и если именно их вывести из общей последовательности репликации, то можно существенно выиграть в скорости.

Инициализируйте кэш для потока репликации

Если позволяет рабочая нагрузка, то можно получить выигрыш от распараллеливания ввода/вывода на подчиненные сервера за счет предварительной загрузки данных в память. Эта техника не очень хорошо известна, но мы знаем, что она успешно применяется в некоторых крупных приложениях.

Идея состоит в том, чтобы использовать программу, которая немного опережает поток SQL в чтении данных из журналов ретрансляции и выполняет запросы в виде команд `SELECT`. В результате сервер считывает информацию с диска в память, так что к тому моменту, как поток SQL дойдет до соответствующей команды в журнале ретрансляции, ему не придется ждать выборки данных с диска. По существу, мы распараллеливаем ввод/вывод, который поток SQL обычно должен производить последовательно. Пока одна команда изменяет данные, в память с диска загружаются значения для следующей команды.

Насколько эта программа должна опережать поток SQL, зависит от обстоятельств. Попробуйте установить интервал в несколько секунд или задать эквивалентное количество байтов в журнале ретрансляции. Если забежать слишком далеко вперед, то данные, помещенные в кэш, могут уже быть вытеснены оттуда к тому моменту, когда они потребуются потоку SQL.

Рассмотрим на примере, как можно переписать команды, чтобы воспользоваться этой идеей. Возьмем такой запрос:

```
mysql> UPDATE sakila.film SET rental_duration=4 WHERE film_id=123;
```

Следующая команда SELECT извлекает ту же строку и столбцы:

```
mysql> SELECT rental_duration FROM sakila.film WHERE film_id=123;
```

Этот прием работает только при подходящей рабочей нагрузке и конфигурации оборудования. Необходимы (но, возможно, не достаточны) следующие условия:

- Поток SQL выполняет большой объем ввода/вывода, но в целом подчиненный сервер не перегружен этими операциями. Если это не так, то предварительная выборка не даст никакого выигрыша, поскольку диски сервера не простаивают.
- Рабочее множество гораздо больше доступного объема памяти (именно поэтому поток SQL тратит много времени на ожидание завершения ввода/вывода). Если рабочее множество умещается в памяти, то предварительная выборка не поможет[1], так как по прошествии некоторого времени сервер «прогреется» и ожидание ввода/вывода станет редким явлением.
- Подчиненный сервер оснащен большим количеством дисков. У наших знакомых, успешно применявших эту тактику, было по восемь и более дисков на каждом подчиненном сервере. Возможно, и меньшего числа будет достаточно, но заведомо необходимо минимум от двух до четырех дисков.
- Используется подсистема хранения с блокировкой на уровне строк, например InnoDB. Попытка проделать то же самое для таблицы типа MyISAM, скорее всего, приведет к конкуренции между потоком SQL и потоком предварительной выборки за табличные блокировки, что лишь замедлит работу. Однако если таблиц много, а операции записи распределены между ними равномерно, то теоретически репликацию MyISAM-таблиц, возможно, удастся ускорить.

В качестве примера рабочей нагрузки, которая выигрывает от предварительной выборки, можно назвать большое количество команд UPDATE, обновляющих по одной строке в разных местах базы данных. Такая нагрузка обычно допускает высокую степень конкуренции на главном сервере. Команды DELETE тоже можно ускорить таким образом. Для команд INSERT это менее вероятно, особенно если строки вставляются последовательно, поскольку конец индекса и так уже «горячее» место от предыдущих вставок.

Если по столбцам таблицы построено много индексов, то не всегда возможно предварительно выбрать все данные, которые может изменить команда. Команда UPDATE в состоянии модифицировать каждый индекс, тогда как SELECT обычно читает только первичный ключ и в лучшем случае один вторичный индекс. Поскольку UPDATE должна будет прочи-

[1] Вернее, будет не столь важна. – *Прим. науч. ред.*

тать и другие индексы перед тем, как их модифицировать, то тактика предварительной выборки оказывается не так эффективна.

Чтобы понять, достаточно ли свободного времени у дисков, чтобы обслуживать запросы на предварительную выборку, можно воспользоваться утилитой *iostat*. Обращайте внимание на длину очереди и время обслуживания (примеры см. в предыдущей главе). Короткая очередь говорит о том, что какой-то механизм на более высоком уровне сериализует запросы. Длинная очередь свидетельствует о высокой нагрузке – вряд ли ее создает поток SQL при наличии большого количества дисков. Если время обслуживания велико, то диску одновременно отправляется слишком много запросов.

Описанная техника – не панацея. Существует много причин, по которым она может не давать эффекта и даже усугублять ситуацию. Применять ее следует лишь после тщательного изучения оборудования и операционной системы. Мы знаем людей, которым этот подход позволил увеличить скорость репликации на 300–400%, но когда мы попробовали его сами, у нас ничего не вышло. Очень важно правильно выставить параметры, но не всегда требуемое сочетание настроек существует. Иногда файловая система или ядро ОС могут свести на нет выгоды от параллельного ввода/вывода. Каждый случай уникален!

Инструмент *mk-slave-prefetch*, входящий в состав комплекта Maatkit, – одна из возможных реализаций идей, описанных в этом разделе.

Чрезмерно большие пакеты от главного сервера

Еще одна трудно обнаруживаемая проблема репликации связана с тем, что параметр `max_allowed_packet` на главном и подчиненном сервере не совпадает. В этом случае главный сервер может поместить в журнал пакет, который подчиненный сочтет слишком большим, и при попытке прочитать событие из двоичного журнала возникнут разного рода сбои. Это может, в частности, привести к бесконечному циклу ошибок и повторных попыток чтения или к повреждению журнала ретрансляции.

Ограниченная пропускная способность сети

Если репликация производится по сети с ограниченной пропускной способностью, то можно включить на подчиненном сервере режим `slave_compressed_protocol` (реализован в версии MySQL 4.0 и более поздних). Когда подчиненный сервер соединяется с главным, он запрашивает сжатие – точно такое же, какое доступно любому клиенту MySQL. Для сжатия используется библиотека *zlib*, и наши тесты показывают, что текстовые данные иногда удается сжать до трети первоначального размера. Но при этом требуется дополнительное процессорное время как на главном сервере (для упаковки), так и на подчиненном (для распаковки).

Если канал на стороне главного сервера медленный, а на другом его конце много подчиненных серверов, то имеет смысл поставить между

главными и подчиненными серверами сервер-распространитель. Тогда с главным сервером по медленному каналу будет соединяться только один сервер, что уменьшит и потребление пропускной способности, и нагрузку на процессор главного сервера.

Отсутствие места на диске

Репликация может заполнить диски двоичными журналами, журналами ретрансляции и временными файлами, особенно если на главном сервере выполняется много команд `LOAD DATA INFILE`, а на подчиненном включен режим `log_slave_updates`. Чем сильнее отстает подчиненный сервер, тем больше места необходимо для журналов ретрансляции, которые уже прочитаны с главного сервера, но еще не обработаны. Чтобы предотвратить такие ошибки, следите за свободным местом на дисках и используйте параметр `relay_log_space`.

Ограничения репликации

Репликация в MySQL может остановиться или привести к рассинхронизации, с сообщением об ошибке или без, просто из-за внутренне присущих ей ограничений. Перечень функций SQL и приемов программирования, которые невозможно надежно реплицировать, довольно обширен (многие из них уже упоминались в данной главе). Трудно гарантировать, что ничего из этого списка не просочится в код промышленной системы, особенно если приложение большое или коллектив насчитывает много разработчиков[1].

Немало неприятностей причиняют и ошибки в коде сервера. Не хотим показаться критиканами, но в большинстве основных версий MySQL встречались ошибки в коде репликации, особенно в первых выпусках. Появление новых средств, в частности хранимых процедур, приводило и к новым ошибкам.

Для большинства пользователей это еще не причина сторониться новых возможностей. Это лишь повод для более тщательного тестирования, особенно при переходе на новую версию приложения или MySQL. Важен и мониторинг – проблема не должна оставаться незамеченной.

Репликация в MySQL устроена сложно, а чем сложнее приложение, тем тщательнее должно быть тестирование. Но если понять, как работать с репликацией, то все будет хорошо.

Насколько быстро работает репликация?

Говоря о репликации, часто спрашивают: «насколько быстро она работает?» Коротко можно ответить, что в общем случае очень быстро, на-

[1] Увы, в MySQL отсутствует параметр `forbid_operations_unsafe_for_replication` (запретить операции, небезопасные для репликации).

столько быстро, насколько MySQL успевает копировать события с главного сервера и воспроизводить их. Если сеть медленная, а события в двоичном журнале очень велики, то задержка между записью в двоичный журнал и выполнением на подчиненном сервере может оказаться заметной. Если запросы выполняются долго, а сеть быстрая, то можно ожидать, что время выполнения запроса на подчиненном сервере будет составлять превалирующую долю общего времени, необходимого для репликации события.

Чтобы дать более точный ответ, нужно измерить каждый шаг процедуры и понять, на какие шаги тратится больше всего времени. Некоторым читателям достаточно того, что обычно промежуток времени между протоколированием событий на главном сервере и их копированием в журнал ретрансляции очень мал. Для тех же, кто любит детали, мы провели небольшой эксперимент.

Мы усовершенствовали процедуру, описанную в первом издании этой книги, и воспользовались методикой Джузеппе Максиа (Giuseppe Maxia)[1], для того чтобы с высокой точностью измерить скорость репликации. Мы написали недетерминированную пользовательскую функцию (UDF), которая возвращает системное время с точностью до микросекунд (ее исходный код приведен в разделе «Определяемые пользователем функции» главы 5, на стр. 291):

```
mysql> SELECT NOW_USEC()
+----------------------------+
| NOW_USEC( ) |
+----------------------------+
| 2007-10-23 10:41:10.743917 |
+----------------------------+
```

Это позволило нам измерить скорость репликации, для чего мы вставляли значение `NOW_USEC()` в таблицу на главном сервере, а затем сравнивали его со значением на подчиненном сервере.

Мы измеряли задержку, когда оба экземпляра MySQL были установлены на одной машине, чтобы избежать неточностей из-за расхождений в показаниях системных часов. Один экземпляр мы сконфигурировали как подчиненный другому, а затем выполнили на главном экземпляре следующие запросы:

```
mysql> CREATE TABLE test.lag_test(
    -> id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    -> now_usec VARCHAR(26) NOT NULL
    -> );
mysql> INSERT INTO test.lag_test(now_usec) VALUES( NOW_USEC( ) );
```

Мы воспользовались столбцом типа `VARCHAR`, потому что встроенные в MySQL типы для работы с временем не могут хранить данные с суб-

[1] См. *http://datacharmer.blogspot.com/2006/04/measuring-replication-speed.html*.

секундной точностью (хотя некоторые функции умеют производить подобные вычисления). Остается лишь определить разность между моментами времени на главном и подчиненном сервере. Для этого отлично подойдет таблица типа Federated. На подчиненном сервере был выполнен запрос:

```
mysql> CREATE TABLE test.master_val (
    -> id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    -> now_usec VARCHAR(26) NOT NULL
    -> ) ENGINE=FEDERATED
    -> CONNECTION='mysql://user:pass@127.0.0.1/test/lag_test';
```

Простое соединение таблиц и функция `TIMESTAMPDIFF()` позволяют узнать, сколько микросекунд прошло между выполнением запроса на главном и подчиненном сервере:

```
mysql> SELECT m.id, TIMESTAMPDIFF(FRAC_SECOND, m.now_usec, s.now_usec) AS
usec_lag
    -> FROM test.lag_test as s
    -> INNER JOIN test.master_val AS m USING(id);
+----+----------+
| id | usec_lag |
+----+----------+
|  1 |      476 |
+----+----------+
```

Мы написали Perl-сценарий, которой вставляет 1000 строк в таблицу на главном сервере с 10-миллисекундной задержкой между последовательными вставками, чтобы предотвратить состязание между главным и подчиненным экземпляром за процессорное время. Затем мы построили временную таблицу, содержащую задержки для каждого события:

```
mysql> CREATE TABLE test.lag AS
    -> SELECT TIMESTAMPDIFF(FRAC_SECOND, m.now_usec, s.now_usec) AS lag
    -> FROM test.master_val AS m
    -> INNER JOIN test.lag_test as s USING(id);
```

После этого мы сгруппировали результаты по времени, чтобы узнать, какая задержка встречалась чаще всего:

```
mysql> SELECT ROUND(lag / 1000000.0, 4) * 1000 AS msec_lag, COUNT(*)
    -> FROM lag
    -> GROUP BY msec_lag
    -> ORDER BY msec_lag;
+----------+----------+
| msec_lag | COUNT(*) |
+----------+----------+
|   0.1000 |      392 |
|   0.2000 |      468 |
|   0.3000 |       75 |
|   0.4000 |       32 |
|   0.5000 |       15 |
|   0.6000 |        9 |
```

```
|    0.7000 |         2 |
|    1.3000 |         2 |
|    1.4000 |         1 |
|    1.8000 |         1 |
|    4.6000 |         1 |
|    6.6000 |         1 |
|   24.3000 |         1 |
+-----------+-----------+
```

Как видно, для большинства мелких запросов между выполнением на главном и на подчиненном сервере проходит менее 0,3 мс.

Чего мы *не можем* измерить таким образом, так это время доставки события на подчиненный сервер после того, как оно было записано в двоичный журнал главного. А хорошо бы его знать, поскольку чем раньше подчиненный сервер получит событие, тем лучше. Если событие получено, то подчиненный сервер будет иметь копию в случае сбоя главного.

Хотя наши измерения не показывают, сколько времени занимает эта часть процедуры, теоретически она должна выполняться чрезвычайно быстро (ограничена только скоростью сети). Процесс дампа двоичного журнала *не* опрашивает главный сервер о появлении новых событий, это было бы медленно и неэффективно. Вместо этого главный сервер уведомляет подчиненный о новых событиях. Чтение события из двоичного журнала главного сервера – это блокирующий системный вызов, который начинает посылать данные практически сразу после записи в журнал. Таким образом, можно с большой долей уверенности сказать, что событие прибывает на подчиненный сервер за то время, которое требуется, чтобы «разбудить» (wake up) поток репликации на подчиненном сервере и передать информацию по сети.

Перспективы репликации в MySQL

У механизма репликации в MySQL есть целый ряд дефектов, которые компания MySQL AB планирует устранить в будущем. Кое-что в этом направлении уже сделали третьи стороны. Так, компания Google выпустила набор заплат к серверу MySQL, в которых реализована полусинхронная репликация и много других возможностей (см. раздел «Синхронная репликация в MySQL», стр. 558).

Еще одно возможное дополнение – репликация с несколькими главными серверами, когда один подчиненный сервер получает обновления от нескольких главных. Скорее всего, это окажется сложной задачей, для решения которой придется реализовать механизм обнаружения и разрешения конфликтов. Построчная репликация в версии MySQL 5.1 – шаг в этом направлении. В будущем алгоритм построчной репликации, возможно, будет изменен так, чтобы обрабатывать события на подчиненном сервере могли сразу несколько потоков, что устранит узкое место, обусловленное нынешней однопоточностью.

Существуют также планы интегрировать API оперативного резервного копирования с репликацией и наделить MySQL возможностью автоматически конфигурировать себя в качестве подчиненного другому серверу.

В настоящее время MySQL не дает гарантий согласованности и корректности данных. По результатам опроса, проведенного на сайте MySQL, самым востребованным дополнением является механизм оперативной проверки согласованности, который позволял бы узнать, отличаются ли данные на главном и подчиненном сервере. Компания MySQL AB открыла рабочий журнал, посвященный этой теме, с кратким описанием того, как это будет сделано. Многие пользователи также просили расширить формат двоичного журнала так, чтобы можно было обнаруживать повреждения, и MySQL AB согласилась, что это важная задача.

Эти и многие другие усовершенствования должны в будущем сделать репликацию более мощным и надежным механизмом. Очень поучительно оглянуться назад и вспомнить, какие изменения были внесены за последние несколько лет. Однако не стоит забывать и о том, что большая часть функций, появление которых предсказывалось в первом издании этой книги, так и не увидели света, пусть даже некоторые из них частично реализованы. Например, проект отказоустойчивой репликации имеется в исходных кодах MySQL, но сейчас заброшен.

9

Масштабирование и высокая доступность

В этой главе мы покажем, как на основе СУБД MySQL можно построить решение, которое масштабируется в очень широких пределах, оставаясь при этом быстрым и надежным.

Проблемы, связанные с масштабированием, как правило, проявляются без всякого предупреждения, совершенно неожиданно. Если вы не планировали масштабирование заранее, то, скорее всего, придется немало поработать, чтобы сохранить приемлемое время реакции приложения. Компании, которые не научились масштабировать свои продукты, часто вообще сходят со сцены. Да, это печальная истина: чересчур большой успех может прикончить ваш бизнес.

Необходимо также гарантировать, что приложение при любых обстоятельствах остается работоспособным. Помех может быть много, но чаще всего проблемы вызваны обыкновенными аппаратными и программными сбоями. Приложение должно считать их нормой и по возможности обрабатывать автоматически.

Требования к масштабируемости и обеспечению высокой доступности часто идут рука об руку. Высокая доступность не так важна для маленьких приложений по нескольким причинам: обычно такое ПО работает на одном сервере, поэтому отказ сервера менее вероятен, а поскольку приложение невелико, то время простоя вряд ли обойдется слишком уж дорого. Ну и наконец, небольшая аудитория с высокой вероятностью будет готова будет смириться с простоем. Но с увеличением количества серверов в 10 раз вероятность выхода из строя сервера тоже возрастает десятикратно, да к тому же пользователей становится больше и их ожидания растут.

MySQL хорошо масштабируется, если выбрана и умело реализована правильная архитектура. То же относится и к обеспечению высокой доступности. В этой главе мы постараемся насколько возможно развести

эти понятия и рассмотреть их по отдельности. После начального обзора терминологии идут два больших раздела, посвященных отдельно масштабированию и высокой доступности (попутно мы поговорим о балансировании нагрузки). Каждая часть начинается с перечисления требований, поскольку их формулирование на раннем этапе – залог успеха при разработке больших приложений. Требования оказывают серьезное влияние на проектирование и архитектуру приложения. Затем мы перейдем к обсуждению различных методов и решений и остановимся на плюсах и минусах каждого.

Терминология

Прежде всего, необходимо четко представлять себе смысл понятий. В разговоре такие термины, как «масштабируемость» и «производительность», часто употребляют как синонимы, но на самом деле они обозначают совершенно разные вещи. Приведем свои определения основных терминов, встречающихся в этой главе.

Производительность

Способность приложения отвечать заданным требованиям, к которым может относиться, например, желаемое время реакции, пропускная способность и другие метрики, обсуждавшиеся в главе 2.

Пропускная способность

Полная нагрузка, которую может выдержать приложение. О том, что означает слово «нагрузка», мы еще поговорим в этом разделе.

Масштабируемость

Способность приложения поддерживать производительность по мере возрастания какой-то метрики (например, количества серверов). Говоря о производительности в широком смысле, мы имеем в виду сочетание пропускной способности и масштабируемости.

Доступность

Доля времени, в течение которого приложение может отвечать на запросы. Обычно оно измеряется в «девятках»; так, «пять девяток» означает, что приложение доступно 99.999% времени, то есть время простоя составляет примерно пять минут в год. Для большинства приложений это очень высокая доступность.

Отказоустойчивость

Способность приложения и системы в целом корректно справляться с отказами. Даже если система изначально проектировалась с учетом высокой доступности, отказы все равно возможны. В этом случае отказоустойчивое приложение продолжит работу, возможно, с урезанной функциональностью, а не прекратит обслуживать запросы полностью.

Ясно объяснить, что такое масштабируемость, очень сложно. Приведем аналогию.

- Производительность – это скорость, с которой может ехать автомобиль.
- Пропускная способность – это ограничение скорости и количество полос на автомагистрали.
- Масштабируемость – до какого предела можно увеличивать количество автомобилей и автомагистралей без снижения скорости транспортного потока.
- Доступность – какой процент времени автомагистраль открыта для движения.

В этой аналогии масштабируемость зависит от таких факторов, как удобство развязок, количество поломок машин и дорожно-транспортных происшествий и интенсивность перестроения из ряда в ряд, но в общем случае *не* зависит от мощности двигателей.

Другими словами, масштабируемость – это *возможность по мере надобности увеличивать пропускную способность без снижения производительности*. Ключевая фраза здесь – «способность увеличивать». Даже если архитектура MySQL допускает масштабирование, ваше приложение может вести себя по-другому. Если по какой-то причине увеличить пропускную способность трудно, то приложение не масштабируется.

Отказоустойчивость определяется способностью приложения частично функционировать при выходе из строя какого-то компонента. Отказоустойчивость – не то же самое, что самовосстанавливаемость, под которой подразумевается способность приложения восстановить или поддерживать полную функциональность в случае отказа. Отказоустойчивость часто является важным компонентом масштабируемости, поскольку этот аспект нужно учитывать на этапе проектирования приложения. Если вы разрабатываете компоненты так, что их нельзя легко отключить, или так, что они не способны продолжать работу после отказа других компонент, то сбой может вывести из строя более обширную часть приложения, чем необходимо. Для обеспечения отказоустойчивости также необходимо четко разграничивать компоненты, а этого довольно трудно достичь, если не заложить в систему с самого начала.

Если масштабируемость – это возможность увеличивать пропускную способность, а пропускная способность – это нагрузка, которую способно выдержать приложение, то можно сказать, что масштабируемость – это возможность обрабатывать все большую и большую нагрузку. На самом деле «нагрузка» – это сложное понятие, поскольку его определение зависит от конкретного приложения. Приведем несколько общеупотребительных метрик типичного сайта «социальной сети» (приложение, на примере которого удобно обсуждать многие из рассматриваемых здесь понятий).

Объем данных

Рост объема данных, накапливаемых приложением, – одна из самых типичных проблем масштабирования. Особенно это актуально для многих современных веб-приложений, которые никогда не удаляют данные. Например, сайты социальных сетей обычно не удаляют старые сообщения и комментарии.

Количество пользователей

Даже если каждый пользователь порождает небольшой объем данных, при увеличении их числа он суммируется. Помимо этого общий объем данных может расти несоизмеримо быстрее, чем количество пользователей. Кроме того, обычно, чем больше пользователей, тем больше транзакций, и зависимость между количеством транзакций и количеством пользователей тоже может быть нелинейной. Наконец, с увеличением числа пользователей обычно возрастает и сложность запросов, особенно если запросы зависят от количества связей между пользователями (количество связей ограничено сверху величиной $(N * (N-1)) / 2$, где N – число пользователей).

Активность пользователей

Не все пользователи одинаково активны, и их активность распределена неравномерно. Если внезапно активность пользователей возрастет, например, из-за появления новой функции, то нагрузка на приложение может значительно увеличиться. Активность пользователей не сводится к количеству просмотров страниц – при неизменном числе просмотров нагрузка может стать больше, если популярной окажется часть сайта, требующая большей работы для генерации страниц. Кроме того, некоторые пользователи гораздо активнее прочих: у них больше друзей, сообщений или фотографий, чем у среднего пользователя.

Размер взаимосвязанных наборов данных

Если между пользователями имеются связи, то, возможно, приложению приходится выполнять запросы или вычисления для целых групп связанных пользователей. Это сложнее, чем работать с индивидуальными пользователями и их данными. Сайты социальных сетей нередко сталкиваются с непростыми проблемами из-за особо популярных групп или пользователей, у которых много друзей.

Масштабирование MySQL

Размещение всех данных приложения на единственном экземпляре MySQL – не тот подход, который хорошо масштабируется. Рано или поздно увеличившаяся нагрузка на сервер приведет к появлению узких мест и, как следствие, к снижению производительности. Традиционно в этом случае просто покупают более мощные серверы. Такой подход называется «масштабированием по вертикали», или «вертикальным

масштабированием» (scale up). Альтернативный подход – распределить работу между несколькими компьютерами; называется «масштабированием по горизонтали» (scale out). В большинстве приложений имеются данные, которые используются редко или не используются вовсе; их можно удалить или заархивировать. Такой подход мы называем «масштабированием наоборот» (scale back), просто чтобы сохранить аналогию с другими терминами. Наконец, в некоторых СУБД поддерживается масштабирование посредством *федерации*, когда к удаленным данным можно обращаться как к локальным. В MySQL поддержка этой технологии ограничена.

Любой разработчик мечтает о таком способе масштабирования, при котором в его распоряжении имеется единая логическая база, способная содержать сколько угодно данных, обслуживать сколько угодно запросов и неограниченно расти. Часто первое, что приходит в голову, – создать «кластер», или «грид» (grid), которые могли бы прозрачно поддерживать такую архитектуру, освободив приложение от грязной работы и от необходимости знать, что на самом деле данные находятся не на одном, а на нескольких серверах. До некоторой степени такой подход реализован в технологии MySQL NDB Cluster, но он не очень хорошо работает для большинства веб-приложений и обладает рядом ограничений. Поэтому большие приложения на базе MySQL обычно масштабируются по-другому. Тем не менее в конце этой части главы мы все-таки рассмотрим масштабирование на основе кластеров.

Планирование масштабируемости

Типичный симптом плохой масштабируемости – возникновение трудностей при увеличении нагрузки. Обычно это проявляется как снижение производительности – в виде замедления обработки запросов, смещения рабочей нагрузки в сторону большего потребления процессора, а не ввода/вывода, роста конкуренции между запросами и увеличения задержки. Как правило, причина связана с увеличением сложности запросов или с тем, что какая-то часть данных или индекса, которая раньше помещалась в память, теперь не помещается. Возможно, ухудшение будет заметно лишь для некоторых видов запросов, а другие будут работать, как раньше. Например, нередко длинные и сложные запросы начинают страдать быстрее, чем более простые.

Если приложение масштабируемо, то достаточно добавить несколько серверов, и трудности с производительностью исчезнут. В противном случае вы будете заниматься только проблемами производительности, пытаться настроить серверы и т. д. То есть лечить симптомы, а не причину. Этого можно избежать, если планировать масштабируемость с самого начала.

Самая трудная часть планирования – оценить, с какой нагрузкой придется иметь дело. Абсолютно точная оценка не нужна, но желательно знать хотя бы порядок величины. Если оценка завышена, то будет зря

потрачено некоторое время на разработку, а если занижено, то возросшая нагрузка застанет вас врасплох.

Кроме того, следует приблизительно оценить темпы роста нагрузки, то есть знать, где находится «горизонт». Для некоторых приложений простенький прототип вполне может работать в течение нескольких месяцев, а за это время вы сможете привлечь инвестиции и построить более масштабируемую архитектуру. Для других текущая архитектура должна обладать запасом пропускной способности хотя бы на два года.

Планируя масштабируемость, вы должны задать себе следующие вопросы.

- Насколько полна функциональность приложения? Многие решения, которые мы собираемся предложить, усложняют реализацию некоторых возможностей. Если вы еще не реализовали какие-то базовые функции, то не исключено, что потом их будет трудно встроить в масштабированное приложение. Аналогично, не всегда легко принять правильное решение о масштабировании, не зная, как эти функции работают в действительности.
- Какова ожидаемая пиковая нагрузка? Приложение должно ее выдерживать. Что случится, если ваш сайт станет таким же популярным, как начальная страница Yahoo! News или Slashdot? Но даже если ваше приложение не является сверхпопулярным веб-сайтом, все равно оно может испытывать пиковые нагрузки. Например, для сайта интернет-магазина пиковыми являются праздничные дни, особенно те, которые традиционно посвящают шопингу, например несколько предрожденственских недель. В США выходной перед Днем матери часто становится пиком продаж в интернет-магазинах, торгующих цветами.
- Если требуется, чтобы нагрузку выдерживали все части системы, то что случится, если какая-то часть откажет? Например, если операции чтения должны быть распределены по реплицируемым подчиненным серверам, то сохранит ли система работоспособность, если один из них выйдет из строя? Не придется ли отключить часть функциональности? На случай возникновения подобных проблем можно заложить избыточную пропускную способность.

Перед тем как приступать к масштабированию

Как хорошо было бы жить в идеальном мире, где все распланировано заранее, разработчиков достаточно, бюджетных ограничений не существует и т. д. Увы, реальность не так радужна, и при масштабировании приложения приходится идти на компромиссы. В частности, не исключено, что некоторые крупные изменения в проекте придется на время отложить. Прежде чем вплотную заняться вопросами масштабирования MySQL, приведем перечень того, что можно сделать сразу, еще не предпринимая серьезных усилий.

Оптимизировать производительность

Часто существенного выигрыша можно добиться ценой сравнительно простых изменений, например правильно проиндексировав таблицы или выбрав подходящую подсистему хранения. Если производительность ограничена уже сейчас, то первым делом нужно включить и проанализировать журнал медленных запросов, чтобы выяснить, какие из них следует оптимизировать. Дополнительную информацию по этому поводу см. в разделе «Протоколирование запросов» на стр. 97.

Существует точка сокращения отдачи. После того как большая часть серьезных проблем исправлена, становится все сложнее улучшать производительность за счет оптимизации запросов. Каждая следующая оптимизация требует все больше усилий и дает все менее заметный результат, хотя сложность приложения при этом быстро возрастает.

Приобрести более мощное оборудование

Иногда модернизация имеющихся серверов или добавление новых дает неплохие результаты. Приобрести несколько лишних серверов особенно имеет смысл, когда приложение находится на ранней стадии жизненного цикла. В качестве альтернативы можно попробовать оставить приложение на одном сервере. Хотя красивый и элегантный проект, возможно, и позволит осуществить это на практике, но если для запуска такого проекта требуется месяц работы трех человек, то, пожалуй, все-таки практичнее купить дополнительные серверы. Особенно если время поджимает, а разработчиков не хватает.

Покупка дополнительного оборудования – приемлемое решение, если приложение либо невелико, либо спроектировано так, что может воспользоваться новым оборудованием. Это типичная ситуация для вновь созданных продуктов, которые еще не выросли или правильно спроектированы. Для зрелых крупномасштабных приложений приобретение оборудования может ничего не дать либо оказаться слишком дорогостоящим предприятием. Например, переход от одного сервера к трем обойдется дешево, а от 100 к 300 – совсем другое дело, дороговато. В таком случае имеет смысл направить время и усилия на выжимание всей возможной производительности из существующей системы.

Масштабирование по вертикали

Масштабирование по вертикали может работать в течение некоторого времени, но лишь до тех пор, пока масштаб приложения не превысит критическую отметку.

Первая причина – деньги. Вне зависимости от программного обеспечения в какой-то момент наращивание мощности оборудования становится неприемлемым из финансовых соображений. Существует некий диапазон аппаратных решений с оптимальным соотношением цены и про-

изводительности. За пределами этого диапазона находится в основном специализированное и, соответственно, более дорогое оборудование. Тем самым положен практический предел тому, как далеко можно зайти при масштабировании по вертикали.

Но даже если отбросить экономические доводы, то сам MySQL не слишком хорошо масштабируется по вертикали, поскольку его трудно заставить эффективно использовать несколько процессоров и дисков. Сколько именно оборудования он может задействовать с пользой, сильно зависит от характеристик рабочей нагрузки, аппаратной конфигурации и операционной системы. В качестве грубой оценки мы полагаем, что для текущих версий MySQL пределом является 8 ЦП и 14 дисков[1]. У многих возникали проблемы и при менее мощном оборудовании.

Даже если главный сервер способен эффективно использовать много процессоров, маловероятно, что вам удастся подобрать подчиненный сервер, который успевал бы реплицировать данные. Сильно загруженный главный сервер легко справляется с гораздо большим объемом работы, чем подчиненный сервер с таким же оборудованием. Связано это с тем, что поток репликации на подчиненном сервере не может толком задействовать несколько процессоров и дисков.

Да и невозможно масштабировать по вертикали до бесконечности, поскольку даже самые мощные компьютеры имеют свои пределы. Обычно приложения, работающие на одном сервере, сначала упираются в ограничения при чтении, особенно при обработке сложных запросов выборки. Такие запросы внутри MySQL функционируют в однопоточном режиме, а следовательно, используют всего один ЦП, и ни за какие деньги вы не сможете существенно повысить их производительность. Самые быстрые имеющиеся на рынке процессоры, специально предназначенные для серверов, всего-то раза в два быстрее стандартных потребительских ЦП. Заметьте, что по мере того, как данные перестают умещаться в кэше, сервер начинает ощущать нехватку памяти. Обычно это проявляется в виде интенсивного обращения к дискам, а диски – один из самых медленных компонентов современного компьютера.

Проблемой может стать и масштабируемость приложения. Свойственные конкретному приложению проектные решения, а также ограничения, обусловленные рабочей нагрузкой, могут повлиять на то, сколько оборудования удастся эффективно задействовать.

По этим причинам мы рекомендуем не планировать вертикальное масштабирование или, по крайней мере, не считать, что оно безгранично. Если вы знаете, что приложение будет расти очень быстро, то вполне допустимо купить более мощный сервер на тот непродолжительный про-

[1] Увеличение объема памяти дает больший эффект при условии, что используется 64-разрядная операционная система и оборудование. Некоторые ограничения все же есть, но они не так суровы и очевидны. Мы подробно рассматривали вопросы, относящиеся к памяти, в главе 7.

межуток времени, пока вы будете работать над другим решением. Однако в общем случае вам рано или поздно придется перейти к горизонтальному масштабированию, что и является темой следующего раздела.

Масштабирование по горизонтали

Простейший и наиболее распространенный способ масштабирования по горизонтали – распределить данные по нескольким серверам с помощью репликации, а затем выполнять чтение на подчиненных серверах. Эта техника хорошо работает в приложениях, где много запросов на чтение. У нее есть недостатки, например дублирование кэша, но даже эта проблема может оказаться не слишком серьезной, если размер данных ограничен. Мы немало говорили на данную тему в предыдущей главе и еще вернемся к ней в этой.

Еще один употребительный метод горизонтального масштабирования – *секционировать* рабочую нагрузку по нескольким «узлам». Конкретный способ секционирования требует тщательного обдумывания. Вспомните «систему-мечту», которая автоматически масштабируется невидимо и неограниченно – увы, на базе MySQL такие системы обычно не строятся. Большинство крупномасштабных приложений на основе MySQL не автоматизируют секционирование или, по крайней мере, автоматизируют не полностью. В этом разделе мы рассмотрим некоторые варианты секционирования и обсудим их сильные и слабые стороны.

Узлом (*node*) называется функциональная единица в архитектуре MySQL. Если вы заранее не планируете избыточность и высокую доступность, то узлом может быть и один сервер. Если же проектируется система с резервированием и возможностью аварийного переключения при отказе, то узлом может быть:

- Пара серверов в топологии репликации главный–главный с активным главным и пассивным подчиненным сервером
- Один главный и много подчиненных серверов
- Активный сервер, использующий распределенное реплицируемое блочное устройство (DRBD) в качестве резерва
- Кластер на базе SAN

В большинстве случаев на всех серверах внутри одного узла должны храниться одинаковые данные. Нам нравится схема с репликацией типа главный–главный из двух серверов в активном и пассивном режиме. Подробнее об этой топологии см. в разделе «Главный–главный в режиме активный–пассивный» на стр. 453.

Функциональное секционирование

Функциональное секционирование, или разделение обязанностей означает, что под разные задачи выделяются разные узлы. Мы уже встречались с несколькими подобными подходами; например, в предыдущей

главе упоминалось о назначении разных серверов для OLTP- и OLAP-запросов. Функциональное секционирование развивает эту идею, предлагая назначать отдельные сервера или узлы разным приложениям, так что в каждом узле хранятся только данные, необходимые конкретному приложению.

Здесь мы употребляем слово «приложение» расширительно. Имеется в виду не отдельная программа, а набор взаимосвязанных программ, которые можно легко отделить от другого ПО, не имеющего к ним отношения. Например, если веб-сайт включает разделы, не связанные общими данными, то можно секционировать его по функциональному назначению. Нередко можно встретить порталы, на которых представлены не связанные между собой области; из портала можно перейти в новостной раздел, в форумы, в область технической поддержки и базы знаний и т. д. Данные для каждой из этих функциональных областей могут храниться на отдельном сервере MySQL. Такая организация представлена на рис. 9.1.

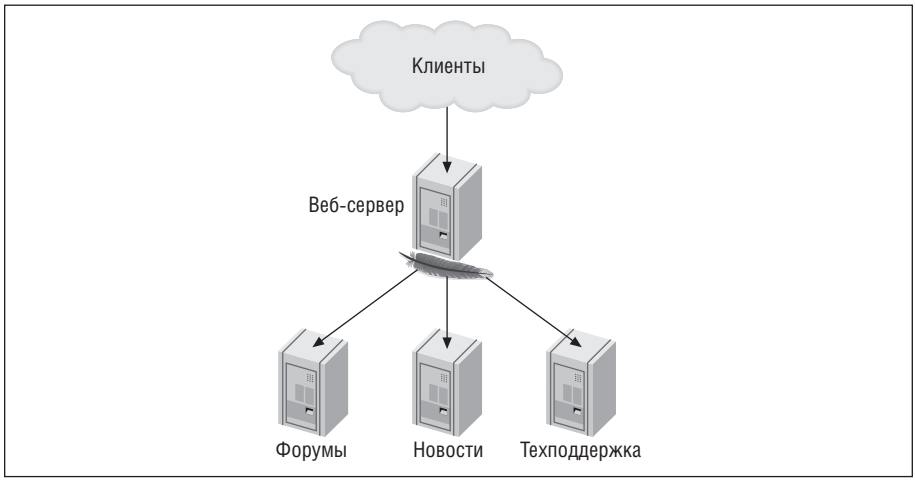


Рис. 9.1. *Портал и узлы, выделенные под различные функциональные области*

Если приложение очень велико, то под каждую функциональную область можно отвести свой веб-сервер, хотя так делают реже.

Еще один подход к функциональному секционированию заключается в том, чтобы разбить данные одного приложения на множество таблиц, которые никогда не соединяются друг с другом. В случае необходимости можно иногда соединять таблицы из разных множеств либо на прикладном уровне, либо с помощью подсистемы хранения Federated, если не предъявляется жестких требований к производительности. У этой идеи есть несколько вариаций, но общим для них является то, что данные каждого типа находятся на одном узле. Такой способ секциониро-

вания применяется не часто, поскольку его очень трудно реализовать эффективно, а существенных преимуществ по сравнению с другими методами он не дает.

Однако и возможности функционального секционирования небезграничны, так как если некоторая область связана с одним узлом MySQL, то она должна масштабироваться по вертикали. Рано или поздно какое-нибудь приложение или функциональная область разрастется слишком сильно, и тогда придется искать другую стратегию. А если зайти по этому пути слишком далеко, то впоследствии изменить архитектуру на более масштабируемую может оказаться сложно.

Секционирование данных

Секционирование данных (data sharding[1]) на сегодня является наиболее распространенным и успешным подходом к масштабированию очень больших приложений на базе MySQL. Для этого данные разбиваются на меньшие куски, или секции (shards) и хранятся на разных узлах.

Секционирование данных хорошо сочетается с некоторыми видами функционального секционирования. В большинстве подобных систем есть также некоторые «глобальные» значения, которые не секционируются вовсе (скажем, списки городов). Обычно такие данные хранятся на одном узле, доступ к которому часто организуется через специальный кэш, например *memcached*.

На практике в большинстве приложений секционируется только информация, которая в этом нуждается, – как правило, те части набора данных, которые будут расти особенно быстро. Предположим, что вы создаете службу блогов. Если ожидается 10 миллионов пользователей, то, быть может, секционировать информацию о них и необязательно, так как вся она (или, по крайней мере, относящаяся к активным пользователям) помещается целиком в памяти. Если же ожидается 500 миллионов пользователей, то секционировать их профили имеет смысл. Генерируемый пользователями контент – сообщения и комментарии – почти наверняка придется секционировать в любом случае, поскольку эти записи гораздо длиннее и их намного больше.

В больших приложениях может быть несколько логических наборов данных, которые можно секционировать по-разному. Их можно хранить на разных серверах, хотя это и необязательно. Кроме того, одни и те же данные можно секционировать разными способами в зависимости от того, как организован доступ к ним. Пример такого подхода будет приведен ниже.

Проектирование приложений, построенных на основе секционирования данных, принципиально отличается от того, как обычно строят

[1] Употребляются также термины «splintering» и «partitioning», но Google называет эту технологию именно «sharding».

приложения на начальном этапе. Переход от монолитного хранилища данных к секционированной архитектуре может оказаться весьма непростым делом. Поэтому гораздо проще с самого начала закладывать идеологию секционирования, если есть основания полагать, что это понадобится в будущем.

Большинство приложений, в которые секционирование не было встроено изначально, впоследствии, по мере роста, проходят через промежуточные этапы. Например, в службе блогов можно поначалу воспользоваться репликацией для масштабирования запросов на чтение, но в конце концов это решение перестанет работать. Тогда ничто не мешает разбить службу на три части: пользователи, сообщения и комментарии. Их можно поместить на разные серверы (функциональное секционирование), а соединение таблиц выполнять на уровне приложения. На рис. 9.2 показан постепенный переход от одного сервера к функциональному секционированию.

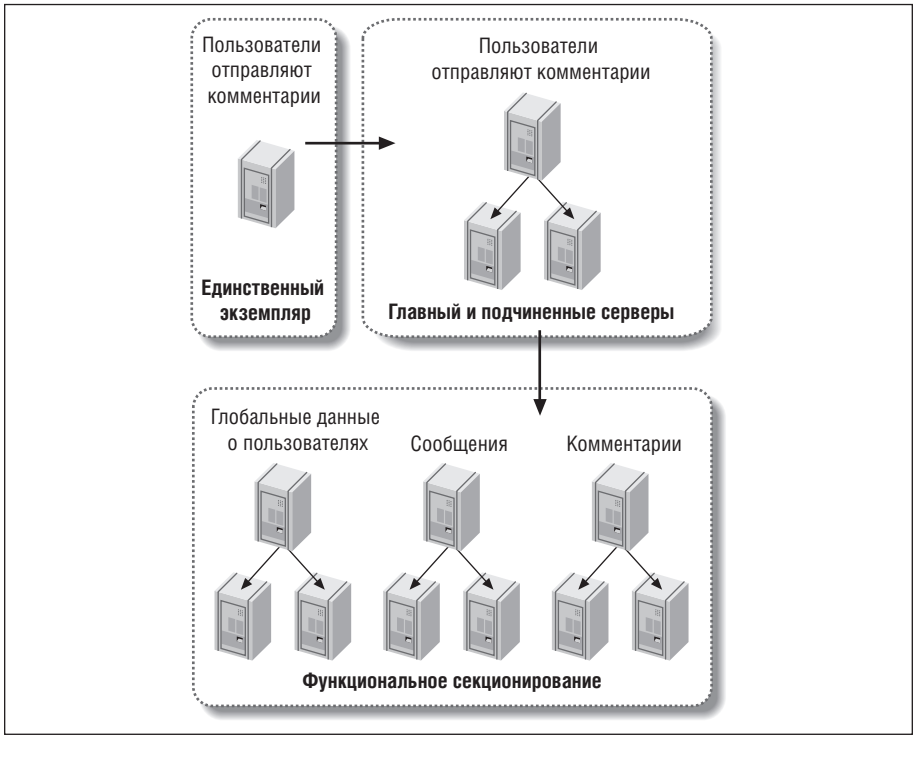


Рис. 9.2. *От одного экземпляра к функционально секционированному хранилищу данных*

Наконец, сообщения и комментарии можно секционировать по идентификатору пользователя, а информацию о пользователях хранить на отдельном узле. Если применять конфигурацию главный–подчиненный

для узла с глобальными данными и пары главный–главный для узлов с секционированными данными, то окончательно хранилище примет вид, показанный на рис. 9.3.

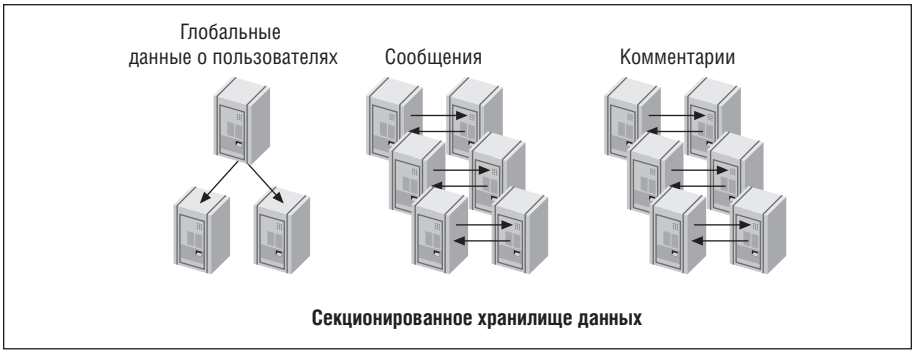


Рис. 9.3. *Хранилище данных с одним глобальным узлом и шестью парами главный–главный*

Если заранее известно, что потребуется масштабировать очень большую систему, и понятны ограничения функционального секционирования, то можно пропустить промежуточные этапы и сразу перейти от единственного узла к секционированному хранилищу.

В секционированных приложениях часто применяется та или иная библиотека абстрагирования базы данных, которая упрощает взаимодействие между программой и секционированным хранилищем. Обычно такие библиотеки не скрывают секционирование полностью, поскольку часто приложение знает о запросе нечто такое, что неизвестно хранилищу данных.

Слишком высокий уровень абстракции может порождать неэффективность, например опрос всех узлов в поисках информации, которая заведомо хранится только на одном узле. Это одна из причин, по которым подсистема хранения NDB Cluster часто плохо работает в веб-приложениях: она скрывает тот факт, что должна опрашивать много узлов, и представляет весь кластер как единственный сервер.

Секционированное хранилище данных может показаться элегантным решением, но его трудно реализовать. Так почему же мы голосуем за такую архитектуру? Ответ прост: если вы хотите масштабировать операции записи, то просто *обязаны* секционировать данные. Имея единственный главный сервер, невозможно масштабировать запись, сколько бы ни было подчиненных серверов. Секционирование данных при всех своих недостатках – предпочтительное, на наш взгляд, решение этой проблемы.

Полностью автоматический, высокопроизводительный, прозрачный способ секционировать информацию, так чтобы извне казалось, будто она находится на одном сервере, был бы чудесным подарком, но пока

такого не существует. Возможно, в будущем подсистема хранения NDB Cluster станет настолько быстрой и надежной, что ее удастся использовать для этой цели.

Выбор ключа секционирования

Самая важная и сложная задача, возникающая при секционировании информации, – поиск и выборка данных. Способ поиска зависит от того, как они секционированы. Сделать это можно разными методами – одни получше, другие похуже.

Наша цель состоит в том, чтобы самые важные и часто встречающиеся запросы затрагивали как можно меньше секций. В этом плане очень важно выбрать *ключ* (или ключи) *секционирования* данных. Ключ секционирования определяет, в какую секцию попадает та или иная строка. Если известен ключ секционирования некоторого объекта, то можно ответить на два вопроса:

- Где следует сохранить данные?
- Где найти запрошенные данные?

Ниже мы покажем разные способы выбора и использования ключа секционирования. А пока рассмотрим пример. Допустим, что мы поступаем так же, как NDB Cluster, то есть для распределения данных по секциям применяем хеш первичного ключа таблицы. Это очень простой подход, но он плохо масштабируется, поскольку часто заставляет вас искать информацию во всех секциях. Пусть, например, нужно найти все сообщения в блоге пользователя 3. Где их искать? Вероятнее всего, они равномерно распределены по всем секциям, так как секционирование производилось по первичному ключу, а не по идентификатору пользователя.

Запросы к нескольким секциям хуже, чем запросы к одной из них, но если они затрагивают не слишком много секций, то все еще не так плохо. Самый худший случай – когда у вас нет ни малейшего представления о том, где находятся данные, и приходится просматривать все секции без исключения.

Обычно хорошим ключом секционирования является идентификатор какой-нибудь важной сущности в базе данных. Такие сущности называются *единицами секционирования* (*unit of sharding*). Например, если информация секционируется по идентификатору пользователя или клиента, то единицей секционирования является соответственно пользователь или клиент.

Для начала очень неплохо изобразить модель данных в виде диаграммы сущность–связь или эквивалентного представления, на котором видны все сущности и связи между ними. Постарайтесь нарисовать диаграмму так, чтобы взаимосвязанные сущности располагались близко друг к другу. Часто можно с помощью визуального изучения диаграммы найти таких кандидатов на роль ключа секционирования, которых

иначе бы вы не заметили. Но не ограничивайтесь одной лишь диаграммой, примите также во внимание запросы, выполняемые приложением. Даже если между двумя сущностями существует какая-то связь, но соединение по ней производится редко или вообще никогда, то при реализации секционирования эту связь можно не принимать во внимание.

Одни модели данных лучше поддаются секционированию, чем другие, все зависит от степени связности графа сущность–связь. На рис. 9.4 слева изображена модель, которая легко секционируется, а справа – секционируемая с трудом.

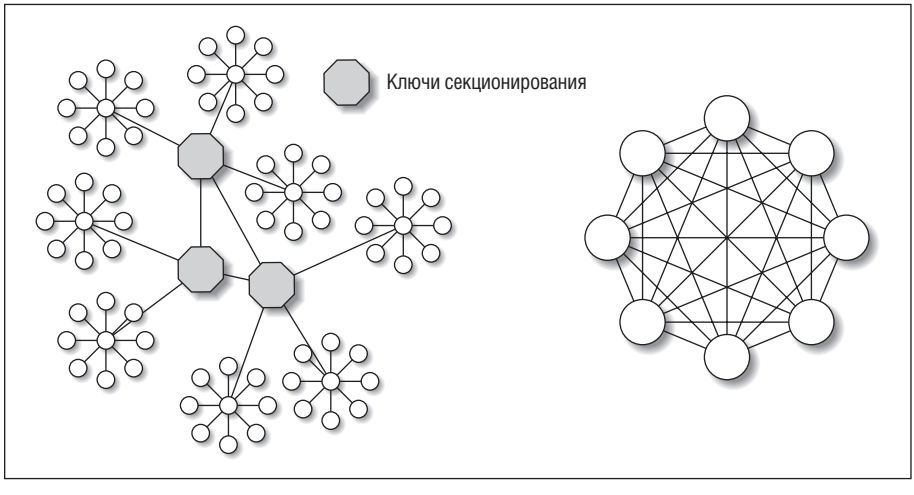


Рис. 9.4. *Две модели данных – одна секционируется легко, другая с трудом. (Спасибо проекту HiveDB и Бритту Кроуфорду за эти красивые диаграммы.)*

Левую модель данных легко секционировать, потому что в ней много связных подграфов, состоящих в основном из узлов, между которыми есть всего одна связь, и можно сравнительно легко «разрезать» связи между подграфами. Правую модель трудно секционировать, потому что в ней таких подграфов нет. Большинство моделей данных представляются скорее левой, чем правой диаграммой.

Несколько ключей секционирования

Если модель данных сложна, то и секционировать ее труднее. Во многих приложениях существует более одного ключа секционирования, особенно если в данных можно выделить несколько важных «измерений». Иными словами, приложение должно иметь возможность эффективно взглянуть на информацию под разными углами зрения и получить целостное представление. Это означает, что некоторые данные придется хранить в двух разных видах.

Например, данные приложения для ведения блогов можно секционировать по идентификатору пользователя и по идентификатору сообщения,

поскольку то и другое – осмысленные способы взглянуть на данные. Действительно, часто вы хотите видеть все сообщения одного пользователя и все комментарии к сообщению. Но секционирование по пользователям не поможет в поиске комментариев к сообщению, а секционирование по сообщениям бесполезно для поиска сообщений данного пользователя. Если вы хотите, чтобы оба запроса затрагивали только одну секцию, то придется секционировать обоими способами.

Тот факт, что необходимо несколько ключей секционирования, еще не означает, что требуется проектировать два полностью дублирующих друг друга хранилища данных. Рассмотрим еще один пример: сайт социальной сети книголюбов, на котором пользователи могут оставлять свои комментарии о книгах. На таком сайте могут отображаться все комментарии к некоторой книге, а также все книги, которые прочитал и прокомментировал данный пользователь.

В таком случае можно построить одно секционированное хранилище с данными о пользователях и второе – с данными о книгах. В комментариях хранится как идентификатор пользователя, так и идентификатор книги, поэтому они пересекают границы секций. Вместо того чтобы полностью дублировать комментарии, можно хранить их вместе с данными о пользователях. Тогда в секции данных о книгах достаточно хранить только заголовок и идентификатор комментария. При этом для построения *большинства* страниц, содержащих комментарии к книгам, не придется обращаться к обоим секциям, ну а если потребуется вывести полный текст комментария, то его можно будет достать из хранилища данных о пользователях.

Межсекционные запросы

В большинстве секционированных приложений встречаются запросы, требующие агрегирования или соединения данных, хранящихся в разных секциях. Например, если на сайте книголюбов нужно показать наиболее активных или наиболее популярных пользователей, то по определению необходимо обращаться ко всем секциям. Оптимизация подобных запросов – самая сложная часть реализации секционирования, поскольку то, что приложение видит как один запрос, на самом деле надо расщепить и параллельно выполнить в каждой секции. Хорошо написанный слой абстрагирования базы данных может в какой-то мере облегчить решение этой задачи, но все равно такие запросы выполняются гораздо медленнее и обходятся настолько дорого по сравнению с запросами к одной секции, что обычно необходимо применять агрессивное кэширование.

Некоторые языки, в частности PHP, плохо поддерживают параллельное выполнение нескольких запросов. Чтобы обойти эту трудность, обычно разрабатывают вспомогательное приложение, часто на языке C или Java, которое отправляет запросы и объединяет результаты. Тог-

да PHP-страница может обратиться к этому приложению, обычно представляющему собой веб-сервис.

Запросы к нескольким секциям можно также ускорить за счет наличия сводных таблиц. Для их заполнения нужно обойти все секции и по завершении сохранить данные также на серверах каждой секции, чтобы обеспечить резервирование. Если дублирование информации на каждой секции слишком расточительно, то можно собрать все сводные таблицы в каком-нибудь специальном хранилище. Несекционированные данные часто размещаются на «глобальном» узле, а для защиты его от перегрузки активно применяют кэширование.

В некоторых приложениях используется случайное секционирование в тех случаях, когда важно равномерное распределение данных или подходящего ключа секционирования не существует. Хорошим примером могут служить распределенные программы поиска, где межсекционные запросы и агрегирование являются нормой, а не исключением.

Межсекционные запросы – не единственная сложность, встречающаяся при организации секционирования. Трудно также поддерживать согласованность данных. Внешние ключи через границы секций не работают, поэтому обычно контроль производится на уровне приложения. Можно обратиться к XA-транзакциям, но из-за высоких накладных расходов так поступают редко. Дополнительную информацию см. в разделе «Распределенные (XA) транзакции» на стр. 329.

Можно также спроектировать процесс очистки, который будет запускаться время от времени. Например, если срок хранения учетной записи в клубе книголюбов истек, то необязательно удалять ее немедленно. Можно написать периодическое задание, которое удалит комментарии пользователя из секции с данными о книгах. Можно также подготовить сценарий, который будет периодически запускаться и проверять согласованность информации в разных секциях.

Распределение данных по секциям и узлам

Между секциями и узлами не обязательно должно существовать взаимно однозначное соответствие. Часто бывает разумнее сделать размер секции значительно меньше, чем емкость узла, так чтобы в одном узле можно было хранить несколько секций.

Чем меньше размер секции, тем проще обслуживать находящиеся в ней данные. Упрощается резервное копирование и восстановление информации, а кроме того, для небольших таблиц быстрее завершаются такие действия, как изменение схемы. Пусть, например, имеется таблица размером 100 Гбайт. Ее можно хранить целиком в одном месте или разбить на 100 секций по 1 Гбайт, разместив их все в одном узле. Теперь предположим, что над этой таблицей нужно построить новый индекс. На 100-гигабайтной секции это заняло бы гораздо больше времени, чем совокупное индексирование гигабайтных кусочков, поскольку секция размером 1 Гбайт целиком помещается в память. К тому же во время

работы команды ALTER TABLE данные недоступны, так уж лучше заблокировать 1 Гбайт, а не все 100.

Также секции меньшего размера удобнее перемещать, что упрощает задачу перераспределения емкости и поиска оптимального баланса между секциями и узлами. Вообще говоря, перемещения секции – не слишком эффективная процедура. Обычно нужно перевести секцию в режим чтения (эту функцию необходимо встроить в приложение), выгрузить данные и переместить их в другой узел. Для выгрузки чаще всего используется программа *mysqldump*, а для загрузки – *mysql* (таблицы типа MyISAM можно переносить простым копированием файлов; см. главу 11).

Помимо перемещения секций между узлами, нужно еще подумать о перемещении данных между секциями, желательно, не прекращая обслуживания пользователей. Если секции слишком велики, то балансировать емкость путем перемещения секций целиком будет труднее, так что придется придумать способ переноса отдельных логических блоков данных (например, данных об одном пользователе) из одной секции в другую. Перемещение данных между секциями обычно намного сложнее, чем перемещение самих секций, поэтому по возможности старайтесь этого избегать. Вот почему мы и рекомендуем ограничивать размер секции так, чтобы он поддавался управлению.

Относительные размеры секций зависят от нужд приложения. Определяя, что такое «поддающийся управлению размер», мы обычно предлагаем такую грубую оценку – размер, при котором таблицы настолько малы, что для регулярного выполнения таких задач обслуживания, как ALTER TABLE, CHECK TABLE, OPTIMIZE TABLE, достаточно 5–10 минут.

Если размер секций слишком мал, то может появиться чересчур много таблиц, что негативно отразится на файловой системе или на внутренних структурах MySQL (см. раздел «Кэш таблиц» в главе 6 на стр. 348). К тому же, чем меньше секции, тем больше придется выполнять межсекционных запросов.

Организация секций в узлах

Необходимо решить, как организовывать секции на одном узле. Есть несколько распространенных методов:

- Размещать на каждом узле одну базу данных и называть все эти базы одинаково. Такой метод обычно применяется, когда хотят, чтобы каждая секция повторяла структуру исходного приложения. Он неплохо работает, если запускается несколько экземпляров приложения, каждый из которых знает только об одной секции.
- Помещать таблицы из нескольких секций в одну базу данных и включать номер секции в имя каждой таблицы (например, bookclub.comments_23). При такой конфигурации в одной базе данных может храниться несколько секций.

- Использовать одну базу данных на каждую секцию и включать в эту базу все таблицы приложения. Номер секции тогда является частью имени базы, а не таблицы (то есть полное имя таблицы имеет вид `bookclub_23.comments`, `bookclub_23.users` и т. д.). Такой метод применяется, когда приложение соединяется с одной базой данных, а имя базы в запросах не указывается. Преимущество состоит в том, что запросы не нужно подстраивать под каждую секцию. За счет этого упрощается переход к секционированной архитектуре для приложений, работающих только с одной базой.
- Использовать одну базу данных на каждую секцию и включать номер секции как в имя базы, так и в имена таблиц (то есть полное имя таблицы имеет вид `bookclub_23.comments _23`).

Если номер секции является частью имен таблицы, то нужен какой-то способ подставить его в шаблон запроса. Как правило, в шаблон включают специальную последовательность, например спецификаторы в духе функции `sprintf` (скажем, %s) или строковую интерполяцию с помощью переменных. Вот один из способов записать шаблон запроса в PHP:

```
$sql = "SELECT book_id, book_title FROM bookclub_%d.comments_%d... ";
$res = mysql_query(sprintf($sql, $shardno, $shardno), $conn);
```

А вот как можно применить строковую интерполяцию:

```
$sql = "SELECT book_id, book_title FROM bookclub_$shardno.comments_$shardno ...";
$res = mysql_query($sql, $conn);
```

Эта идея легко реализуется в новых приложениях, но добавить шаблоны в существующую программу сложнее. При создании нового ПО, когда с шаблонами запросов проблем не возникает, мы предпочитаем заводить по одной базе на секцию, включая номер секции в имя базы и имена таблиц. Хотя такое решение несколько усложняет задачи типа включения команды `ALTER TABLE` в сценарии, у него есть и преимущества.

- Если секция целиком находится в одной базе данных, то ее легко переместить с помощью программы *mysqldump*.
- Поскольку база данных представляет собой каталог в файловой системе, легко манипулировать файлами секции.
- Если одна секция не перемешана с другими, то ее размер легко определяется.
- Глобально уникальные имена таблиц помогают избежать ошибок. Если имена таблиц, находящихся в разных местах, совпадают, то можно случайно обратиться не к той секции, установив соединение не с тем узлом, что надо, или импортировать данные, предназначенные одной секции, в таблицу, принадлежащую другой.

Имеет смысл задаться вопросом, не характерна ли для данных приложения *«близость секций»* (*shard affinity*). Возможно, удастся получить выигрыш, поместив некоторые секции «недалеко» друг от друга (на одном

сервере, в одной подсети, в одном центре обработки данных или на узлах сети, обслуживаемых одним коммутатором), чтобы воспользоваться сходством типичных способов доступа к данным. Например, можно секционировать информацию по пользователям и поместить всех пользователей из одной страны в секции, находящиеся на одних и тех же узлах.

Когда поддержка секций добавляется в существующее приложение, часто предпочитают размещать по одной секции на каждом узле. При таком упрощенном подходе минимизируется объем изменений в коде запросов, выполняемых приложением. С введением секционирования приложение и так придется серьезно переработать, а лишние проблемы никому не нужны. Если секционирование производится так, что в каждом узле оказывается уменьшенная копия всех данных приложения, то большую часть запросов изменять не придется, как не придется и задумываться о том, каким образом маршрутизировать запросы нужному узлу.

Фиксированное распределение

Существует два основных способа распределения данных по секциям: *фиксированное* и *динамическое*. Для обеих стратегий необходима функция секционирования, которая принимает на входе ключ секционирования строки и возвращает номер секции, в которой эта строка находится[1].

Для фиксированного распределения применяется функция разбиения, которая зависит только от значения ключа секционирования. В качестве примеров можно привести деление по модулю или хеш-функции. Такие функции отображают значения ключей секционирования на конечное число «ячеек» (buckets), в которых хранятся данные.

Пусть имеется всего 100 ячеек и требуется найти, в какую из них поместить пользователя 111. С применением деления по модулю ответ ясен: остаток от деления 111 на 100 равен 11, поэтому пользователь должен находиться в секции 11.

Если же для хеширования задействована функция `CRC32()`, то в результате получается 81:

```
mysql> SELECT CRC32(111) % 100;
+------------------+
| CRC32(111) % 100 |
+------------------+
|               81 |
+------------------+
```

[1] Здесь слово «функция» употребляется в математическом смысле как отображение множества входных значений (области определения) во множество выходных значений (область значений). Как будет показано, такую функцию можно реализовать разными способами, в том числе с использованием справочной таблицы в базе данных.

Основные достоинства фиксированного распределения – простота и низкие накладные расходы. К тому же ее легко встроить в приложение.

Однако у этой стратегии есть и недостатки.

- Если секции велики и их немного, то балансировать нагрузку между ними будет сложно.
- При фиксированном распределении вы лишены возможности решать, куда помещать конкретную запись, а это важно в приложениях, где нагрузка на единицы секционирования неравномерна. Некоторые части данных используются гораздо чаще, чем остальные, и когда такие обращения по большей части адресуются к одной и той же секции, стратегия фиксированного распределения не позволяет снять нагрузку, переместив часть данных в другую секцию. Эта проблема не так серьезна, если элементы данных малы, но их количество в каждой секции велико; закон больших чисел все расставит по своим местам.
- Как правило, изменить алгоритм секционирования сложнее, потому что требуется перераспределить все существующие данные. Например, если секционирование производилось делением по модулю 10, то имеется 10 секций. Когда приложение вырастет, и секции станут слишком большими, возникнет желание увеличить их количество до 20. Но для этого придется перехешировать все заново, обновить очень много данных и перераспределить их по новым секциям.

В силу указанных ограничений мы обычно предпочитаем в новых приложениях пользоваться динамическим распределением. Но при добавлении секционирования в существующее приложение бывает проще применить фиксированную стратегию.

Иногда мы применяем фиксированное распределение и в новых проектах. В частности, оно неплохо зарекомендовало себя на сайте BoardReader (*http://www.boardreader.com*) – системе поиска по форумам, написанной одним из авторов настоящей книги. На этом сайте индексируется очень большой объем данных. У нас возникало искушение секционировать форумы по хешу идентификатора сайта. При этом все форумы на одном сайте попали бы в одну секцию, что было бы прекрасно с точки зрения запросов, которые обращаются к данным из нескольких форумов на одном сайте, например для запроса, который ищет самые популярные форумы. Однако на некоторых сайтах размещаются тысячи форумов с десятками или сотнями миллионов сообщений. Такие секции оказались бы слишком велики для обслуживания, поэтому мы решили остановиться на секционировании по хешу идентификатора форума.

Динамическое распределение

Альтернативой фиксированному распределению является динамическое распределение, описание которого хранится отдельно в виде ото-

бражения единицы секционирования на номер секции. Примером может служить таблица с двумя столбцами: идентификатор пользователя и идентификатор секции:

```
CREATE TABLE user_to_shard (
  user_id INT NOT NULL,
  shard_id INT NOT NULL,
  PRIMARY KEY (user_id)
);
```

Функцией разбиения служит сама таблица. Зная ключ секционирования (идентификатор пользователя), можно найти соответствующий номер секции. Если подходящей строки не существует, можно выбрать нужную секцию и добавить строку в таблицу. Впоследствии сопоставление можно будет изменить, потому стратегия и называется динамической.

С динамическим распределением связаны определенные накладные расходы, так как требуется обращение к внешнему ресурсу, например серверу каталогов (узлу, на котором хранится отображение). Для поддержания приемлемой эффективности такая архитектура часто нуждается в дополнительных слоях программного обеспечения. Например, можно использовать распределенную систему кэширования, в памяти которой хранятся данные с сервера каталогов, которые на практике изменяются довольно редко.

Основное достоинство динамического распределения – более точное управление местом хранения данных. Это упрощает равномерное разделение данных по секциям и позволяет гибко адаптироваться к непредвиденным изменениям.

Кроме того, динамическое распределение дает возможность выстраивать многоуровневую стратегию секционирования поверх простого отображения ключей на секции. Например, можно организовать двойное отображение, при котором каждой единице секционирования сопоставляется некоторая группа (например, группа членов клуба книголюбов), а сами группы по возможности размещаются в одной секции. Это позволяет воспользоваться преимуществами близости секций и избегать межсекционных запросов.

Динамическое распределение дает возможность создавать несбалансированные секции. Это полезно, когда не все сервера одинаково мощные или когда некоторые серверы используются еще и для других целей, например для архивирования данных. Если при этом имеется еще и возможность в любой момент перебалансировать секции, то можно поддерживать взаимно однозначное соответствие между секциями и узлами, не растрачивая впустую емкость дисков. Некоторым нравится та простота, которая свойственна хранению в каждом узле ровно одной секции (но не забывайте, что имеет смысл делать секции относительно небольшими).

Динамическое распределение и толковое применение близости секций может предотвратить рост межсекционных запросов по мере масштабирования. Представьте себе межсекционный запрос к хранилищу данных с четырьмя узлами. При фиксированном распределении может случиться так, что любой запрос затрагивает все секции, тогда как динамическое распределение позволит сделать возможным запуск всего на трех узлах. Вроде бы разница невелика, но подумайте, что произойдет, когда количество секций возрастет до 400: фиксированное распределение заставляет опрашивать каждую из 400 секций, а динамическое – все те же три.

Динамическое распределение позволяет организовать стратегию секционирования любой сложности, фиксированное такого богатства выбора не предоставляет.

Комбинирование динамического и фиксированного распределений

Можно также применять комбинацию динамического и фиксированного распределения. Это часто бывает полезно, а иногда даже необходимо. Динамическое распределение хорошо работает, когда отображение не слишком велико. С ростом количества единиц секционирования его эффективность падает.

В качестве примера рассмотрим систему, в которой хранятся ссылки между сайтами. В ней необходимо хранить десятки миллиардов строк, а ключом секционирования является комбинация исходного и конечного URL. На любой из двух URL могут быть сотни миллионов ссылок, поэтому ни один URL по отдельности не является достаточно избирательным. Однако невозможно хранить все комбинации исходного и конечного URL в таблице сопоставления, поскольку их слишком много, а каждый URL занимает много места.

Одно из возможных решений – конкатенировать URL и написать хеш-функцию, отображающую получившиеся строки на фиксированное число ячеек, которые затем можно динамически отображать на секции. Если количество ячеек достаточно велико, скажем, миллион, то в каждую секцию их можно поместить довольно много. В результате мы получаем многие преимущества динамического секционирования, не заводя гигантскую таблицу отображения.

Явное распределение

Третья стратегия позволяет приложению явно выбрать секцию при создании строки. Если данные уже существуют, то сделать это довольно трудно, поэтому такое решение редко применяется при переработке имеющегося приложения с учетом секционирования. И все-таки иногда оно бывает полезно.

Идея в том, чтобы закодировать номер секции в идентификаторе: похожая техника применяется, чтобы избежать дублирования значений

ключей в репликации типа главный–главный (см. раздел «Запись на обоих главных серверах в конфигурации главный–главный» на стр. 492).

Пусть, например, приложение хочет создать пользователя 3 и назначить ему секцию 11, и для номера секции отведено 8 старших битов в столбце типа BIGINT. Тогда пользователю будет присвоен идентификатор (11 << 56) + 3, или 792633534417207299. Впоследствии приложение без труда сможет извлечь из этого числа и номер пользователя, и номер секции. Например:

```
mysql> SELECT (792633534417207299 >> 56) AS shard_id,
    -> 792633534417207299 & ~(11 << 56) AS user_id;
+----------+---------+
| shard_id | user_id |
+----------+---------+
|       11 |       3 |
+----------+---------+
```

Предположим далее, что требуется создать для этого пользователя комментарий и сохранить его в той же самой секции. Приложение может присвоить комментарию номер 5 и точно так же, как выше, объединить его с номером секции – 11.

Достоинство такого подхода заключается в том, что идентификатор каждого объекта уже несет в себе собственный ключ секционирования, тогда как остальные решения подразумевают соединение таблиц или иную операцию поиска для нахождения такого ключа. Если вы хотите извлечь из базы конкретный комментарий, то вовсе не обязательно знать, какому пользователю он принадлежит; сам идентификатор объекта скажет, где его искать. Если бы объекты динамически секционировались по идентификатору пользователя, то пришлось бы сначала найти владельца комментария, а затем запросить у сервера каталогов, в какой секции этот пользователь находится.

Еще одно решение – хранить ключ секционирования вместе с объектом, но в разных столбцах. Например, вы никогда не обращаетесь просто к комментарию с номером 5, а всегда к комментарию 5, принадлежащему пользователю 3. Возможно, кому-то такой подход понравится больше, поскольку он не нарушает первую нормальную форму, однако дополнительные столбцы требуют лишних накладных расходов, написания лишнего кода и вообще вызывают всяческие неудобства. (Это тот случай, когда, на наш взгляд, хранение двух значений в одном столбце оправдано.)

Недостаток явного распределения кроется в том, что секционирование фиксировано, и перебалансировать секции становится сложнее. С другой стороны, этот метод хорошо работает в сочетании с комбинацией фиксированного и динамического распределения. Вместо того чтобы хешировать в заранее заданное количество ячеек и затем отображать ячейки на узлы, ячека просто кодируется в самом объекте. Это позво-

ляет приложению управлять местонахождением данных и, в частности, помещать взаимосвязанные значения в одну и ту же секцию.

На сайте BoardReader применяется вариант этой техники: ключ секционирования закодирован в идентификаторе документа в системе Sphinx. Поэтому становится просто найти данные, ассоциированные с каждым результатом поиска, в секционированном хранилище данных. Дополнительную информацию о системе Sphinx см. в приложении А.

Мы включили описание комбинированного распределения, поскольку встречали случаи, когда оно полезно, но, вообще говоря, не рекомендуем применять этот подход. Мы предпочитаем по возможности использовать динамическое распределение и избегать явного.

Перебалансирование секций

При необходимости можно переместить данные из одной секции в другую, чтобы сбалансировать нагрузку. Например, многие читатели, наверное, слышали, как разработчики больших сайтов фотогалерей или популярных социальных сетей рассказывали о своих инструментах перемещения пользователей в другие секции.

Способность перемещать данные открывает целый ряд возможностей. Например, чтобы модернизировать оборудование, можно перенести всех пользователей из старой секции в новую, не останавливая секцию целиком и не переводя ее в режим чтения.

Однако мы стараемся по возможности избегать перебалансирования, так как это может вызвать приостановку обслуживания. Из-за перемещения данных становится сложнее добавлять в приложение новые функции, поскольку их приходится учитывать в сценариях перебалансирования. Если секции не слишком велики, то прибегать к этому, возможно, и не понадобится; часто для балансирования нагрузки достаточно перенести секцию целиком, а это гораздо проще, чем перемещать часть секции (и более эффективно, если говорить о стоимости в расчете на одну строку данных).

Одна из возможных стратегий, хорошо зарекомендовавшая себя, – распределять новые данные по секциям случайным образом. Когда секция достаточно заполнена, можно выставить флаг, который говорит приложению, что больше туда не нужно добавлять информацию. Если в будущем вы захотите снова открыть секцию для записи, то флаг можно будет сбросить.

Предположим, что установлен новый узел MySQL и на нем размещено 100 секций. В самом начале флаг каждой секции равен 1, то есть они открыты для записи. Как только в секции накопится достаточно данных (скажем, 10 000 пользователей), ее флаг сбрасывается в 0. Если по прошествии времени узел окажется недогруженным из-за того, что владельцы забросили свои учетные записи, некоторые секции можно будет заново открыть для добавления новых пользователей.

Если в ходе развития приложения были добавлены новые функции, повысившие нагрузку на каждую секцию, или если вы просто просчитались при оценке нагрузки, то некоторые секции можно будет перенести на новые узлы. Неприятность заключается в том, что на время выполнения этой операции вся секция должна быть переведена в режим чтения или вообще выведена из оперативного доступа. Вам и вашим пользователям решать, насколько такое решение приемлемо.

Генерация глобально уникальных идентификаторов

По ходу переработки системы под секционированное хранилище данных зачастую необходимо генерировать глобально уникальные идентификаторы на разных машинах. В монолитной базе для этой цели обычно используют автоинкрементные столбцы, но по умолчанию модификатор `AUTO_INCREMENT` предназначен для использования на одном сервере, где гарантировать уникальность несложно.

Есть несколько способов решения этой проблемы:

Использовать конфигурационные параметры `auto_increment_increment` *и* `auto_increment_offset`

Они говорят серверу MySQL, на какую величину увеличивать автоинкрементный столбец и с какого значения начинать нумерацию. Например, в простейшем случае, когда есть всего два сервера, можно сконфигурировать их так, что первый начнет нумерацию с 1, а второй – с 2 (впрочем, значение 0 тоже годится), а увеличивать счетчик каждый сервер будет на 2. Тогда на одном сервере в этом столбце будут только четные числа, а на другом – только нечетные. Эти параметры применяются ко всем таблицам на данном сервере.

Из-за своей простоты и независимости от какого-либо центрального узла этот способ генерации значений стал очень популярен, но он требует внимания при конфигурировании серверов. Очень легко случайно настроить их так, что будут генерироваться дубликаты, особенно если роль сервера меняется вследствие добавления новых серверов или в ходе восстановления после сбоя.

Создать таблицу на глобальном узле

Можно создать таблицу с автоинкрементным столбцом в глобальной базе данных и обращаться к ней из приложений для генерации уникальных чисел.

Использовать сервер memcached

В API сервера *memcached* имеется функция `incr()`, позволяющая атомарно инкрементировать число и вернуть результат.

Выделять значения сериями

Приложение может запросить у глобального узла серию чисел, использовать ее, а затем запросить новую.

Использовать комбинацию значений

Чтобы добиться уникальности идентификаторов на каждом сервере, можно использовать комбинацию значений, например, номер секции и увеличивающееся число. Эта техника обсуждалась в предыдущем разделе.

Использовать автоинкрементные ключи по двум столбцам

Это работает только для таблиц типа MyISAM:

```
mysql> CREATE TABLE inc_test(
    ->   a INT NOT NULL,
    ->   b INT NOT NULL AUTO_INCREMENT,
    ->   PRIMARY KEY(a, b)
    -> ) ENGINE=MyISAM;
mysql> INSERT INTO inc_test(a) VALUES(1), (1), (2), (2);
mysql> SELECT * FROM inc_test;
+---+---+
| a | b |
+---+---+
| 1 | 1 |
| 1 | 2 |
| 2 | 1 |
| 2 | 2 |
+---+---+
```

Использовать значения GUID

Глобально уникальные идентификаторы можно сгенерировать функцией `UUID()`. Но имейте в виду: эта функция реплицируется некорректно, хотя все будет работать нормально, если приложение прочитает результат функции к себе в память, а затем подставит в команду в виде литерала. Значения типа GUID – длинные и следуют не по порядку, поэтому плохо подходят на роль первичных ключей в таблицах InnoDB. См. раздел «Вставка строк в порядке первичного ключа в InnoDB» на стр. 158.

Разработчики MySQL создали функцию `UUID_SHORT()`, которая возвращает более короткие и последовательные значения, гораздо лучше приемлемые в качестве первичных ключей. Возможно, ее код войдет в будущие версии MySQL, но пока он официально не выпущен.

Если вы пользуетесь для создания значений каким-то глобальным генератором, следите за тем, чтобы эта единственная точка конкурентного доступа не стала узким местом приложения.

Хотя решение на основе *memcached* может работать очень быстро (десятки тысяч значений в секунду), сервер не сохраняет свое состояние. После каждой перезагрузки сервера *memcached* придется инициализировать начальное значение. Для этого предстоит найти максимальное использованное значение во всех секциях, что может потребовать очень много времени, к тому же затруднительно реализовать эту операцию атомарно.

Если вы предпочтете использовать ресурсы MySQL, то мы рекомендуем создать таблицу типа MyISAM с одной строкой и одним автоинкрементным столбцом, к которой для повышения скорости можно обращаться вне транзакций. Можно позволить этой таблице расти по мере добавления новых строк или оставить в ней всего одну строку, воспользовавшись командой REPLACE:

```
CREATE TABLE single_row (
   col1 int NOT NULL AUTO_INCREMENT,
   col2 int NOT NULL,
   PRIMARY KEY(col1),
   UNIQUE KEY(col2)
) ENGINE=MyISAM;
```

Для генерации значений эту таблицу можно использовать следующим образом:

```
mysql> REPLACE INTO single_row(col2) VALUES(1);
```

После того как данная команда завершится, для получения вновь сгенерированного значения следует обратиться к API-функции mysql_insert_id(). На разных языках это делается по-разному, в частности, ниже приведен код на Perl:

```
my $sth = $dbh->prepare('REPLACE INTO single_row(col2) VALUES(1)');
while ( my $item = @work_to_do ) {
   $sth->execute( );
   my $id = $dbh->{mysql_insert_id};
   # Сделать то, что нужно...
}
```

Для получения значения необязательно выполнять еще один запрос, например SELECT LAST_INSERT_ID(). Это потребовало бы дополнительного обращения к серверу, которое лишь снизило бы эффективность.

Инструменты для выполнения секционирования

При проектировании секционированного приложения одна из главных стоящих перед вами задач – написание кода для опроса нескольких источников данных.

Раскрытие приложению нескольких источников данных без какого бы то ни было абстрагирования считается порочной практикой, поскольку сильно увеличивает сложность кода. Лучше скрыть источники за каким-нибудь уровнем абстракции. Этот уровень мог бы решать следующие задачи:

- Соединение с нужной секцией и отправка ей запроса
- Распределенная проверка согласованности
- Объединение результатов, полученных от нескольких секций
- Межсекционные запросы

- Управление блокировками и транзакциями
- Создание секций (или, по крайней мере, автоматическое обнаружение новых секций) и их перебалансировка (если есть время для реализации такой функциональности)

Совсем не обязательно создавать всю инфраструктуру секционирования с нуля. Существует несколько инструментов и систем, которые либо уже частично предоставляют необходимую функциональность, либо специально презназначены для реализации секционированной архитектуры. Самое простое решение – воспользоваться инструментом типа MySQL Proxy, чтобы абстрагировать сложность, связанную с наличием нескольких источников данных. В зависимости от того, как MySQL Proxy будет развиваться в будущем, он может стать одной из ключевых частей многих секционированных хранилищ данных.

Из существующих уровней абстракции баз данных с поддержкой секционирования назовем Hibernate Shards *(http://shards.hibernate.org)* – расширение библиотеки объектно-реляционного отображения (ORM) с открытым исходным кодом Hibernate, написанной на Java. Компания Google разработала Hibernate Shards в числе своих знаменитых двадцатипроцентных проектов, а затем предоставила код всему сообществу. В эту библиотеку входят реализации интерфейсов Hibernate Core, поэтому для использования секционированного хранилища приложение необязательно переписывать; на самом деле ему даже не нужно знать, что оно работает с таким хранилищем. Hibernate Shards дает прекрасный способ прозрачно сохранять и извлекать данные, распределенные между несколькими серверами, но в ней отсутствуют такие средства, как перебалансирование секций и объединение результатов запросов. Для распределения данных по секциям применяется стратегия фиксированного распределения.

Еще одна система секционирования называется HiveDB *(http://www.hivedb.org)*; это среда с открытым исходным кодом, авторы которой сделали попытку лаконично и ясно реализовать базовые идеи секционирования. HiveDB написана на Java и с самого начала спроектирована для создания, использования и управления секционированным хранилищем данных. В ней есть ряд функций, отсутствующих в других системах, в частности создание секций и перемещение данных из одной секции в другую (перебалансирование). В HiveDB применяется динамическое распределение, а вместо термина «sharding» употребляется «horizontal partitioning».

Система Sphinx предназначена для полнотекстового поиска, а не для хранения и извлечения секционированных данных, но для некоторых видов запросов к секционированному хранилищу она, тем не менее, полезна. Она умеет параллельно опрашивать удаленные системы и объединять результаты, а это одна из наиболее сложных функций при работе с секционированным хранилищем (дополнительную информацию о Sphinx см. в приложении С).

Масштабирование наоборот

Один из самых простых способов справиться с увеличившимся объемом данных и ростом рабочей нагрузки состоит в том, чтобы архивировать и удалять информацию, ставшую ненужной. В зависимости от рабочей нагрузки это может дать весьма ощутимый эффект. Конечно, такой подход не устраняет необходимость в горизонтальном масштабировании, но в качестве краткосрочной стратегии, позволяющей выиграть время, годится. Да и в более долгосрочной перспективе он может применяться для борьбы с ростом объема данных.

При проектировании стратегии архивирования и удаления данных следует принимать во внимание следующие факторы.

Влияние на приложение

Хорошая стратегия архивирования позволяет убрать данные с сильно нагруженного OLTP-сервера, не оказывая заметного влияния на обработку транзакций. Основная идея состоит в том, что сначала нужно найти подлежащие удалению строки, а затем удалять их небольшими порциями. Обычно необходимо отыскать такое соотношение между количеством строк в одной порции и размером транзакции, которое обеспечивает приемлемый баланс между уровнем конкурентности и транзакционными издержками. Задания архивирования должны быть написаны так, чтобы при необходимости они уступали сервер транзакционным задачам.

Какие строки архивировать

Если вы точно знаете, что некоторые данные более не понадобятся, то можете удалить или архивировать их. Однако можно спроектировать приложение и так, чтобы оно архивировало данные, к которым редко обращаются. Архивированные данные можно хранить в отдельных таблицах и обращаться к ним через представления или даже полностью переместить на другой сервер.

Поиск в глубину или в ширину?

Наличие связей между данными может усложнить архивирование и удаление. Хорошо спроектированная программа архивирования сохраняет логическую согласованность информации, по крайней мере, в той мере, в которой это нужно приложению, не открывая гигантские транзакции с несколькими таблицами.

При наличии связей решение о том, какие таблицы архивировать в первую очередь, всегда вызывает трудности. Необходимо учитывать возможность появления в ходе архивирования «осиротевших» строк. Обычно вы должны либо пойти на нарушение ограничений внешних ключей (в InnoDB отключить такие ограничения можно командой `SET FOREIGN_KEY_CHECKS=0`), либо оставить на время записи с «висячими указателями». Что именно предпочесть, зависит от того, как приложение обращается к данным. Если программа просматривает набор взаимосвязанных таблиц сверху вниз, то и архивировать

их следует в таком же порядке. Например, если заказы всегда читаются до счетов-фактур, то сначала архивируйте заказы: приложение не должно видеть осиротевших счетов-фактур, а архивировать их можно будет позже.

Как избежать потери данных

Если архивируются данные, расположенные на нескольких серверах, то не стоит применять распределенные транзакции, тем более что архивирование может производиться в таблицы MyISAM или иного нетранзакционного типа. Поэтому, чтобы избежать потери данных, следует сначала вставить значения в конечную таблицу, а только потом удалять их из исходной. Кроме того, разумно попутно записывать архивированные данные в файл. Программа архивирования должна быть спроектирована так, чтобы ее можно было в любой момент прервать, а затем перезапустить, и это не приводило бы к рассогласованности данных или повреждению индексов.

Извлечение из архива

Зачастую можно убрать значительно больше данных, если процедура архивирования дополняется стратегией извлечения информации из архива (разархивирование). Такой подход позволяет архивировать данные, не будучи уверенными в том, что они больше не понадобятся, поскольку всегда есть возможность в случае необходимости восстановить их. Если удается выявить несколько точек, в которых система может проверить, нужно ли извлечь архивированные данные, то такую стратегию реализовать довольно просто. Например, если архивируются неактивные пользователи, то такой точкой может быть процедура входа в систему. Если вход невозможен из-за отсутствия пользователя, то программа посмотрит, нет ли такого пользователя в архиве, и если есть, то извлечет его оттуда и продолжит выполнение процедуры входа.

В комплекте Maatkit имеется инструмент, помогающий эффективно архивировать и очищать таблицы MySQL. Однако разархивирования он не поддерживает.

Отделение активных данных

Даже если вы не перемещаете устаревшую информацию на другой сервер, производительность многих приложений может увеличиться, если отделить активные данные от неактивных. Это повышает эффективность использования кэша и позволяет применять для активных и неактивных данных различные аппаратные и программные архитектуры. Ниже перечислено несколько способов решения этой задачи.

Разбиение таблицы на несколько частей

Часто имеет смысл разбить таблицу на части, особенно если она не помещается в память целиком. Например, таблицу `users` можно раз-

делить на `active_users` и `inactive_users`. Возможно, вам кажется, что это необязательно, поскольку в кэше базы данных в любом случае задерживаются только часто используемые («горячие») данные, однако это зависит от подсистемы хранения. В InnoDB единицей кэширования является страница. Если на одной странице умещается 100 пользователей и только 10% всех пользователей активны, то с точки зрения InnoDB каждая страница будет «горячей», и тем не менее 90% данных на странице – пустая трата памяти. Разбиение таблицы могло бы существенно улучшить использование ОЗУ.

В подсистеме хранения Falcon кэширование организовано построчно, поэтому кэш вроде бы должен использоваться более эффективно. Однако это не означает, что таблицы типа Falcon ничего не выигрывают от разбиения на активную и неактивную часть. Индексы в Falcon кэшируются постранично, поэтому перемешивание активных и неактивных данных снижает эффективность кэша индексов.

Секционирование на уровне MySQL

В версии MySQL 5.1 появился встроенный механизм секционированния таблиц, который может помочь удерживать недавно использовавшиеся данные в памяти. Подробнее о секционировании см. раздел «Объединенные таблицы и секционирование» на стр. 318.

Секционирование данных по времени

Если приложение постоянно получает новые данные, то вполне вероятно, что недавние данные будут гораздо активнее более старых. Например, нам известна одна служба блогов, в которой основной трафик порождают сообщения и комментарии, созданные за последние семь дней. Большая часть обновлений также относится к этим данным. В результате разработчики хранят информацию за последнюю неделю целиком в памяти и применяют репликацию для создания копии на другом диске на случай сбоя. Прочая информация постоянно хранится в другом месте.

Нам также встречались системы, в которых данные, относящиеся к каждому пользователю, содержатся в секциях на двух узлах. Новые значения поступают на «активный» узел, в котором очень много памяти и быстрые диски. Эти данные оптимизированы для очень быстрого доступа. Второй узел, где хранятся более старые данные, оснащен очень большими, но сравнительно медленными дисками. Приложение исходит из предположения, что к старым записям обращения маловероятны. Для большинства приложений это вполне здравое предположение, позволяющее удовлетворить 90% и даже более запросов на основе лишь 10% данных, поступивших в последнее время.

Такую стратегию несложно реализовать посредством динамического секционирования. Например, определение таблицы с каталогом секций могло бы выглядеть следующим образом:

```
CREATE TABLE users (
  user_id int unsigned not null,
  shard_new int unsigned not null,
  shard_archive int unsigned not null,
  archive_timestamp timestamp,
  PRIMARY KEY (user_id)
);
```

Сценарий архивирования может перемещать старые данные с активного узла на архивный, обновляя по ходу дела столбец `archive_timestamp`. В столбцах `shard_new` и `shard_archive` хранится информация о том, в каких секциях находятся соответственно новые и архивированные данные.

Масштабирование посредством кластеризации

Кластеризация – это еще один способ масштабирования за счет распределения нагрузки по многим серверам. В информатике термин «кластеризация» сильно перегружен, но в общем случае кластерная система состоит из нескольких компьютеров, находящихся в одной локальной сети и сконфигурированных как единый сервер. Вариантом кластеризации является федерация, то есть доступ к удаленным серверам, организованный так, будто они являются локальными. В результате образуется один гигантский «виртуальный сервер», играющий роль фасада, за которым скрывается множество серверов.

Кластеризация

Входящая в состав MySQL подсистема хранения NDB Cluster представляет собой распределенную, расположенную целиком в памяти, организованную по принципу «ничего не разделяется» подсистему хранения с синхронной репликацией и автоматическим секционированием по нескольким узлам. Ее характеристики производительности принципиально отличаются от любой другой подсистемы MySQL, и для ее оптимального функционирования необходимо специализированное оборудование. Хотя для некоторых прикладных задач эта система позволяет добиться исключительной производительности, но для большинства веб-приложений она пока не годится.

Подсистема NDB Cluster хороша, когда данных сравнительно мало, а запросы просты. В частности, ее можно использовать для хранения сеансов работы с веб-сайтом, метаданных файлов и т. д. Для сложных запросов, включающих соединения, она работает очень медленно. Проблема в том, что любой запрос, не сводящийся к поиску в одной таблице по индексу, требует передачи данных между узлами, поэтому выполняется долго.

NDB Cluster – транзакционная система, но она не поддерживает технологию MVCC, и операция чтения является блокирующей. Кроме того, не реализовано обнаружение взаимоблокировок. NDB разрешает взаимоблокировку посредством тайм-аута. Сочетание блокирующих чтений

и разрешения взаимоблокировок с помощью таймаутов означает, что эта подсистема не подходит для интерактивных многопользовательских систем и веб-приложений.

Поверх этой подсистемы, перед ней или ниже MySQL можно реализовать целый ряд других кластерных решений. В качестве примера упомянем систему Continuent *(http://www.continuent.com)*[1], которая поддерживает синхронную репликацию, балансирование нагрузки и переключение при отказе MySQL за счет ПО промежуточного уровня.

Федерация

У термина «федерация» тоже множество значений. В мире баз данных под этим понятием обычно имеется в виду доступ с одного сервера к данным, хранящимся на другом. Примером могут служить распределенные представления в СУБД Microsoft SQL Server.

MySQL предоставляет ограниченную поддержку федераций за счет подсистемы хранения Federated. Как и NDB Cluster, она лучше всего работает для очень простых поисковых запросов, хотя может применяться и для вставки строк на другой сервер. Текущая архитектура такова, что запросы `DELETE` и `UPDATE` менее эффективны, причем в худшем случае – существенно.

Подсистема Federated очень плохо работает для запросов `SELECT`, которые выполняют соединения и обрабатывают много данных. Например, запрос с фразой `GROUP BY` извлекает из таблицы все данные и с помощью функции `mysql_store_result`[2] выбирает данные с удаленного сервера в память локального сервера. По мере роста приложения это может стать причиной многих неприятностей. К тому же таблицы типа Federated усложняют репликацию, так как одно обновление может выполняться на нескольких серверах.

Балансирование нагрузки

Основная идея балансирования нагрузки проста: распределить ее максимально равномерно по нескольким серверам. Обычно это решается путем внедрения балансировщика нагрузки (часто он представляет собой специализированное оборудование) перед серверами. Балансировщик передает запрос на установление соединения наименее загруженному из имеющихся серверов. На рис. 9.5 изображена типичная схема балансирования нагрузки для большого веб-сайта, когда один балансировщик обслуживает HTTP-трафик, а другой – MySQL-трафик.

---

[1] Или предлагаемую разработчиками версию с открытым исходным кодом Sequoia (*http://sequoia.continuent.org*).

[2] Подробнее о функции `mysql_store_result` см. раздел «Клиент-серверный протокол MySQL» на стр. 211.

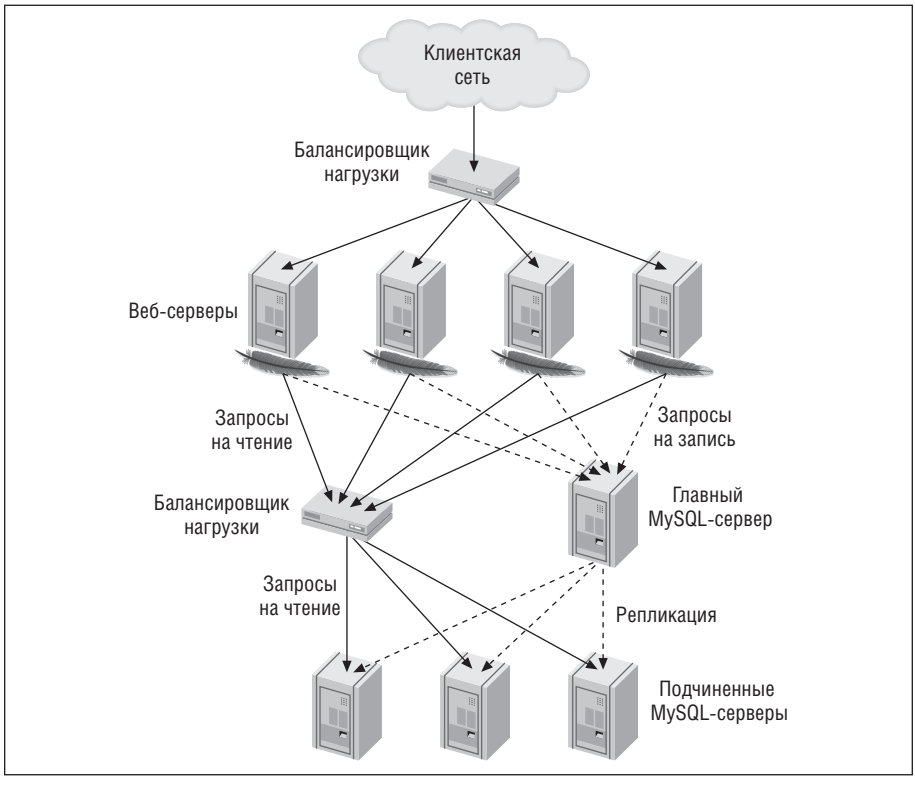


Рис. 9.5. *Типичная архитектура балансирования нагрузки для веб-сайта с большим количеством запросов на чтение*

Перед балансированием нагрузки обычно ставятся следующие цели.

Масштабируемость

Пропускную способность правильно спроектированной системы можно увеличить путем добавления новых серверов в узел. Но при этом нужно балансировать нагрузку между ними.

Эффективность

Балансирование нагрузки позволяет более эффективно использовать ресурсы, поскольку вы можете управлять тем, как маршрутизируются запросы. Это особенно важно, когда используются сервера разной мощности: более мощным можно направлять больше работы.

Доступность

Интеллектуальный балансировщик нагрузки передает запросы только тем серверам, которые в данный момент доступны.

Прозрачность

Клиентам не нужно знать о том, как сконфигурировано балансирование нагрузки. Им все равно, сколько машин находится за балансировщиком и как они называются; балансировщик представляет внешним клиентам единый виртуальный сервер.

Согласованность

Если приложение наделено состоянием (транзакции базы данных, сеансы веб-сайта и т. д.), то сам балансировщик должен направлять взаимосвязанные запросы одному и тому же серверу, чтобы состояние не терялось. Это освобождает приложение от обязанности следить за тем, с каким сервером оно соединилось.

В мире MySQL балансирование нагрузки часто применяется в сочетании с секционированием и репликацией. Можно как угодно комбинировать эти технологии и размещать их на том уровне, где это наиболее удобно приложению. Например, можно балансировать нагрузку между несколькими узлами в кластере MySQL. Или балансировать нагрузку между центрами обработки данных, а внутри каждого центра применять секционирование между узлами, каждый из которых на самом деле является реплицируемой парой главный-главный с несколькими подчиненными серверами, нагрузка между которыми также балансируется. То же относится и к стратегии обеспечения высокой доступности; переключение при отказе может быть организовано на нескольких уровнях архитектуры.

У технологии балансирования нагрузки есть много нюансов. Например, как управлять политиками чтения/записи? Некоторые балансировщики делают это самостоятельно, тогда как другие требуют, чтобы приложение знало, какие узлы доступны только для чтения, а какие еще и для записи.

Вы должны учитывать это, принимая решение о том, как реализовать балансирование нагрузки. Решений существует множество, начиная от одноранговых реализаций типа Wackamole (*http://www.backhand.org/wackamole/*) до подходов на основе системы доменных имен (DNS), виртуального сервера Linux (LVS – Linux Virtual Server; *http://www.linuxvirtualserver.org*), аппаратных балансировщиков, MySQL Proxy и управления балансированием нагрузки из приложения.

Прямое подключение

Некоторые подсознательно ассоциируют балансирование нагрузки с центральной системой, которая ставится между приложением и серверами MySQL. Но это не единственный вариант. Можно одновременно балансировать нагрузку и сохранить возможность прямого подключения к серверу MySQL из приложения. На самом деле, централизованные системы балансирования обычно хорошо работают только тог-

да, когда имеется пул серверов, которые с точки зрения приложения эквивалентны. Если приложение должно принимать решение, например, безопасно ли читать данные с некоторого подчиненного сервера, то обычно требуется уметь напрямую подключаться к этому серверу.

Принятие решений о балансировании нагрузки в приложении позволяет не только учесть особые случаи, но и добиться очень высокой эффективности. Например, если имеются два идентичных подчиненных сервера, то можно на одном из них выполнять все запросы к некоторому подмножеству секций, а на другом – запросы ко всем остальным секциям. Это позволит эффективно использовать память подчиненных серверов, так как каждому придется кэшировать лишь часть информации на диске. В случае отказа одного из них второй по-прежнему располагает всеми данными, необходимыми для обслуживания запросов ко всем секциям.

В следующих разделах обсуждаются некоторые общепринятые способы прямого подключения из приложения и те факторы, которые нужно принимать во внимание при выборе того или иного варианта.

Распределение запросов на чтение и запись при репликации

Механизм репликации в MySQL позволяет поддерживать несколько копий данных и решать, где выполнять запросы: на главном или подчиненном сервере. Основная сложность состоит в том, что делать с неактуальностью данных на подчиненном сервере, неизбежной в силу асинхронной природы репликации. Кроме того, подчиненные серверы должны использоваться только для чтения, тогда как главный может выполнять запросы любого вида.

Обычно приложение приходится модифицировать с учетом этих соображений[1]. Приложение может посылать все запросы на запись главному серверу, а запросы на чтение распределять между главным и подчиненными серверами. В случае, когда некоторое устаревание данных несущественно, можно читать с подчиненных серверов, а если необходимы гарантированно актуальные данные, то с главного.

Те же рассуждения применимы к паре главный-главный с активным и пассивным главным сервером, хотя в данном случае запросы на запись должны отправляться только активному серверу. Чтение можно выполнять и на пассивном сервере, если потенциальное устаревание не страшно.

Самая серьезная проблема – как избежать аномалий при чтении неактуальных данных. Классическая ситуация такого рода возникает, когда пользователь вносит некое изменение, скажем, добавляет комментарий в блог, затем перезагружает страницу и не видит своего изменения, поскольку приложение прочитало неактуальные данные с подчиненного сервера.

---

[1] Если для распределения запросов применяется MySQL Proxy, то, возможно, изменять приложение и не придется.

Ниже перечислены некоторые часто применяемые методы распределения запросов на чтение и запись.

Распределение с учетом запроса

Самый простой способ – направлять все запросы на запись, а также те запросы на чтение, для которых неактуальные данные неприемлемы, активному или главному серверу. Все остальные запросы на чтение попадают на подчиненный или пассивный сервер. Эту стратегию легко реализовать, но на практике подчиненный сервер будет задействован не в полной мере, поскольку не так уж много существует запросов, которым всегда безразлична актуальность данных.

Распределение по степени неактуальности данных

Это некоторое усовершенствование стратегии распределения с учетом запроса. От приложения требуется сравнительно немного дополнительной работы, чтобы понять, насколько отстает подчиненный сервер, и решить, являются ли данные слишком устаревшими. Так поступают многие приложения, формирующие отчеты: коль скоро все данные за прошлый день реплицированы на подчиненный сервер, наличие стопроцентной синхронизации с главным сервером не имеет значения.

Распределение с учетом сеанса

Чуть более сложный алгоритм решения вопроса о том, откуда читать данные – учитывать, изменял ли их конкретный пользователь. Работающему с приложением человеку необязательно видеть все модификации, внесенные другими пользователями, но свои он наблюдать должен. Чтобы реализовать это на уровне сеанса, достаточно установить в нем флажок, который показывает, что в сеансе было произведено изменение, и в течение некоторого времени после этого события направлять запросы на чтение, исходящие от этого пользователя, главному серверу.

Эту тактику можно сочетать с мониторингом отставания репликации; если пользователь изменил какие-то данные 10 секунд назад, и ни один подчиненный сервер не отстает более чем на 5 секунд, то читать с подчиненного сервера безопасно. Очень разумно было бы выбрать какой-то один подчиненный сервер и задействовать его на протяжении всего сеанса, в противном случае пользователь может наблюдать странные эффекты, вызванные тем, что некоторые подчиненные серверы отстают больше, чем другие.

Распределение с учетом версии

Эта стратегия похожа на распределение с учетом сеанса: можно отслеживать номера версий и/или временные метки объектов и читать номер версии или временную метку с подчиненного сервера, чтобы понять, насколько свежие на нем данные. Если информация на подчиненном сервере сильно устарела, то можно обратиться за актуальными данными к главному серверу. Можно также увеличивать

номер версии объекта верхнего уровня даже в том случае, когда сам объект не изменяется; это упрощает проверку на устаревание (необходимо заглядывать только в одно место – в объект верхнего уровня). Например, можно обновлять версию пользователя, если он помещает в свой блог новую запись. В таком случае запросы на чтение будут направляться главному серверу.

Чтение версии объекта с подчиненного сервера означает дополнительные накладные расходы, которые можно уменьшить за счет кэширования. Вопрос о кэшировании и версиях объектов мы рассмотрим ниже в этой главе.

Глобальное распределение с учетом версии или сеанса

Это вариация на тему распределений с учетом версии и учетом сеанса. Производя операцию записи, приложение выполняет запрос `SHOW MASTER STATUS` после фиксации транзакции. Координаты в журнале главного сервера сохраняются в кэше и считаются модифицированным номером версии объекта и/или сеанса. Затем, когда приложение подключается к подчиненному серверу, оно выполняет команду `SHOW SLAVE STATUS` и сравнивает координаты подчиненного сервера с запомненной версией. Если подчиненный сервер дошел хотя бы до той точки, где главный зафиксировал транзакцию, то читать с него безопасно.

В большинстве решений по распределению запросов на чтение и запись необходимо следить за отставанием подчиненного сервера и, исходя из этого, решать, куда отправлять запрос. Но имейте в виду, что столбец `Seconds_behind_master` в результатах команды `SHOW SLAVE STATUS` – ненадежный способ измерения отставания подчиненного сервера. Подробнее см. раздел «Измерение отставания подчиненного сервера» на стр. 470.

Если ваша цель – масштабируемость любой ценой, и неважно, сколько оборудования для этого потребуется, то можно упростить себе жизнь и либо не пользоваться репликацией вовсе, либо применять ее только для обеспечения высокой доступности, но не для балансировки нагрузки. Это поможет избежать сложностей, связанных с распределением запросов между главным и подчиненными серверами. Некоторые считают такой подход оправданным, другие – расточительным использованием оборудования. Такое различие во мнениях отражает разные цели – чего именно требуется достичь: только масштабируемости или еще и эффективности? Если эффективность является одной из целей и, следовательно, хотелось бы задействовать подчиненные серверы не только для хранения копий данных, то, вероятно, вам придется пойти на увеличение сложности.

Изменение конфигурации приложения

Один из способов распределения нагрузки – изменение конфигурации приложения. Например, можно настроить несколько машин так, чтобы они разделяли между собой бремя генерации больших отчетов. В этом

случае разные компьютеры подключаются к разным серверам MySQL, и каждый генерирует отчет для каждого N-го заказчика или офиса.

Реализовать подобную систему обычно совсем несложно, но, если при этом потребуется вносить изменения в код, в том числе в конфигурационные файлы, то система станет слишком хрупкой и громоздкой. Любое решение, требующее изменений программного кода на каждом сервере или даже одного изменения в каком-то центральном пункте, откуда оно «публикуется» путем копирования файлов или команд обновления в системе управления версиями, принципиально порочно. С другой стороны, если конфигурация хранится в базе данных и/или в кэше, то можно обойтись без публикации изменений, внесенных в код.

Изменение доменных имен

Создание доменных имен для разных целей – это очень прямолинейная техника балансирования нагрузки, которая тем не менее неплохо работает в некоторых простых приложениях. Можно написать периодически запускаемое задание, которое ведет мониторинг серверов MySQL и по мере необходимости изменяет соответствие между доменным именем и сервером. В простейшем случае имеется одно доменное имя для серверов, допускающих только чтение, и еще одно – для сервера, на котором выполняются операции записи. Если подчиненные серверы успевают за главным, то первое доменное имя указывает на подчиненные серверы; когда они начинают отставать, это имя перенаправляется на главный сервер.

Технику на основе доменных имен очень легко реализовать, но и недостатков у нее масса. И самое главное – тот факт, что система DNS вам не полностью подконтрольна.

- Изменения в системе DNS не происходят мгновенно. Требуется довольно длительное время для того, чтобы изменение распространилось по сети или между сетями.
- Данные DNS кэшируются в разных местах, и срок хранения является рекомендательным, а не обязательным.
- Для того чтобы изменение в системе DNS было полностью воспринято, возможно, придется перезагрузить приложение или сервер.
- Не стоит использовать несколько IP-адресов для одного доменного имени и полагаться на алгоритм циклического перебора (round-robin) для балансирования запросов. Поведение этого алгоритма не всегда предсказуемо.
- Изменения в системе DNS не атомарны.
- Администратор базы данных не всегда имеет прямой доступ к системе DNS.

Достаточно опасно полагаться на систему, которая не полностью контролируема. Это можно делать только для очень простых приложений. Степень контролируемости немного повысится, если вносить изменения в файл */etc/hosts*, а не в систему DNS. Можно с уверенностью сказать, что изменение вступило в силу сразу после модификации этого файла. Это лучше, чем дожидаться, пока истечет срок хранения DNS-записи в кэше, но все равно не идеально.

Обычно мы не рекомендуем создавать приложения, хоть как-то зависящие от DNS. Лучше избегать такой зависимости даже в простом ПО, поскольку никто не знает, как оно разрастется в будущем.

Переключение IP-адресов

Некоторые способы балансирования нагрузки полагаются на переключение виртуальных IP-адресов[1] между серверами; иногда такая техника прекрасно работает. На первый взгляд выглядит похоже на изменение доменных имен, но на самом деле общего между этими технологиями мало. Серверы прослушивают сетевой трафик не по доменному имени, а по конкретному IP-адресу, поэтому переключение IP-адресов сохраняет статичные доменные имена. Известить об изменении IP-адреса можно очень быстро и атомарно с помощью команд протокола ARP (Address Resolution Protocol).

Такая техника применяется в системах Wackamole и LVS. Например, они позволяют ассоциировать один IP-адрес с ролью «только чтение», а затем по мере необходимости передают его от одной машины другой. Система Wackamole способна управлять множеством IP-адресов и гарантирует, что в любой момент времени прослушивать каждый адрес из числа находящихся в пуле будет одна и только одна машина. Wackamole обладает уникальной особенностью – это пиринговая система, поэтому в ней нет единой точки отказа.

Полезный прием состоит в том, чтобы назначить фиксированный IP-адрес каждому физическому серверу. Этот адрес – принадлежность самого сервера, и он никогда не изменяется. Далее для каждой логической «службы» выделяется виртуальный IP-адрес. Эти адреса могут передаваться от одного сервера другому, следовательно, возникает возможность перемещать экземпляры служб и приложений без переконфигурирования приложения. Это очень удобно, даже если для балансирования нагрузки или обеспечения высокой доступности не приходится часто переключать IP-адреса.

Посредники

До сих пор мы предполагали, что приложение взаимодействует напрямую с серверами MySQL. Однако во многих схемах балансирования

[1] Виртуальный IP-адрес не связан ни с каким конкретным компьютером или сетевым интерфейсом; он «мигрирует» от одного компьютера к другому.

нагрузки встречается посредник, задача которого – выступать в роли прокси-сервера сетевого трафика. На одном конце посредник принимает весь трафик, а на другом направляет его нужному серверу, при этом ответы сервера доставляются компьютеру, с которого поступил исходный запрос. Иногда посредник реализуется аппаратно, а иногда программно[1]. Такая архитектура изображена на рис. 9.6. Подобные решения обычно работают очень хорошо, но если сам балансировщик нагрузки не резервирован, то он становится единой точкой отказа.

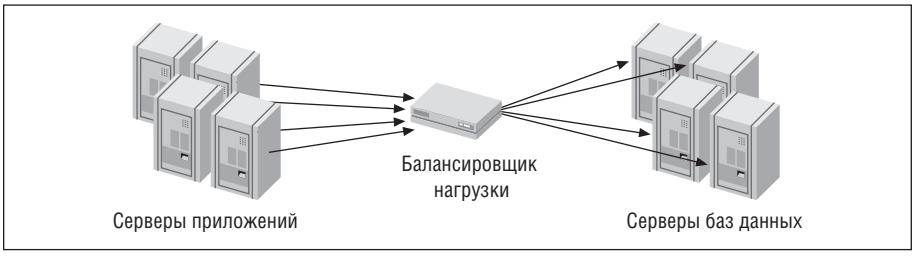


***Рис. 9.6.** Балансировщик нагрузки, работающий как посредник*

Балансировщики нагрузки

На рынке представлено немало аппаратных и программных решений для балансирования нагрузки, но лишь немногие спроектированы специально для серверов MySQL[2]. Гораздо чаще в балансировании нагрузки нуждаются веб-сервера, поэтому во многие универсальные устройства встроена специальная поддержка протокола HTTP и лишь самые базовые средства для всего остального.

Исключение составляет программа MySQL Proxy, которая помогает разделять операции чтения и записи для некоторых приложений. Она увеличивает сложность и добавляет накладные расходы, но также повышает гибкость и позволяет программировать сценарии для описания нестандартных ситуаций распределения записи и чтения. Хотя это сравнительно новая система, уже существует немало пособий и примеров ее использования для специализированного оперативного балансирования нагрузки. Поскольку она способна «заглядывать» внутрь ретранслируемого клиент-серверного диалога, то потенциально обладает возможностью очень изощренной маршрутизации запросов.

---

[1] Систему LVS **можно сконфигурировать так, что она вступает в игру только тогда, когда приложению необходимо создать новое соединение, а после этого не выполняет функции посредника.**

[2] Выше в этой главе мы упоминали некоторые программные реализации (Sequoia, Continuent). Существует также модуль `DBIx::DBCluster` для Perl, и не зависящая от языка система SQL Relay (*http://sqlrelay.sourceforge.net*).

Поскольку соединения с сервером MySQL – это обычные TCP/IP-соединения, то для MySQL можно использовать балансировщики нагрузки общего назначения. Однако отсутствие специальной поддержки MySQL налагает следующие ограничения:

- Если балансировщик нагрузки не знает об истинной нагрузке на MySQL, то он способен лишь *распределять запросы*, а не *балансировать нагрузку*. Не все запросы одинаковы, но универсальный балансировщик обычно не видит различий между ними.
- Большинство балансировщиков знают о структуре HTTP-запроса и умеют ассоциировать сеанс с сервером, сохраняя тем самым состояние веб-сервера. У соединений MySQL тоже имеется состояние, но балансировщик понятия не имеет, как связать все запросы на соединение, исходящие из одного сеанса, с одним сервером MySQL. Результатом оказывается снижение эффективности (если бы все запросы от одного сеанса попадали одному серверу MySQL, то кэш этого сервера использовался бы более эффективно).
- Наличие пула соединений и постоянных соединений может помешать балансировщику нагрузки распределять запросы на соединение. Пусть, например, пул открывает заданное количество соединений, а балансировщик нагрузки распределяет их между четырьмя серверами MySQL. А теперь предположим, что добавлено еще два сервера. Поскольку пул не запрашивает создание новых соединений, то эти серверы простаивают. Кроме того, соединения из пула могут распределяться между серверами несправедливо, так что некоторые окажутся перегруженными, а другие – недогруженными. Эти проблемы можно до некоторой степени решить, задавая на различных уровнях срок хранения соединений в пуле, но это сложно и труднореализуемо. Пул соединений работает оптимально, когда самостоятельно занимается балансированием нагрузки.
- Большинство универсальных балансировщиков нагрузки умеет контролировать состояние соединения только для HTTP-серверов. Простой балансировщик может проверить, что сервер принимает соединение с некоторым TCP-портом, это абсолютно необходимый минимум. Более развитый балансировщик может отправить HTTP-запрос и проанализировать код ответа, чтобы удостовериться в нормальной работе веб-сервера. Но MySQL не принимает HTTP-запросы на порт 3306, так что контролировать работоспособность придется самостоятельно. Можно установить на компьютер, где работает сервер MySQL, еще и HTTP-сервер и направить балансировщик нагрузки на специальный сценарий, который уже проверит состояния MySQL и вернет соответствующий код[1]. Обращать внимание нужно,

[1] Если вы способны самостоятельно написать программу, которая будет прослушивать порт 80, и можете сконфигурировать демон *xinetd* так, чтобы он вызывал вашу программу, то и веб-сервер устанавливать не придется.

прежде всего, на загрузку операционной системы (обычно эти данные доступны в файле */proc/loadavg*), состояние репликации и количество соединений с сервером MySQL.

Алгоритмы балансирования нагрузки

Существует много разных алгоритмов для определения того, какой сервер должен получить следующий запрос на соединение. Каждый поставщик применяет собственную терминологию в этой области, но из приведенного ниже перечня должно быть понятно, какие имеются варианты.

Случайное

Балансировщик передает соединение серверу, случайно выбранному из пула доступных серверов.

Циклическое

Балансировщик передает соединения серверам по кругу: A, B, C, A, B, C и т. д.

Минимум соединений

Следующее соединение передается серверу, с которым в данный момент установлено наименьшее количество соединений.

Самый быстрый ответ

Следующее соединение передается серверу, который быстрее всего обрабатывает запросы. Этот алгоритм неплохо работает, когда в пул включены как быстрые, так и медленные компьютеры. Однако применение его к SQL проблематично, поскольку сложность запросов варьируется в очень широких пределах. Даже один и тот же запрос может вести себя очень по-разному в зависимости от обстоятельств, например, от того, обслуживается он с диска или из кэша запросов.

Хешированное

Балансировщик хеширует IP-адрес источника соединения, отображая его на один из серверов в пуле. Таким образом, все соединения с одного источника поступают одному и тому же серверу. Отображение модифицируется лишь при изменении количества компьютеров в пуле.

Взвешенное

Балансировщик может применять сразу несколько алгоритмов, приписывая каждому определенный вес. Например, в наличии могут быть компьютеры с одним и двумя процессорами. Машины с двумя процессорами примерно в два раза мощнее, поэтому балансировщик нагрузки можно сконфигурировать так, чтобы он посылал им в два раза больше соединений (в среднем).

Какой алгоритм окажется для вас наилучшим, зависит от рабочей нагрузки. Например, балансирование по минимуму соединений может привести к перегрузке запросами вновь добавленных в пул серверов –

как раз в тот момент, когда их кэш еще не прогрет. Авторы первого издания настоящей книги сталкивались с подобной проблемой на практике.

Для подбора оптимального для имеющей рабочей нагрузки решения необходимо экспериментировать. Обязательно протестируйте его не только в нормальных, но и в экстраординарных условиях. Только при крайне необычных обстоятельствах – высокая нагрузка, изменение схемы или ненормально большое количество серверов, выведенных из эксплуатации, – можно допустить аномальное поведение.

Мы описали только алгоритмы немедленного обслуживания, когда запросы на соединение не ставятся в очередь. Иногда алгоритмы с очередями оказываются более эффективны. Например, можно поддерживать заданный уровень конкуренции на сервере баз данных, скажем, не более N активных транзакций в каждый момент времени. Если транзакций слишком много, то очередной запрос ставится в очередь и обслуживается, когда какой-нибудь сервер станет «доступен» в соответствии с выбранным критерием. Некоторые пулы соединений поддерживают подобные алгоритмы с очередями.

Добавление серверов в пул и удаление из пула

Чтобы добавить новый сервер в пул, обычно недостаточно просто включить его в сеть и уведомить балансировщик о его существовании. Если вам кажется, что все будет хорошо, если сервер не перегружен запросами на соединение, то и это не всегда верно. Иногда можно увеличивать нагрузку на сервер постепенно, но некоторые серверы с «холодным» кэшем могут отвечать настолько медленно, что в течение некоторого времени не должны получать вообще *никаких* запросов.

Если кэш сервера не прогрет, то даже самый простой запрос может занять длительное время. Если на то, чтобы показать страницу пользователю, уходит 30 секунд, то такой сервер нельзя счесть пригодным для использования даже при небольшом объеме трафика. Чтобы решить эту проблему, можно в течение некоторого времени повторять на новом сервере все запросы `SELECT` с какого-нибудь активного сервера и только потом известить о нем балансировщик нагрузки. Для этого достаточно читать и воспроизводить на новом сервере файлы журналов активного сервера.

Конфигурировать серверы, входящие в пул соединений, следует так, чтобы оставалась достаточная резервная пропускная способность на время вывода некоторых серверов для обслуживания и в случае, если какой-то из них откажет. Пропускная способность каждого узла должна быть заложена с запасом.

Убедитесь, что конфигурационные лимиты достаточно высоки для продолжения работы после удаления некоторых серверов из пула. Например, если каждый сервер MySQL обычно обрабатывает 100 соединений, то параметр `max_connections` должен быть установлен в 200. Тогда, даже

если половина узлов выйдет из строя, тех, что остались в пуле, хватит для обработки такого же количества соединений, что и раньше.

Балансирование нагрузки при наличии одного главного и нескольких подчиненных серверов

Самая распространенная топология репликации – это один главный и несколько подчиненных серверов. Отойти от такой архитектуры не всегда легко. Во многих приложениях предполагается, что существует единственное место для всех операций записи или что имеется один сервер, на котором в любой момент присутствуют все данные. Хотя с точки зрения масштабирования такая архитектура не является наилучшей, ее можно с успехом применить, если воспользоваться балансированием нагрузки. В этом разделе рассматриваются некоторые подходы в указанном направлении.

Функциональное секционирование

Пропускную способность можно немного увеличить, если сконфигурировать некоторые подчиненные серверы или группы таких серверов под конкретные цели. Например, выделить серверы для генерации отчетов и анализа данных, организации хранилищ и полнотекстового поиска. Дополнительные идеи вы найдете в разделе «Специальные схемы репликации» на стр. 460.

Фильтрация и секционирование данных

Данные можно распределить по подчиненным серверам с помощью фильтров репликации (см. выше раздел «Фильтры репликации» на стр. 447). Эта стратегия хорошо работает, если информация на главном сервере уже находится в разных базах или таблицах. К сожалению, не существует встроенного способа производить фильтрацию на уровне строк. Однако такую схему можно реализовать путем репликации на сервер-распределитель и использования таблиц типа Blackhole с триггерами для вставки строк в разные таблицы в зависимости от значения некоторого столбца.

Можно делать даже более экзотические вещи, например, реплицировать в таблицы типа Federated, но не исключено, что это приведет к неразберихе. Таблицы типа Federated привносят зависимости между серверами, чего обычно лучше избегать.

Даже если вы не секционируете данные между подчиненными серверами, эффективность кэша можно повысить за счет секционирования операций чтения вместо того, чтобы распределять их случайно. Например, можно направлять запросы на чтение для пользователей, чьи фамилии начинаются с букв A–M, на один подчиненный сервер, а запросы для всех остальных пользователей – на другой. Тогда кэш каждого сервера будет использоваться эффективнее, так как более вероятно, что данные для повторяющихся запросов уже находятся в кэше. В лучшем случае, когда операций записи нет вообще, такая

стратегия дает кэш, размер которого равен сумме размеров кэшей обоих серверов. Для сравнения, если распределять запросы на чтение между серверами случайно, то данные в кэше каждого сервера дублируются, поэтому его полный эффективный размер не превышает размер самого большого кэша на всех подчиненных серверах независимо от их количества.

Перенос части операций записи на подчиненный сервер

Не всегда главный сервер обязан выполнять все операции записи. И главный, и подчиненные сервера можно освободить от значительного объема работы, если разложить запросы на части и некоторые из них выполнять непосредственно на подчиненных серверах. Дополнительную информацию по этому поводу см. в разделе «Слишком большое отставание репликации» на стр. 494.

Гарантированная актуальность данных на подчиненном сервере

Если на подчиненном сервере требуется запустить некоторый процесс, которому важно, чтобы данные были актуальны хотя бы на известный момент времени, даже если придется подождать, то можно воспользоваться функцией `MASTER_POS_WAIT()`, которая блокирует процесс до тех пор, пока подчиненный сервер не достигнет желаемой степени синхронизации с главным. Или можно применить для контроля актуальности записи сердцебиения (heartbeat) репликации, хотя они и не дают субсекундной точности. Эта техника описана в разделе «Измерение отставания подчиненного сервера» на стр. 470.

Синхронизация записи

Можно воспользоваться функцией `MASTER_POS_WAIT()`, которая позволяет гарантировать, что операция записи действительно дошла до одного или нескольких подчиненных серверов. Если приложение желает эмулировать синхронную репликацию для обеспечения гарантированной безопасности данных, то можно перебирать все подчиненные серверы в цикле, выполняя `MASTER_POS_WAIT()` для каждого. Тем самым создается «барьер синхронизации», для пересечения которого может потребоваться длительное время, если какие-то из подчиненных серверов сильно отстают; поэтому применять такую технику следует только в случае крайней необходимости. Если нужно лишь гарантировать, что какой-то сервер получил событие, то можно дожидаться синхронизации именно этого сервера.

Высокая доступность

Обычно считается, что система доступна, если она отвечает пользователям. Однако суть доступности может оказаться и более сложной. Если часть приложения отказала, но в него встроена возможность продолжать работу и в таких условиях, то оно может отвечать в режиме с ухудшенными характеристиками (degrade mode). Также приложение может быть переведено в режим только чтения для технического обслу-

живания или вследствие аварии; считать ли, что в это время оно «работает», – дело ваше. Например, большинство пользователей сайтов фотогалерей не возражает, если в течение краткого периода они не смогут загружать фотографии; с другой стороны, пользователь банкомата вряд ли обрадуется, увидев сообщение «техобслуживание – работа возможна только в режиме чтения». Стало быть, веб-сайт можно считать в этом случае работающим, а банкомат – нет.

Обеспечить высокую доступность в принципе очень просто: нужно лишь встроить в систему избыточность и гарантировать, что в случае, когда какая-то часть выходит из строя, тут же подключается резервная. Проблема в том, как сделать это быстро и надежно.

Планирование с учетом высокой доступности

У разных приложений требования к доступности очень разнятся. Прежде чем формулировать, какую долю времени система должна быть доступна, подумайте, чего вы на самом деле хотите добиться. Каждое следующее увеличение степени доступности, как правило, обходится дороже предыдущего; отношение доступности к потраченным усилиям и стоимости нелинейно.

Самый важный принцип в деле обеспечения высокой доступности – выявить и устранить единые точки отказа в системе. Внимательно исследуйте свое приложение и постарайтесь отыскать все такие точки. Это может быть жесткий диск, сервер, коммутатор или маршрутизатор, или источник питания для какой-то стойки. Все ли компьютеры находятся в одном центре обработки данных (ЦОД)? Предоставляются ли «резервные» ЦОД той же компанией, что и основные? Любой нерезервированный участок системы может стать единой точкой отказа. Среди других распространенных единых точек отказа назовем зависимость от таких служб, как DNS, от единственного поставщика доступа к сети (убедитесь, что резервные подключения действительно выходят на разные магистральные сети Интернет) и от единственного поставщика электроэнергии.

Попытайтесь учесть все компоненты, влияющие на доступность, составьте взвешенную картину рисков и начинайте работу с наиболее значимых. Некоторые разработчики прилагают массу усилий к тому, чтобы создать программу, которая сможет пережить любые аппаратные сбои, но простои из-за ошибок в таком ПО могут оказаться больше, чем если бы его вообще не было. Те, кто строит «непотопляемые» системы, в которых резервировано все, что только можно, забывают, что ЦОД может быть обесточен или лишен доступа к сети. А быть может, разработчики не принимают во внимание злоумышленные действия хакеров или потенциальные ошибки программистов, приведшие к удалению или повреждению данных; случайная команда `DROP TABLE` тоже может стать причиной простоя.

Выявить наиболее значимые опасности можно, вычислив подверженность риску, то есть вероятность отказа, умноженную на стоимость этого отказа. Простенькая таблица рисков, содержащая столбцы «вероятность», «стоимость», «подверженность риску», поможет приоритезировать усилия.

Не всегда единую точку отказа удается устранить. Резервирование компонента может оказаться невозможным из-за того или иного ограничения, которое никак не обойти, например, связанного с географическим местоположением, финансированием или временными рамками.

На следующем шаге подумайте о переключении на резервную систему в случае сбоя, модернизации оборудования или системного ПО, модификации приложения или планового техобслуживания. Для всего, что может сделать приложение недоступным, необходим план переключения в случае сбоя, и вам предстоит решить, насколько быстро должна происходить данная процедура.

С этим связан другой вопрос: как быстро следует заменить вышедший из строя компонент после выполнения переключения? Пока пропускная способность не восстановлена в полном объеме, степень резервирования уменьшена, и система подвергается дополнительному риску. Следовательно, наличие резервной системы не устраняет необходимость в замене отказавших компонентов. Сколько времени уйдет у вас на подготовку нового резервного сервера, установку на него ОС и копирование актуальных данных? Достаточно ли у вас запасных компьютеров? Возможно, понадобится больше одного.

Еще один фактор, который следует учитывать: возможна ли потеря информации, хотя приложение и не переставало работать? В случае катастрофического отказа сервера, данные могут быть частично потеряны, по крайней мере, те транзакции, которые были записаны в утраченный двоичный журнал, но не попали в журнал ретрансляции подчиненного сервера. С этим можно смириться? Для большинства приложений ответ положительный; если же это не так, то потребуется, как правило, дорогостоящее и сложное решение, сопряженное с дополнительными накладными расходами. Например, можно воспользоваться заплатами Google для синхронной репликации (подробнее об этом позже) или поместить двоичный журнал на устройство, реплицируемое с помощью механизма DRBD, так что журнал не будет потерян даже при полном отказе системы.

Продуманная архитектура приложения зачастую может снизить требования к доступности, по крайней мере, для части системы, так что обеспечить высокую доступность в целом будет проще. Разделение системы на критические и некритические части поможет сэкономить немало труда и денег, поскольку чем меньше система, тем легче обеспечить резервирование и высокую доступность.

В общем случае сделать приложение высокодоступным и предотвратить потерю данных становится трудно и дорого по достижении определен-

ного порога, поэтому мы рекомендуем ставить реалистичные цели и избегать чрезмерных «наворотов». К счастью, обеспечить две или три девятки не так уж сложно, хотя все зависит от конкретного приложения.

Избыточность

Избыточность системы может принимать две формы: запасная пропускная способность и дублирование компонентов.

Обеспечить запас пропускной способности довольно просто – можно воспользоваться любым из упомянутых в настоящей главе методов. Один из способов повышения доступности состоит в том, чтобы создать кластер или пул серверов и воспользоваться тем или иным способом балансирования нагрузки. Если какой-нибудь сервер выходит из строя, то нагрузка перераспределяется между остальными. Вообще говоря, разумно по возможности не загружать компоненты «под завязку», поскольку в этом случае у вас остается запас для поддержания требуемой производительности при возрастании нагрузки или отказе некоторых компонентов.

Во многих ситуациях может потребоваться дублировать компоненты и держать резервный наготове, чтобы выполнить аварийное переключение на резерв в случае отказа основного. Дублировать можно сетевые карты, маршрутизаторы, накопители на жестких дисках – все, что с большой вероятностью способно выйти из строя.

Дублировать сервер MySQL целиком несколько сложнее, поскольку без данных сервер бесполезен. Следовательно, нужно позаботиться о том, чтобы резервные сервера имели доступ к базам основного сервера. В следующих разделах описывается, как можно решить эту задачу.

Архитектуры с общим хранилищем

Общая внешняя память – это один из способов устранить некоторые виды единых точек отказа. Обычно в этом качестве выступают SAN-сети (см. раздел «Сети хранения данных» в главе 7, стр. 406). В этом случае активный сервер монтирует свою файловую систему и функционирует как обычно. Если активный сервер выходит из строя, то резервный может подмонтировать ту же самую файловую систему, выполнить операции восстановления и запустить MySQL для работы с файлами отказавшего сервера. Логически этот процесс ничем не отличается от починки отказавшего узла, только все происходит намного быстрее, так как резервный сервер уже загружен и готов к работе. Время уходит разве что на проверку файловой системы и восстановление на уровне InnoDB.

Общее внешнее хранилище помогает предотвратить потерю данных в некоторых случаях, но само остается единой точкой отказа. Если оно выходит из строя, то перестает работать вся система. А если сбой приводит к повреждению файлов данных, то резервный сервер может оказаться не в состоянии выполнить восстановление. Мы настоятельно

рекомендуем применять в сочетании с общей внешней памятью InnoDB иную транзакционную подсистему хранения. Любой сбой почти наверняка повредит таблицы типа MyISAM, а для их восстановления иногда требуется длительное время.

Архитектуры с реплицируемыми дисками

Реплицируемый диск – еще один способ уберечь данные в случае катастрофического отказа главного сервера. Совместно с MySQL чаще всего используется система DRBD (*http://www.drbd.org*) в сочетании с инструментами из проекта Linux-HA (подробнее об этом ниже).

DRBD – это синхронный механизм репликации блочного уровня, реализованный в виде модуля ядра Linux. Он копирует каждый блок первичного устройства по сети на вторичное блочное устройство и записывает этот блок в место назначения еще до того, как он будет зафиксирован на первичном устройстве[1].

DRBD работает только в активно-пассивном режиме. Пассивное устройство играет роль горячего резерва, к нему вообще нельзя обращаться – даже в режиме чтения, – пока оно не станет первичным. Поскольку любая операция записи должна завершиться на вторичном устройстве раньше, чтобы запись на первичное считалась состоявшейся, то вторичное устройство должно быть, как минимум, таким же быстрым, как первичное, иначе оно будет ограничивать производительность основной системы. Кроме того, если для создания резерва используется DRBD, то оборудование резервного сервера должно быть таким же, как у основного.

Если активный сервер выходит из строя, можно сделать вторичное устройство первичным. Но, поскольку DRBD реплицирует диск на блочном уровне, файловая система рискует оказаться рассогласованной. Следовательно, для быстрого восстановления лучше использовать журналируемую файловую систему. По завершении ее воссоздания MySQL, скорее всего, должен будет запустить собственные средства восстановления. Когда первый сервер снова вступит в строй, он синхронизирует свое устройство с новым первичным устройством и примет на себя роль вторичного.

С точки зрения фактической реализации переключения на резерв при отказе система DRBD аналогична SAN: имеется резервный компьютер, который начинает обслуживать те же данные, что машина, вышедшая из строя. Основное отличие заключается в том, что это реплицируемая, а не общая память, поэтому при использовании DRBD вы обслуживаете копию данных, тогда как в случае SAN речь идет об обслуживании

[1] Уровень синхронизации DRBD поддается настройке. Можно установить асинхронный режим, режим, в котором система ждет получения данных удаленным устройством, или режим блокировки до завершения записи на удаленный диск. Кроме того, настоятельно рекомендуется отводить по DRBD отдельную сетевую карту.

той же самой информации, расположенной на том же самом физическом устройстве, к которому обращалась отказавшая машина. В обоих случаях кэши сервера MySQL будут пусты в тот момент, когда он запускается на резервной машине. Напротив, кэши подчиненного сервера репликации, скорее всего, будут хотя бы частично «прогреты».

У системы DRBD есть особенности, позволяющие предотвратить проблемы, свойственные ПО кластеризации. В качестве примера назовем синдром split-brain, который возникает, когда два узла пытаются сделать себя первичными одновременно. DRBD можно сконфигурировать таким образом, что этот феномен никогда не возникнет. Однако система DRBD годится не для всех ситуаций. Рассмотрим некоторые присущие ей недостатки.

- Переключение на резервный сервер при сбое в DRBD осуществляется сравнительно медленно. Обычно для превращения вторичного устройства в первичное требуется не менее пяти секунд, и это не считая восстановления файловой системы и MySQL.
- Работа DRBD обходится дорого, поскольку она функционирует только в активно-пассивном режиме. Когда реплицируемое устройство на сервере горячей замены работает в пассивном режиме, оно больше ни для чего не пригодно. Считать ли это недостатком, зависит от принятой точки зрения. Если требуется по-настоящему высокая доступность и вы не можете смириться с понижением уровня обслуживания при сбое, то ни на какую пару машин нельзя возлагать нагрузку, превышающую нагрузку на одну машину, поскольку в этом случае при выходе любой машины из строя оставшаяся не сможет справиться с нагрузкой. Конечно, можно использовать резервный сервер еще для чего-нибудь, например, в роли подчиненного сервера репликации, но часть ресурсов все равно будет простаивать.
- Для таблиц типа MyISAM этот механизм практически бесполезен, поскольку их проверка и восстановление занимают слишком много времени. Вообще, подсистема хранения MyISAM – неудачный выбор для обеспечения высокой доступности; применяйте вместо нее InnoDB или любую другую подсистему с хорошей производительностью восстановления.
- Система DRBD не заменяет резервного копирования. Если данные на диске повреждены в результате злонамеренного вмешательства, ошибки, оплошности или аппаратного сбоя, то DRBD не поможет: реплицированные данные представляют собой точный дубликат испорченного оригинала. Для защиты от такого рода проблем необходима резервная копия (или репликация на уровне MySQL с задержкой по времени).

Мы предпочитаем использовать DRBD для репликации только тех устройств, на которых хранятся двоичные журналы. При выходе из строя активного сервера можно запустить на пассивном узле сервер журналов и использовать восстановленные двоичные журналы для

приведения всех серверов, подчиненных отказавшему главному, к самой последней точке (см. раздел «Создание сервера журналов» главы 8 на стр. 465). После этого можно выбрать какой-то из подчиненных серверов и выдвинуть его на роль главного, заменив тем самым систему, вышедшую из строя.

Синхронная репликация MySQL

При синхронной репликации транзакция не может завершиться на главном сервере, пока не будет зафиксирована на одном или нескольких подчиненных. Существуют различные уровни синхронной репликации, для которых применяются общеупотребительные названия. В настоящее время в состав MySQL средства синхронной репликации не входят, но существуют решения от сторонних компаний. Одним из них является набор заплат, разработанный Google для внутреннего использования.

Компания Google включила в этот набор множество дополнительных функций для MySQL и InnoDB, в частности, полусинхронную репликацию, когда главный сервер не фиксирует транзакцию, пока хотя бы один подчиненный не получит соответствующее событие. Google выпустила свои заплаты для версий MySQL 4.0.26 и 5.0.37. Скачать сами заплаты и ряд относящихся к ним инструментов можно с сайта *http://code.google.com/p/googlemysql-tools*.

Еще один вариант – технология обеспечения высокой доступности, разработанная компанией Solid Information Technology, которая была перенесена для работы с подсистемой хранения solidDB для MySQL. У этого решения есть несколько преимуществ по сравнению с репликацией в MySQL, а именно:

- Подчиненный сервер никогда не отстает от главного.
- Запись на подчиненном сервере осуществляется несколькими потоками, что во многих случаях повышает производительность репликации.
- Уровень надежности репликации задается пользователем. В режиме 1-Safe транзакция возвращает управление сразу после того, как зафиксирована на главном сервере. В режиме 2-Safe транзакция не возвращает управление, пока не будет зафиксирована также на подчиненном сервере, что повышает надежность в случае сбоя.

Однако все это работает только для подсистемы хранения solidDB и не годится для MyISAM, InnoDB и других подсистем. Возможно, компания Solid перенесет еще какие-то части своей технологии в будущих версиях.

Помимо этих двух дополнений к самому серверу MySQL можно воспользоваться ПО промежуточного уровня, например Continuent.

Переключение на резервный и возврат на основной сервер при отказе

Переключением на резервный сервер при отказе (*failover*) называется процедура включения в систему другого сервера взамен вышедшего из строя. Это одна из важнейших составных частей архитектуры обеспечения высокой доступности.

Прежде чем двигаться дальше, определим некоторые термины. Фраза «переключение на резервный сервер при отказе» употребляется в общепринятом смысле; иногда говорят не «failover», а «fallback», подразумевая то же самое. Некоторые употребляют слово «switchover», желая подчеркнуть, что речь идет о запланированном переключении серверов, а не о реакции на отказ.

Мы используем термин «возврат на основной сервер» (failback) для обозначения процедуры, обратной переключению на резервный. Если в систему встроен механизм возврата на основной сервер, то переключение на резервный становится двухэтапным процессом: после того как сервер A отказал и был заменен сервером B, вы можете починить сервер A и вернуть его на роль основного.

Есть много разновидностей переключения на резервный сервер при отказе. Некоторые мы уже обсуждали, поскольку во многих отношениях балансирование нагрузки и переключение на резервный сервер схожи, и граница между ними размыта. Вообще говоря, мы полагаем, что полное решение по переключению на резервный сервер при отказе как минимум обладает средствами мониторинга и автоматической замены сервера. В идеале все это должно быть прозрачно для приложения. Балансирование нагрузки такими возможностями не располагает.

В мире UNIX для переключения на резервный сервер при отказе обычно используются инструменты, разрабатываемые в рамках проекта High Availability Linux (*http://linux-ha.org*), которые, несмотря на название, работают во многих UNIX-подобных операционных системах. Программа отправки тактовых сообщений (heartbeat) лежит в основе мониторинга, а другие инструменты обеспечивают передачу IP-адреса и балансирование нагрузки. Все это можно сочетать с DRBD и LVS.

Самая важная часть мехнизма переключения на резервный сервер при отказе – это возврат. Если невозможно по желанию менять роли серверов, то переключение на резервный сервер – тупик, который только отсрочивает останов. Поэтому нам так нравятся симметричные топологии репликации, например конфигурация с двумя главными серверами, и не нравится топология «кольцо», когда главных серверов три или больше. Если конфигурация симметрична, то переключение и возврат – одна и та же операция, выполняемая в разных направлениях (стоит отметить, что в систему DRBD встроены средства возврата).

Для некоторых приложений критически важно, чтобы переключение и возврат управления производились настолько быстро и атомарно, на-

сколько возможно. Но даже если это не так, все равно не стоит полагаться на то, что вам неподконтрольно, например, на изменение параметров DNS или конфигурационных файлов приложения. Некоторые наиболее серьезные проблемы не проявляются, пока система не станет достаточно велика и такие неприятности, как вынужденный перезапуск приложений или необходимость в атомарной операции, не встанут во весь рост. Всякий, кто хоть раз пытался атомарно обновить код на нескольких серверах, знает, насколько это трудно.

Поскольку балансирование нагрузки и переключение при отказе так тесно связаны, и часто для решения обеих задач применяется одно и то же аппаратное и программное обеспечение, мы рекомендуем выбирать такую методику балансирования нагрузки, которая поддерживала бы и переключение. Именно по этой причине мы всячески избегаем применять для балансирования нагрузки изменение DNS или кода. При подобном подходе вы только создаете себе лишнюю работу: существующий код придется переписать, когда потребуется обеспечить высокую доступность.

В следующих разделах обсуждаются некоторые распространенные методы переключения на резервный сервер при отказе.

Повышение подчиненного сервера, или перемена ролей

Повышение подчиненного сервера до уровня главного, или перемена ролей (активной и пассивной), в топологии репликации главный-главный – важная составная часть многих решений по переключению на резерв при отказе для MySQL. О том, как это сделать, подробно написано в разделе «Смена главного сервера» главы 8 на стр. 474.

Виртуальные IP-адреса или передача IP-адреса

Для обеспечения высокой доступности можно назначить логический IP-адрес экземпляру MySQL, который должен выполнять определенные функции. Если этот экземпляр выйдет из строя, IP-адрес можно будет передать другому серверу MySQL. По существу, это тот же самый подход, который был описан в разделе «Переключение IP-адресов» на стр. 546, только теперь он используется для переключения, а не для балансирования нагрузки.

Достоинство этого подхода в его прозрачности для приложения. Конечно, существующие соединения будут разорваны, но изменять конфигурацию ПО не придется. Кроме того, передачу IP-адреса можно осуществить атомарно, так что все приложения увидят результат изменения одновременно. Это особенно важно, когда сервер «раскачивается» между доступным и недоступным состоянием.

Есть и недостатки:

- Все IP-адреса должны либо находиться в одном и том же сегменте сети, либо необходимо организовать соединение сетей типа мост (bridge).

- Для изменения IP-адреса требуются привилегии суперпользователя.
- Иногда приходится обновлять кэши протокола разрешения адресов (ARP). Некоторые сетевые устройства хранят записи в ARP-кэше слишком долго, поэтому соответствие между IP-адресом и MAC-адресом компьютеров изменяется не мгновенно.
- Сетевое оборудование должно поддерживать передачу IP-адреса. Иногда для правильной работы требуется возможность клонирования MAC-адресов.
- Бывает, что сервер сохраняет за собой IP-адрес, хотя и не функционирует в полном объеме. Тогда необходимо физически останавливать сервер и отключать его от сети.

Плавающие IP-адреса и передача адреса от одного компьютера другому хорошо работают, когда нужно обеспечить перехват управления машиной, которая расположена рядом с отказавшей, то есть в той же самой подсети.

Ожидание распространения изменений

Часто при организации резервирования на некотором уровне приходится ждать, пока нижележащий уровень выполнит изменение. Ранее в этой главе мы уже отмечали, что замена серверов средствами DNS – плохое решение, так как изменения в системе DNS распространяются слишком медленно. Замена IP-адресов дает больший контроль, но распространение IP-адресов в локальной сети зависит от уровня, который находится еще ниже, – протокола ARP.

Диспетчер репликации главный–главный в MySQL

Инструмент MySQL Master-Master Replication Manager (*http://code.google.com/p/mysql-master-master*), или сокращенно *mmm*, представляет собой набор сценариев для мониторинга, переключения при отказе и управления конфигурациями репликации типа главный-главный. Вопреки своему названию он может автоматизировать процесс переключения и для других топологий, в том числе главный-подчиненный или главный-главный с одним или несколькими подчиненными серверами. В этом инструменте используется абстракция *роли*, например читатель или писатель, и комбинация постоянных и плавающих IP-адресов. Он замечает отказ сервера и при необходимости переназначает IP-адреса для «смены ролей». Он также может помочь в реализации switchover для проведения планового обслуживания и в решении других задач.

Типичная конфигурация состоит из пары главных серверов MySQL, на каждом из которых работает процесс *mmmd_agent*. Для каждого процесса необходимо задать кое-какую информацию, в частности IP-

адреса, имена и пароли пользователей. Каждый процесс *mmmd_agent* знает о своем партнере.

Существует также отдельный узел мониторинга. Не следует реализовывать его на том же компьютере, где работает какой-либо из главных серверов. Этот узел наблюдает за обоими серверами и обрабатывает перехват управления, то есть переназначает роль писателя. Всего участвует три виртуальных IP-адреса, по которым можно соединяться с серверами MySQL: два для роли читателя и один для писателя. Для управления экземплярами MySQL и переназначения полей по желанию предназначена программа *mmm_control*.

В целях дальнейшего повышения надежности и доступности инструмент *mmm* можно сочетать с другими методиками (например, с заплатами Google для полусинхронной репликации, о которой упоминалось выше).

Решения на основе посредника

Для реализации переключения на резерв и возврата на основной сервер при отказе можно использовать прокси-сервера, переадресацию портов, трансляцию сетевых адресов (NAT) и аппаратные балансировщики нагрузки. Однако они сами становятся единственными точками отказа, поэтому во избежание проблем их необходимо резервировать.

Среди прочего, с помощью такого рода решения можно сделать так, что удаленный центр обработки данных (ЦОД) будет выглядеть как находящийся в той же сети, что и приложение. Это позволяет использовать такие методы, как плавающие IP-адреса, чтобы приложение начало взаимодействовать с совершенно другим ЦОД. Можно сконфигурировать сервер приложений в каждом ЦОД так, чтобы он подключался к своему собственному посреднику, который маршрутизирует трафик компьютерам в активном ЦОД. Подобная конфигурация изображена на рис. 9.7.

Если активный ЦОД полностью выходит из строя, то посредник может начать маршрутизировать трафик пулу серверов в другом ЦОД, и приложение об этом никогда не узнает.

Основной недостаток такой конфигурации – большая задержка между сервером Apache в одном ЦОД и серверами MySQL в другом. Чтобы сгладить эту проблему, можно запускать веб-сервер в режиме переадресации. В этом случае весь трафик перенаправляется в ЦОД, где находится пул активных серверов MySQL. Того же эффекта можно добиться с помощью HTTP-прокси.

На рис. 9.7 средством соединения серверов MySQL является программа MySQL Proxy, но подойдут и другие архитектуры с посредником, например LVS и аппаратные балансировщики нагрузки.

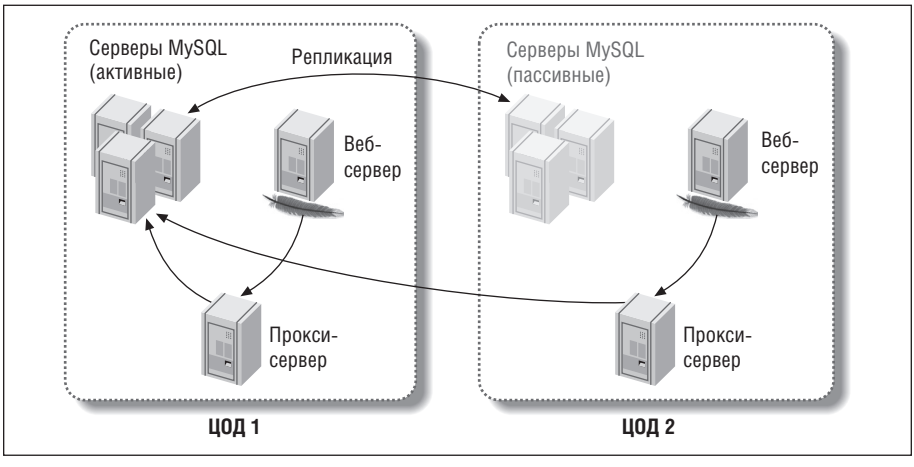


Рис. 9.7. *Использование MySQL Proxy для маршрутизации соединений между центрами обработки данных*

Обработка переключения на резерв при отказе на уровне приложения

Иногда проще или гибче обрабатывать переключение на резерв при отказе на уровне самого приложения. Например, если в приложении возникает ошибка, которая обычно не обнаруживается внешним наблюдателем, скажем, сообщение, свидетельствующее о повреждении данных, то оно может само запустить процедуру переключения.

Хотя идея интеграции процесса переключения при отказе может показаться привлекательной, на самом деле этот подход не очень удачен. Большинство приложений состоит из многих компонентов: задания *cron*, конфигурационные файлы, сценарии на разных языках программирования. Поэтому добавление сюда еще и переключения делает приложение излишне громоздким, особенно по мере его естественного роста и усложнения.

Однако встроить в ПО средство мониторинга и позволить ему *инициировать* процесс переключения – вполне здравая идея. Кроме того, приложение должно уведомлять пользователей о снижении уровня функциональности с помощью соответствующих сообщений.

10

Оптимизация на уровне приложения

Эта книга была бы неполной без главы об оптимизации приложений, обращающихся к серверу MySQL, поскольку именно на них зачастую лежит ответственность за многие проблемы с производительностью, ошибочно приписываемые СУБД. Хотя в основном мы уделяем внимание оптимизации самого сервера MySQL, не хотелось бы упускать из виду и более общую картину. Никакими ухищрениями невозможно оптимизировать MySQL так, чтобы компенсировать неудачную архитектуру приложения. На самом деле, иногда следует выполнять некоторые операции вообще вне MySQL, поручив их самому ПО или другим инструментам, способным предложить гораздо лучшую производительность.

Эта глава не является справочником по построению высокопроизводительного программного обеспечения, но надеемся, что она поможет избежать некоторых типичных ошибок, от которых страдает производительность MySQL. Мы сосредоточимся только на веб-приложениях, так как именно совместно с ними MySQL применяется чаще всего.

Общие сведения о производительности приложений

Стремление к более высокой производительности начинается с простого факта: приложение реагирует слишком медленно, и с этим нужно что-то делать. Но в чем корень зла? Обычные узкие места – это долго выполняющиеся запросы, блокировки, перегрузка процессора, сетевые задержки и файловый ввод/вывод. Любое из них может стать проблемой, если приложение неправильно сконфигурировано или неподобающе использует ресурсы.

Поиск источника проблемы

Первым делом нужно найти виновного. Это будет гораздо проще сделать, если в приложение встроены средства профилирования. Если данное

условие соблюдается, но все равно не видно, в чем причина падения производительности, то, возможно, придется включить дополнительные обращения к профилировщику. Ищите места, где либо используется медленный ресурс, либо к нему обращаются многократно.

Если приложение вынуждено ждать, потому что его производительность ограничена мощностью процессора, и слишком высока степень конкуренции, то причиной замедления может оказаться «потерянное время», о котором мы говорили в разделе «Профилирование приложения» на стр. 87. Поэтому иногда полезно выполнять профилирование в условиях ограниченной конкуренции.

Сетевые задержки могут «съедать» много времени даже в локальной сети. Профилирование на уровне приложения уже включает такие задержки, поэтому профилировщик покажет, как влияют на производительность обращения по сети. Например, если для показа страницы нужно выполнить 1000 запросов, то всего половина миллисекунды на каждое удаленное обращение добавит к времени реакции целых полсекунды. Это очень много для высокопроизводительного ПО.

В случае наличия внутри приложения достаточно подробной системы профилирования найти источник проблемы будет нетрудно. Если такой системы нет, постарайтесь добавить ее. По возможности попробуйте воспользоваться рекомендациями в разделе «Когда не удается добавить код профилирования» главы 2 на стр. 111. Это будет проще и быстрее, чем пустое теоретизирование по поводу того, чем может быть вызвано замедление.

Ищите типичные проблемы

При анализе различных приложений мы постоянно натыкаемся на одни и те же проблемы часто потому, что используются плохо спроектированные готовые системы или популярные среды, упрощающие разработку. Хотя иногда действительно проще и быстрее воспользоваться чем-то написанным другими людьми, на этом пути вас подстерегает опасность: вы не знаете точно, как все устроено внутри. Вот перечень того, на что следует обращать внимание.

- Кто именно потребляет ресурсы процессора, диска, сети и памяти на каждом из составляющих систему компьютеров? Выглядят ли характеристики потребления разумными? Если нет, взгляните пристальнее на приложения, являющиеся основными «пожирателями» ресурсов. Часто решить проблему можно простым изменением конфигурации. Например, если сервер Apache израсходовал всю память, потому что создал 1000 рабочих процессов по 50 Мбайт каждый, то можно сконфигурировать приложение так, чтобы для него требовалось меньше рабочих процессов. Или сконфигурировать систему так, чтобы каждый рабочий процесс потреблял меньше памяти.
- Действительно ли приложение использует все данные, которые получает? Выборка 1000 строк только для того, чтобы вывести 10 из

них и отбросить остальные, – очень распространенная ошибка (однако если приложение кэширует остальные 990 строк, надеясь использовать их в будущем, то это может быть преднамеренной оптимизацией).

- Выполняет ли приложение обработку, которую лучше было бы поручить базе данных, или наоборот? Два примера: выборка всех строк из таблицы для того, чтобы подсчитать их количество, и выполнение сложных манипуляций со строками средствами СУБД. База данных прекрасно справляется с подсчетом строк, тогда как прикладным языкам программирования нет равных в работе с регулярными выражениями. Каждому делу – свой инструмент.
- Не выполняет ли приложение слишком много запросов? Часто следует винить интерфейсы на базе объектно-реляционных отображений (ORM), которые «избавляют программиста от необходимости писать SQL-код». Сервер базы данных изначально спроектирован для сопоставления данных из нескольких таблиц. Уберите вложенные циклы из кода и напишите вместо них запрос с соединением таблиц.
- Не слишком ли мало запросов выполняет приложение? Да, только что мы сказали, что слишком большое количество запросов – это зло. Но иногда «ручное соединение» и другие подобные приемы могут оказаться *благом*, так как позволяют реализовать более детальное и эффективное кэширование, уменьшить количество блокировок (особенно в случае использования MyISAM), а то и ускорить работу приложения за счет эмуляции хеш-соединений в прикладном коде (применяемый в MySQL метод соединения на основе вложенных циклов не всегда эффективен).
- Не подключается ли приложение к MySQL понапрасну? Если можно получить данные из кэша, зачем устанавливать соединение?
- Не подключается ли приложение слишком часто к одному и тому же экземпляру MySQL, быть может, потому, что разные его части открывают собственные соединения? Обычно гораздо лучше использовать одно и то же соединение на протяжении всей работы.
- Не выполняет ли приложение массу «бесполезных» запросов? Типичный пример – выбор требуемой базы данных перед каждым из них. Возможно, было бы лучше всегда подключаться к одной базе, но для таблиц указывать полные имена (заодно упростится анализ запросов в журнале или с помощью команды `SHOW PROCESSLIST`, так как их можно будет выполнить, не изменяя текущую базу). Еще одна типичная проблема – «подготовка» соединения. В частности, драйвер JDBC в языке Java делает во время подготовки массу операций, большую часть которых можно подавить. Другой распространенный источник «бесполезных» запросов – команда `SET NAMES UTF8`, которая в любом случае не нужна (она не изменяет кодировку в клиентской библиотеке, а воздействует только на сервер). Если приложению для

выполнения большей части работы нужна конкретная кодировка, то можно не устанавливать ее при каждом запросе, а назначить в качестве значения по умолчанию.

- Использует ли приложение пул соединений? Это может быть и хорошо, и скверно. С одной стороны, это позволяет ограничить количество соединений, что неплохо, когда на одном соединении выполняется немного запросов (типичный пример – приложения на основе технологии Ajax). С другой стороны, возможны побочные эффекты, например, из-за того что приложение вмешивается в чужие транзакции, видит временные таблицы, изменяет чужие настройки соединения и пользовательские переменные.
- Применяются ли в приложении постоянные соединения? В результате может образовываться слишком много соединений с MySQL. В общем случае так делать не следует, за исключением ситуации, когда подключение к MySQL обходится слишком дорого из-за медленной сети, или когда соединение используется всего для одного-двух быстрых запросов, или когда подключение производится настолько часто, что не хватает номеров локальных портов (см. раздел «Конфигурация сети» главы 7 стр. 410). Если MySQL сконфигурирована правильно, то необходимости в постоянных соединениях может и не возникнуть. Применяйте параметр `skip-name-resolve`, чтобы подавить обратные DNS-запросы и устанавливайте достаточно большое значение параметра `thread_cache`.
- Остаются ли соединения открытыми даже тогда, когда не используются? Если да – особенно при подключении сразу к нескольким серверам, – то, возможно, вы потребляете соединения, которые могли бы пригодиться другим процессам. Пусть, например, приложение подключается к 10 серверам MySQL. Установить 10 соединений из процесса Apache нетрудно, но лишь на одном из них в каждый момент времени происходит что-то полезное. Остальные девять просто проводят время в состоянии `Sleep`. Если один сервер начинает работать медленно или имеет место длительный сетевой вызов, то остальные сервера могут испытывать нехватку соединений. Решение состоит в том, чтобы контролировать использование соединений приложением. Например, можно отправлять каждому экземпляру MySQL пакет запросов и закрывать соединение с ним перед тем, как обращаться к следующим. Если выполняется продолжительная операция, например обращение к веб-службе, то можно даже вообще закрыть соединение с MySQL, дождаться завершения запроса, а потом открыть соединение заново и продолжить работу с базой данных.

Различие между постоянными соединениями и пулом соединений может сбивать с толку. Постоянные соединения могут вызывать те же побочные эффекты, что и пул соединений, поскольку при повторном использовании соединения в любом случае сохраняется его состояние. Однако пул обычно не приводит к открытию очень большого количества

соединений, так как они совместно используются несколькими процессами. С другой стороны, постоянные соединения создаются в рамках одного процесса и не разделяются с другими.

Кроме того, пул позволяет более точно управлять политикой создания соединений. Можно сконфигурировать пул так, что он будет автоматически увеличиваться, но обычно при исчерпании пула новые запросы ставятся в очередь. В этом случае запрос на открытие соединения ожидает на сервере приложений, а не перегружает сервер MySQL.

Существует много способов ускорить выполнение запросов и создание соединений, но общее правило состоит в том, что лучше не ускорять, а попытаться обойтись совсем без них.

Проблемы веб-сервера

Наиболее популярным сервером для создания веб-приложений является Apache. Он хорош для самых разных ситуаций, но при неправильном использовании может потреблять очень много ресурсов. В частности, не следует позволять рабочим процессам жить слишком долго и не нужно эксплуатировать один сервер для всего на свете – лучше оптимизировать различные экземпляры для каждого типа задач.

Обычно Apache запускается с модулями `mod_php`, `mod_perl` и `mod_python` в конфигурации с заранее порожденными рабочими процессами. При этом каждому запросу назначается отдельный процесс. Поскольку сценарии на PHP, Perl и Python могут быть очень требовательны к ресурсам, нередка ситуация, когда каждый процесс потребляет 50, а то и 100 Мбайт памяти. По завершении обработки запроса большая часть памяти возвращается операционной системе, но не вся. Apache не завершает процесс, а повторно использует его для последующих запросов. Следовательно, если следующим поступит запрос на статический ресурс, например CSS-файл или изображение, то этот простенький запрос будет обслуживаться очень «толстым» процессом. Именно поэтому так опасно использовать Apache в качестве универсального веб-сервера. Он *действительно* является универсальным, но специализация позволяет добиться более высокой производительности.

Еще одна серьезная проблема возникает из-за того, что процесс долго остается занятым, если включен режим `Keep-Alive`. Но даже если этот режим не включен, процесс может жить слишком долго, отдавая данные медленному клиенту «в час по чайной ложке»[1].

Часто администраторы совершают ошибку, оставляя включенным подразумеваемый по умолчанию набор модулей Apache. Размер памяти,

[1] Феномен «кормления с ложечки» возникает, когда клиент отправляет HTTP-запрос, но не забирает данные достаточно быстро. Пока клиент не получит все данные, HTTP-соединение, а значит, и процесс Apache, остается открытым.

занимаемый Apache, можно уменьшить, отключив ненужные модули. Сделать это просто: достаточно закомментировать лишние строки в конфигурационном файле Apache, а потом перезапустить его. Можно также удалить ненужные модули PHP из файла *php.ini*.

Подытоживая, повторим, что если Apache запущен в универсальной конфигурации и напрямую обслуживает запросы, поступающие из веб, то может быть порождено много тяжеловесных процессов. Это приводит к расточительному расходованию ресурсов веб-сервера. К тому же эти процессы могут открывать много соединений с MySQL, истощая и его ресурсы тоже. Приведем несколько рекомендаций о том, как уменьшить нагрузку на сервера[1].

- Не используйте Apache для обслуживания статического содержимого или, по крайней мере, используйте для этой цели отдельный экземпляр Apache. Популярными альтернативами являются *lighttpd* и *nginx*.
- Применяйте кэширующий прокси-сервер, например Squid или Varnish, чтобы вообще не допустить запросы до веб-сервера. Даже если на этом уровне невозможно кэшировать страницу целиком, иногда удается кэшировать большую ее часть и воспользоваться технологией Edge Side Includes (ESI, *http://www.esi.org*) для включения небольшого динамического фрагмента в кэшированную статическую часть страницы.
- Задавайте срок хранения для статического и динамического содержимого. Кэширующий прокси-сервер, например Squid, позволяет явно сделать содержимое недействительным. На сайте Википедии эта техника применяется для удаления измененных статей из кэша.
- Иногда, для того чтобы увеличить срок хранения в кэше, требуется внести изменения в приложение. Например, если вы потребуете, чтобы броузер кэшировал CSS и JavaScript-файлы вечно, а затем измените HTML-разметку сайта, то страницы могут прорисовываться неправильно. Для борьбы с этим рекомендуется явно включать в имя файла номер версии. Например, сценарий публикации сайта можно написать так, чтобы CSS-файлы копировались с именем */css/123_frontpage.css*, где *123* – номер редакции в системе Subversion. То же самое относится и к файлам изображений – никогда не используйте повторно одни и те же имена, и после изменения графики страница будет отображаться правильно вне зависимости от того, какой задан срок хранения в кэше броузера.
- Не позволяйте Apache «кормить клиента с ложечки». Мало того, что это медленно, вы еще и распахиваете дверь для атак типа «отказ от

[1] По оптимизации веб-приложений есть хорошая книга Стива Соудерса (Steve Souders) «High Performance Web Sites» (O'Reilly). Хотя она посвящена главным образом созданию быстрых с точки зрения клиента сайтов, предлагаемые рекомендации будут положительно влиять и на сервера.

обслуживания». Аппаратные балансировщики нагрузки обычно буферизуют ответы веб-сервера, поэтому Apache может завершить свою часть работы быстро, а уж балансировщик будет потихоньку отдавать клиенту содержимое из буфера. Можно также поставить перед приложением *lighttpd*, Squid или Apache в режиме управления событиями.

- Включайте *gzip*-сжатие. Для современных процессоров эти накладные расходы ничтожны, зато позволяют сильно сэкономить на трафике. Если вы все же не хотите загружать процессор сжатием, то можете кэшировать сжатые версии страниц и обслуживать их с помощью облегченного веб-сервера, например *lighttpd*.
- Не включайте в Apache режим `Keep-Alive` для соединений на больших расстояниях, поскольку в этом случае процессы Apache будут существовать слишком долго. Лучше поручите заботу о режиме `Keep-Alive` прокси-серверу, который защищает Apache от клиента. Для соединений между Apache и прокси-сервером режим `Keep-Alive` вполне допустим, потому что прокси открывает всего несколько соединений для получения данных от Apache. Это различие иллюстрируется на рис. 10.1.

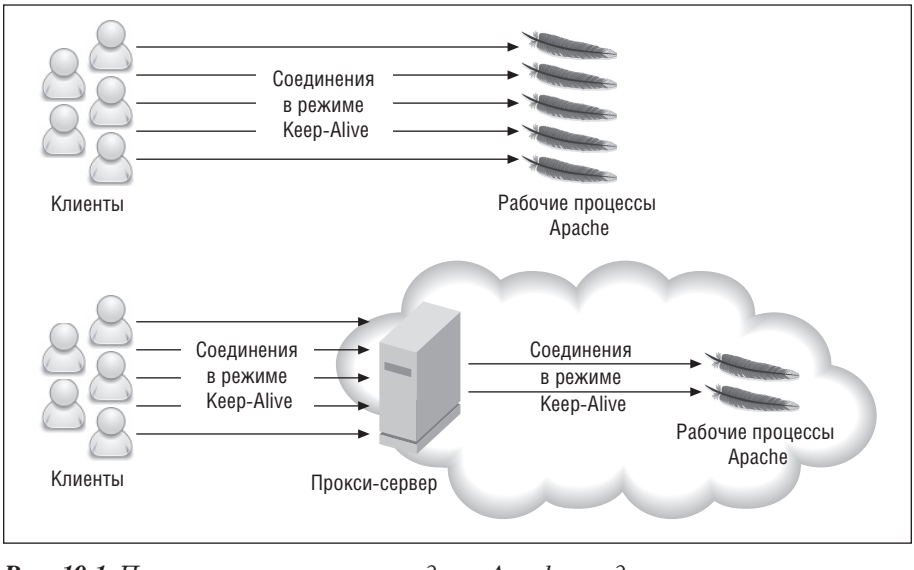


Рис. 10.1. *Прокси-сервер может оградить Apache от долгоживущих соединений в режиме Keep-Alive, что позволяет уменьшить количество рабочих процессов Apache*

Описанные приемы позволят сократить время жизни процессов Apache, так что их будет не больше, чем реально необходимо. Тем не менее, некоторые операции все же могут задержать процесс Apache надолго, потребляя при этом много ресурсов. Примером может служить запрос

к внешнему ресурсу с большой задержкой, в частности, к удаленной веб-службе. Такие проблемы часто оказываются неразрешимыми.

В поисках оптимального уровня конкуренции

У каждого веб-сервера имеется *оптимальный уровень конкуренции*, то есть такое число одновременных соединений, при котором запросы обрабатываются с максимальной скоростью, не перегружая систему. Для отыскания этого «волшебного числа», возможно, придется воспользоваться методом проб и ошибок, но результат того стоит.

Нередко бывает, что сильно нагруженный веб-сервер одновременно обслуживает тысячи соединений. Но лишь малая их часть занята активной обработкой данных. Остальные читают запросы, обеспечивают закачку файлов, заняты «кормлением с ложечки» или просто ждут дальнейших запросов от клиента.

По мере увеличения уровня конкуренции наступает момент, когда сервер достигает пика пропускной способности. После этого она перестает увеличиваться, а часто даже начинает снижаться. Но важнее, что одновременно растет время реакции (задержка).

Чтобы понять, почему так происходит, предположим, что имеется один процессор и сервер одновременно обслуживает 100 запросов. На обслуживание каждого запроса процессор тратит одну секунду. В предположении, что планировщик операционной системы работает вообще без накладных расходов и что на контекстное переключение тоже не тратится время, для обработки всех запросов потребуется 100 секунд процессорного времени.

Как лучше всего обслуживать запросы? Можно выбирать их из очереди один за другим или обрабатывать параллельно, переключаясь с одного на другой и уделяя каждому из них одинаковое количество времени. В любом случае пропускная способность составляет один запрос в секунду. Однако средняя задержка при строго последовательной обработке (уровень конкуренции = 1) равна 50 секундам, а при параллельной (уровень конкуренции = 100) – 100 секундам. На практике при параллельной обработке задержка будет даже выше из-за накладных расходов на переключение контекста.

Если рабочая нагрузка ограничена процессорными ресурсами, то оптимальный уровень конкуренции равен количеству процессоров (или ядер). Однако процесс не постоянно готов к работе, поскольку он выполняет блокирующие системные вызовы, например, запросы ввода/вывода, обращения к базе данных или к сети. Поэтому, как правило, оптимальный уровень конкуренции несколько выше, чем количество процессоров.

Оценить оптимальный уровень конкуренции можно, но для этого потребуется точное профилирование. Обычно проще поэкспериментировать и посмотреть, при каком уровне конкуренции получается пиковая пропускная способность без замедления реакции.

Кэширование

Высоконагруженным приложениям без кэширования не обойтись. Типичное веб-приложение тратит на генерацию страницы гораздо больше ресурсов, чем на ее выборку из кэша (даже с учетом проверки и устаревания данных), поэтому обычно кэширование позволяет повысить производительность на несколько порядков. Сложность кроется в том, чтобы найти правильное сочетание детальности и правил истечения срока хранения. Кроме того, нужно решить, какое содержимое кэшировать и в каком месте.

В типичном высоконагруженном приложении существует много уровней кэширования. Кэширование производится не только на ваших серверах, но и на всем пути следования, включая и броузер пользователя (именно для этой цели предназначены HTTP-заголовки, содержащие указания по сроку хранения содержимого). В общем случае, чем ближе кэш к клиенту, тем больше ресурсов в нем хранится и тем он эффективнее. Извлекать изображение из кэша броузера лучше, чем из памяти веб-сервера, а последнее лучше, чем читать его с диска сервера. Каждому типу кэша присущи свои характеристики, например размер и задержка. Некоторые из них мы рассмотрим в последующих разделах.

Кэши можно подразделить на две большие категории: *пассивные* и *активные*. Пассивный кэш только хранит и возвращает данные. Запрашивая что-то из пассивного кэша, вы либо получаете результат, либо сообщение «такого у меня нет». Примером пассивного кэша является демон *memcached*.

Напротив, активный кэш выполняет какие-то действия, когда искомое не обнаруживается. Обычно он передает поступивший запрос какой-то другой части приложения, которая и генерирует затребованный результат. Затем кэш сохраняет его и возвращает запросившей программе. Кэширующий прокси-сервер Squid – типичный пример активного кэша.

При проектировании приложения обычно стремятся сделать кэши активными (их еще называют *прозрачными*), поскольку они скрывают от программного обеспечения логику проверки, генерации и сохранения. Активные кэши можно строить поверх пассивных.

Кэширование на уровне ниже уровня приложения

У сервера MySQL есть свои внутренние кэши, да и вам никто не запрещает создать собственные кэширующие и сводные таблицы. Кэширующие таблицы можно спроектировать так, чтобы они повышали скорость фильтрации, сортировки, соединения с другими таблицами и решения других специализированных задач. К тому же кэширующие таблицы существуют дольше, чем большинство кэшей на уровне приложения, поскольку они не исчезают при перезапуске сервера.

Кэширование не всегда помогает

Нужно обязательно убедиться, что в результате кэширования производительность действительно повышается, потому что иногда кэш не помогает вовсе. Например, на практике часто быстрее обслужить запрос из памяти *lighttpd*, чем из кэширующего прокси-сервера. Особенно это относится к случаю, когда кэш прокси-сервера хранится на диске.

Причина проста: с кэшированием связаны определенные накладные расходы. Необходимо проверить наличие данных в кэше и обслужить запрос оттуда в случае попадания. Чтобы поместить данные в кэш или сделать их недействительными, тоже нужно время. Кэширование полезно лишь в том случае, когда эти издержки не превышают стоимость генерации и обслуживания страницы без кэша.

Если стоимость всех этих операций известна, то можно подсчитать, в какой мере кэш окажется полезен. Стоимость обслуживания без кэша – это стоимость генерации данных при каждом запросе. Стоимость обслуживания с кэшем равна стоимости проверки наличия данных в кэше плюс вероятность непопадания, умноженная на стоимость генерации данных, плюс вероятность попадания, умноженная на стоимость обслуживания данных из кэша.

Если стоимость с кэшем меньше стоимости без кэша, то производительность может повыситься, хотя это и не гарантируется. Кроме того, имейте в виду, что одни кэши обходятся дешевле, чем другие (пример мы уже приводили – обслуживание из памяти *lighttpd* и из кэша прокси-сервера на диске).

Мы рассказывали об этих стратегиях в главах 3 и 4, поэтому сейчас будем рассматривать только кэширование на уровне приложения и выше.

Кэширование на уровне приложения

Кэш уровня приложения обычно размещается в памяти на том же самом компьютере или в памяти другого компьютера, доступного по сети.

Кэширование на уровне приложения может оказаться более эффективным, чем на нижележащих уровнях, поскольку позволяет приложению сохранять в кэше частично вычисленные результаты. Поэтому кэш избавляет ПО от двух видов деятельности: выборки данных и вычислений с ними. Хороший пример дают блоки HTML-разметки. Приложение может сгенерировать HTML-фрагменты, например заголовки новостей, и поместить их в кэш. При последующих просмотрах страницы нужно будет лишь вставить в нее кэшированный текст. В общем

случае, чем глубже обработка данных перед кэшированием, тем больше времени удается сэкономить при попадании в кэш.

Недостаток состоит в том, что коэффициент попадания в кэш может оказаться низким, а сам кэш будет занимать больше памяти. Предположим, требуется иметь 50 разных вариантов новостных заголовков, чтобы пользователь видел те, что относятся к местности, где он проживает. Тогда кэш займет память, необходимую для хранения всех вариантов, но на каждый вариант будет приходиться меньше попаданий, да и логика удаления из кэша усложнится.

Существует много разных видов кэшей на уровне приложения. Опишем некоторые из них.

Локальные кэши

Такие кэши обычно невелики и существуют только в памяти процесса на протяжении обработки одного запроса. Они полезны, когда нужно избежать повторного обращения к ресурсу, который необходим более одного раза. В таком кэше нет ничего хитроумного: обычно это просто переменная или хеш-таблица в коде приложения. Пусть, например, нужно вывести имя пользователя, зная его идентификатор. Можно написать функцию `get_name_from_id()` и реализовать в ней кэширование, как показано ниже:

```
<?php
function get_name_from_id($user_id) {
   static $name; // ключевое слово static сохраняет
                 // значение переменной между вызовами
   if ( !$name ) {
      // Выбрать из базы данных
   }
   return $name;
}
?>
```

На языке Perl стандартный способ кэширования результатов вызова функций дает модуль `Memoize`:

```
use Memoize qw(memoize);
memoize 'get_name_from_id';
sub get_name_from_id {
   my ( $user_id ) = @_;
   my $name = # получить имя из базы данных
   return $name;
}
```

Эти приемы просты, но позволяют избавить приложение от лишней работы.

Кэши в локальной разделяемой памяти

Это кэши среднего размера (до нескольких гигабайтов), они работают быстро, но с трудом поддаются синхронизации между несколько-

ми машинами. Они хороши для небольших, полустатических фрагментов данных. В качестве примеров можно назвать списки городов в каждом штате, функцию разбиения (таблицу отображения) для секционированного хранилища данных или значения, которые делаются недействительными, исходя из времени жизни (time-to-live – TTL). Основное достоинство разделяемой памяти – очень высокая скорость доступа, обычно гораздо выше, чем для любого удаленного кэша.

Распределенные кэши в памяти

Самый известный пример такого рода – демон *memcached*. Объем распределенных кэшей гораздо больше, чем у кэша в локальной разделяемой памяти, и их размер легко увеличивается. В кэше создается всего одна копия данных, поэтому не нужно тратить время и решать проблемы согласования, возникающие, когда одна и та же информация кэшируется в разных местах. Распределенная память прекрасно подходит для хранения таких разделяемых объектов, как профили пользователей, комментарии и фрагменты HTTP-разметки.

Но время задержки у таких кэшей гораздо выше, чем у кэшей в локальной разделяемой памяти, поэтому наиболее эффективный способ их использования – получение сразу нескольких объектов за одно обращение. Кроме того, вам следует продумать, как будут добавляться новые узлы и что делать, если какой-нибудь узел откажет. В обоих ситуациях приложение должно решить, как распределять или перераспределять объекты между узлами.

Согласованность кэширования важна в случае, когда в кластер кэшей добавляется новый сервер или какой-то сервер из кластера удаляется. Для *memcached* по адресу *http://www.audioscrobbler.net/development/ketama/* имеется библиотека обеспечения согласованности.

Кэши на диске

Диски – медленные устройства, поэтому на них лучше кэшировать объекты, не помещающиеся в памяти, или статическое содержимое (например, предварительно сгенерированные изображения).

Очень полезный прием, применяемый в случае кэширования на диске в контексте веб-сервера, состоит в том, чтобы написать обработчик ошибки 404, который будет перехватывать непопадания в кэш. Предположим, что веб-приложение показывает в заголовке генерируемое на лету изображение, содержащее имя пользователя («С возвращением, Иван!»). На это изображение можно сослаться по адресу */images/welcomeback/john.jpg*. Если оно еще не существует, то возникнет ошибка 404 и будет вызван ее обработчик. Он сгенерирует изображение, поместит его на диск и либо выполнит перенаправление, либо просто вернет броузеру поток байтов, представляющий изображение. Все последующие запросы будут просто возвращать картинку из файла.

Этот прием годится и для многих других типов содержимого. Например, вместо того, чтобы кэшировать заголовки последних новостей в виде блоков HTML, можно сохранить их в виде JavaScript-файла и затем сослаться на этот файл */latest_headlines.js* из заголовка вебстраницы.

Сделать элемент кэша недействительным? Нет ничего проще – достаточно удалить файл. Чтобы реализовать удаление по истечении времени жизни, можно периодически запускать задание, которое будет стирать файлы, созданные более *N* минут назад. А если требуется ограничить размер кэша, то можно реализовать политику вытеснения по давности использования (least recently used – LRU) элементов, удаляя файлы в порядке времени последнего доступа.

Для того чтобы вытеснение по времени последнего доступа работало, необходимо включить в параметрах монтирования файловой системы режим регистрации данного события (для этого нужно лишь опустить флаг `noatime`). Но если вы решите так поступить, то следует пользоваться файловой системой в памяти, иначе диск будет использоваться очень активно. Подробнее этот вопрос освещен в разделе «Выбор файловой системы» главы 7, стр. 414.

Стратегии управления кэшем

Кэшам присуща та же проблема, что денормализованной базе данных: значения дублируются, то есть информацию приходится обновлять в нескольких местах и необходимо принимать меры к тому, чтобы избежать чтения некорректных данных. Ниже описываются некоторые наиболее употребительные стратегии управления кэшем.

Время жизни (TTL)

Вместе с кэшированным объектом хранится момент истечения срока хранения; можно либо удалять все объекты, для которых этот момент уже наступил, с помощью специального процесса, либо дождаться следующего обращения и тогда заменить объект более актуальной версией. Такая стратегия вытеснения больше подходит для редко изменяющихся данных или для которых требования к актуальности не критичны.

Явная инвалидация

Если кэширование устаревших данных неприемлемо, то процесс, который обновляет исходную информацию, может делать недействительной версию в кэше. У такой стратегии есть два варианта: *инвалидировать при записи* и *обновить при записи*. Стратегия инвалидации при записи проста: нужно просто пометить данные в кэше флажком, говорящим, что их срок хранения истек (и, возможно, удалить из кэша). Обновление при записи требует чуть больше работы, поскольку нужно заменить устаревший элемент в кэше обновленным. Но это может дать заметный выигрыш, особенно в тех случаях, когда гене-

рировать кэшируемые значения дорого (а процесс, осуществляющий обновление, возможно, их уже имеет). Если данные в кэше обновлены, то последующие запросы не должны будут ждать, пока приложение сгенерирует их. Если инвалидация производится в фоновом режиме, как, например, в случае инвалидации по времени жизни, то новые версии удаленных элементов можно сгенерировать в процессе, который вообще никак не связан с запросами пользователей.

Инвалидация при чтении

Вместо того чтобы инвалидировать устаревшие данные из кэша в момент изменения исходных значений, можно сохранить информацию, которая позволит определить, истек ли срок хранения данных, в момент чтения их из кэша. По сравнению с явным инвалидированием у этой стратегии есть важное преимущество: фиксированная стоимость операции «размазывается» по времени. Предположим, что изменился объект, от которого зависит миллион других объектов в кэше. Если применять стратегию инвалидации при записи, то придется за один раз сделать недействительными миллион кэшированных элементов, что может занять длительное время, даже если вы придумаете эффективный алгоритм поиска таких элементов. Если же производить инвалидацию при чтении, то запись можно выполнить сразу, а миллион операций чтения отложить на потом. Таким образом, стоимость вытеснения размазывается по времени, и нам удается избежать пиков нагрузки и больших задержек.

Один из самых простых подходов к реализации инвалидации при чтении – *отслеживание версий объектов*. В этом случае вместе с объектом в кэше хранится номер текущей версии или временная метка данных, от которых зависит кэшированный объект. Пусть, например, кэшируется статистика блога пользователя, в том числе и количество отправленных им сообщений. Вместе с объектом `blog_stats` в кэш помещается номер текущей версии пользователя, поскольку статистика зависит именно от него.

При каждом обновлении любых данных, которые также зависят от этого пользователя, обновляется номер его версии. Предположим, что начальный номер версии равен 0, и в этот момент статистика генерируется и сохраняется в кэше. Когда пользователь публикует в своем блоге новое сообщение, номер версии увеличивается на 1 (его следовало бы также сохранить и в самом сообщении, хотя для рассматриваемого примера это необязательно). Позже при отображении статистики мы сравниваем версию объекта `blog_stats` в кэше с версией пользователя. Поскольку номер версии пользователя больше, то мы знаем, что статистика устарела и должна быть вычислена заново.

Это довольно грубый способ инвалидации содержимого, поскольку предполагается, что любые данные, как-то зависящие от пользователя, оказывают влияние на любые другие данные. Не всегда это так. Например, если пользователь редактирует свое сообщение, то номер его вер-

сии будет изменен, что приведет к инвалидации статистики, хотя она (количество сообщений в блоге) и не изменилась. В обмен на такую неточность мы получаем простоту. Незамысловатую стратегию инвалидации не только проще реализовать, не исключено, что и работать она будет также эффективнее.

Отслеживание версий объектов – это упрощенный вариант *тегированного кэша*, который может учитывать и более сложные зависимости. Тегированный кэш знает о разнообразных зависимостях и для каждой из них поддерживает самостоятельные версии. Возвращаясь к примеру с клубом книголюбов из предыдущей главы, можно было бы кэшировать комментарии в зависимости от версии пользователя и версии книги. Для этого нужно было бы снабдить комментарий тегом, состоящим из двух номеров версий: `user_ver=1234` и `book_ver=5678`. Если хотя бы одна версия изменится, то нужно будет обновить комментарий в кэше.

Кэширование иерархий объектов

Иерархическое хранение объектов в кэше может упростить выборку, инвалидацию и способствовать более эффективному использованию памяти. Вместо того чтобы кэшировать только сами объекты, можно кэшировать их идентификаторы, а также группы идентификаторов объектов, которые обычно извлекаются совместно.

Хороший пример применения такой техники дает сайт электронной торговли. Результатом поиска может быть список подходящих товаров, включающий наименования, описания, миниатюры фотографий и цены. Кэшировать список целиком было бы неэффективно: очень может быть, что другой поиск вернет некоторые из уже найденных товаров, так что мы будет хранить дубликаты, впустую транжиря память. Более того, при такой стратегии было бы сложно найти и сделать недействительными те из кэшированных результатов поиска, которые содержат товары с изменившейся ценой, поскольку для этого пришлось бы просматривать все списки без исключения.

Вместо того чтобы кэшировать перечень товаров полностью, можно сохранить лишь минимальную информацию о поиске, например, количество найденных элементов и список их идентификаторов. А каждый товар будет кэшироваться по отдельности. Тем самым мы решаем обе проблемы: результаты не дублируются, и вытеснение можно производить с точностью до отдельного товара.

Недостаток заключается в том, что из кэша приходится извлекать несколько объектов, а не один полный результат поиска. Однако сохранение списка идентификаторов делает эту процедуру достаточно эффективной. При попадании в кэш возвращается список идентификаторов, который затем используется для повторного обращения к кэшу. Результатом второго обращения может быть группа товаров, если кэш позволяет получать несколько объектов за одно обращение (*memcached* поддерживает такую возможность посредством функции `mget()`).

Если проявить беспечность, этот подход может давать странные результаты. Допустим, что применяется стратегия инвалидации результатов поиска по времени жизни, а объекты, описывающие отдельные товары, вытесняются при изменении. Представьте, что произойдет, когда описание товара изменяется таким образом, что перестанет содержать ключевые слова, отвечающие условиям поиска, однако результат поиска еще не удален из кэша. Пользователи увидят устаревшие данные, поскольку кэшированный список идентификаторов ссылается на товар, который больше не удовлетворяет заданным критериям поиска.

Для большинства приложений это не слишком серьезная проблема. Но если для вашего ПО такая ситуация неприемлема, то можно воспользоваться кэшированием с версиями и хранить вместе с результатами поиска версии товаров. Обнаружив поисковый результат в кэше, вы можете сравнить версию товара в нем с текущим (кэшированным) параметром. Если какой-нибудь товар устарел, можно повторить поиск и заново кэшировать результаты.

Предварительная генерация содержимого

Помимо кэширования данных на уровне приложения, можно заранее запрашивать некоторые страницы из фоновых процессов и сохранять полученные результаты в виде статических страниц. Если страницы формируются динамически, то можно заранее сгенерировать какие-то их части, а для построения полной страницы применять технологию включения на стороне сервера (server-side include – SSI) или аналогичную. Это позволит сократить размер и стоимость предварительной генерации содержимого, поскольку в противном случае придется дублировать значительные объемы данных из-за мелких различий в порядке сборки страницы из отдельных частей.

Для кэширования предварительно сгенерированного содержимого может потребоваться много места, и не всегда возможно создать все страницы. Но, как обычно при кэшировании, важно заранее подготовить то содержимое, которое чаще всего запрашивается, поэтому можно применить технику генерации по необходимости с помощью обработчиков ошибки 404 (см. выше в этой главе).

Иногда можно получить выигрыш, организуя хранение предварительно сгенерированного содержимого в файловой системе, размещенной в памяти, чтобы избежать дискового ввода/вывода.

Расширение MySQL

Если MySQL не умеет делать того, что вам надо, можно попытаться расширить его возможности. Мы не станем показывать, как это делается, но упомянем некоторые подходы. Если вы хотите пойти по этому пути, то в сети есть немало отличных ресурсов, и существуют книги, в которых рассматриваются многие из затронутых здесь тем.

Говоря: «MySQL не умеет делать того, что вам надо», мы обычно имеем в виду две вещи: MySQL вообще не умеет этого делать или умеет, но недостаточно быстро или слишком неуклюже. В любом из этих случаев имеет смысл подумать о расширении. К счастью, MySQL становится все более и более модульной и универсальной системой. Например, в версии MySQL 5.1 значительная часть функциональности реализована в виде внешних модулей; даже подсистемы хранения могут быть подключаемыми, поэтому необязательно присоединять их на этапе компиляции сервера.

Подсистемы хранения – отличный способ расширить MySQL для решения узкоспециализированной задачи. Брайан Эйкер (Brian Aker) написал заготовку подсистемы хранения и опубликовал серию статей и презентаций, посвященных началу разработки собственной подсистемы. Они послужили стимулом для создания нескольких значимых подсистем хранения от третьих фирм. Ныне многие компании пишут собственные подсистемы хранения для внутренних целей; в этом легко убедиться, просмотрев список рассылки по внутреннему устройству MySQL. Например, социальная сеть Friendster применяет специальную подсистему хранения для работы с графами, описывающими отношения между людьми. Нам известна также компания, создавшая подсистему хранения для нечеткого поиска. Простенькую подсистему хранения написать не так уж трудно.

Подсистема хранения может использоваться как интерфейс к другой программе. Хороший пример – подсистема Sphinx, которая служит интерфейсом к системе полнотекстового поиска Sphinx (см. приложение С).

В версии MySQL 5.1 допускаются также подключаемые модули для анализа полнотекстовых запросов, а кроме того, можно создавать пользовательские функции (UDF, см. главу 5), способные стать отличным подспорьем для выполнения задач, которые потребляют много процессорного времени, но должны выполняться в контексте потока сервера либо просто реализуются на SQL слишком медленно или неуклюже. Такие функции можно использовать для администрирования, интеграции со службами, для получения информации от операционной системы, обращения к веб-службам, синхронизации данных и т. п.

MySQL Proxy – еще одна область приложения сил для тех, кто хочет расширить функциональность протокола MySQL. А созданная Полом МакКаллаги (Paul McCullagh) масштабируемая инфраструктура потоковых BLOB'ов (*http://www.blobstreaming.org*) открывает целый спектр новых возможностей для хранения больших объектов в MySQL.

Поскольку MySQL – бесплатное ПО с открытым исходным кодом, то вы можете даже подправить программный код сервера, если он не делает того, что вам надо. Например, нам известны компании, которые расширили грамматику языка SQL. В последние годы сторонние разработчики предложили немало интересных дополнений MySQL в области профилирования производительности, масштабируемости и новых функ-

ций. Создатели MySQL всегда готовы помочь тем, кто собирается расширить MySQL. К ним можно обратиться с помощью списка рассылки *internals@lists.mysql.com* (подписаться на него можно на сайте *http://lists.mysql.com*), на форумах по MySQL или через IRC-канал *#mysql-dev* на сервере freenode.

Альтернативы MySQL

MySQL не может удовлетворить все мыслимые потребности. Порой гораздо лучше решать некую задачу вне MySQL, даже если теоретически эта СУБД способна с ней справиться.

Один из самых очевидных примеров – хранение данных в традиционной файловой системе, а не в таблицах. Классический случай – файлы изображений: их можно поместить в столбец типа BLOB, но это редко бывает оправдано[1]. Обычно изображения и другие большие двоичные файлы размещают в файловой системе, а внутри MySQL хранят только их имена; приложение может получить файл в обход MySQL. В веб-приложении это достигается записью имени файла в атрибут `src` тега `<img>`.

Полнотекстовый поиск также лучше реализовывать внешними средствами, поскольку MySQL не способен делать это так хорошо, как, скажем, системы Lucene или Sphinx (см. приложение С).

Для некоторых задач полезен NDB API. Например, хотя подсистема хранения NDB Cluster пока не пригодна для хранения всех данных в высокопроизводительном веб-приложении, ее все же можно использовать в качестве хранилища информации о сеансах или регистрационных данных пользователей. Подробнее о NDB API можно прочитать на странице *http://dev.mysql.com/doc/ndbapi/en/index.html*. Существует также модуль NDB для Apache, *mod_ndb*, который можно скачать с сайта *http://code.google.com/p/mod-ndb/*.

Наконец, для некоторых операций, например обхода деревьев и работы с графами, реляционная база данных просто не предназначена. MySQL плохо приспособлена для взаимодействия с распределенными данными, поскольку ему недостает средств распараллеливания обработки запросов. По-видимому, для этой цели стоит поискать другие инструменты (возможно, в сочетании с MySQL).

[1] Один из случаев, когда хранение файлов в MySQL оправдано, – это использование репликации для быстрой доставки изображений на множество машин, и мы знаем несколько приложений, в которых такая техника применяется.

11

Резервное копирование и восстановление

Очень легко сосредоточить все усилия на том, чтобы «получить нечто работающее», забыв о резервном копировании и восстановлении. Срочное не всегда бывает важным, а важное часто не срочно. Резервное копирование имеет большое значение для обеспечения высокой производительности, а также для восстановления после аварии. Необходимо с самого начала планировать и прорабатывать то, как будет выполняться резервное копирование, чтобы оно не приводило к простоям и снижению производительности.

Если вы не запланировали резервное копирование на ранних этапах проекта, то потом его обычно приходится «прикручивать» наспех. И в этот момент может обнаружиться, что принятые ранее решения исключают самые высокопроизводительные методы подготовки резервных копий. Например, вы уже сконфигурировали сервер, а потом осознали, что на самом деле надо бы воспользоваться LVM для снятия мгновенных снимков файловой системы, но уже поздно. Кроме того, может выясниться, что настройка системы для резервного копирования оказывает серьезное влияние на производительность. И если заранее не сформировать план восстановления и не испытать его на практике, то в момент аварии все пойдет совсем не так гладко, как хотелось бы.

Системы резервного копирования похожи на системы мониторинга и оповещения: большинство системных администраторов время от времени изобретают их заново. Это позор, потому что существуют хорошие, отлично поддерживаемые и гибкие программы для резервного копирования, причем некоторые из них бесплатны и поставляются с открытым исходным кодом. Мы призываем использовать те части таких систем, которые применимы в ваших условиях.

Мы не собираемся рассматривать в этой главе все компоненты хорошо спроектированного решения по резервному копированию и восстановлению. Эта тема заслуживает отдельной книги, и такие книги уже написаны[1]. По некоторым вопросам мы пройдем вскользь, а сосредоточимся на аспектах, относящихся к достижению высокой производительности MySQL. В отличие от первого издания настоящей книги, мы предполагаем, что многие читатели используют InnoDB в дополнение или вместо MyISAM. В этом случае отдельные сценарии резервного копирования усложняются.

Обзор

Мы начнем эту главу с обзора терминологии, а также обсуждения возможных требований и различных проблем, о которых следует помнить при планировании резервного копирования и восстановления. Затем мы представим обзор разнообразных технологий и методов создания резервных копий и рассмотрим, как восстановить данные после аварии. Наконец, мы уделим внимание существующим инструментам резервного копирования и завершим главу некоторыми примерами создания собственных утилит для той же цели.

Терминология

Прежде всего, уточним основные термины. Вы наверняка слышали о так называемом горячем, теплом и холодном резервном копировании. Обычно этими словами описывается влияние резервного копирования на текущую работу; например, «горячее» резервное копирование означает, что сервер не нужно останавливать. Проблема в том, что не все понимают эти термины одинаково. В названиях некоторых инструментов даже присутствует слово «hot» (горячий), хотя они определенно не выполняют то, что мы понимаем под горячим резервным копированием. Мы стараемся избегать таких понятий, а вместо этого говорить о том, в какой мере конкретная техника или инструмент прерывают работу сервера.

Еще два сбивающих с толку слова – *restore* (*возвращать*) и *recover* (*восстанавливать*). В этой главе им придается вполне конкретный смысл. Под возвратом мы понимаем извлечение данных из резервной копии и либо загрузку их в MySQL, либо размещение файлов там, где MySQL ожидает их найти. Под восстановлением обычно понимается весь процесс приведения системы или ее части в рабочее состояние, после того как что-то пошло не так. Сюда входит не только возврат данных из ре-

[1] Полагаем, что стоит обратить внимание на книгу У. Кэртис Престон (W. Curtis Preston) «Backup & Recovery» (издательство O'Reilly).

зервной копии, но и все остальные шаги, необходимые для возобновления функционирования в полном объеме, в частности перезапуск MySQL, изменение конфигурации, прогрев кэшей сервера и т. д.

Для многих восстановление означает лишь исправление поврежденных в результате сбоя таблиц. Но это совсем не то же самое, что восстановление всего сервера. Процедура восстановления подсистемы хранения должна привести в соответствие файлы данных и журналов. Она также должна гарантировать, что в файлах данных отражены только модификации, произведенные зафиксированными транзакциями, и накатить те транзакции из журналов, которые еще не были применены к файлам данных. При использовании транзакционной подсистемы хранения это может быть частью общего процесса восстановления или даже частью резервного копирования. Однако это не то восстановление, которое может потребоваться после случайного выполнения команды `DROP TABLE`.

И это все о восстановлении

Пока все идет хорошо, о восстановлении как-то не задумываешься. Но зато когда момент настанет, не спасет даже самая лучшая в мире система резервного копирования. Вместо этого понадобится хорошая система восстановления.

Проблема в том, что организовать гладкий процесс резервного копирования проще, нежели создать хорошие процедуры и инструменты восстановления. И вот почему:

- Резервное копирование происходит раньше. Если нет резервной копии, то не с чего восстанавливаться, поэтому при построении системы внимание уделяют, прежде всего, резервному копированию. Важно пойти наперекор этому естественному желанию и сначала спланировать процедуру восстановления. Более того, вообще не следует строить систему резервного копирования, пока не определены требования к восстановлению.
- Резервное копирование – регулярно выполняемая процедура. Поэтому вы думаете главным образом о том, как автоматизировать и настроить именно ее, часто даже не отдавая себе в этом отчета. Пять минут в день на улучшение процедуры копирования – вроде бы пустяк, но уделяете ли вы ежедневно такое же внимание восстановлению? Вы должны сознательно отлаживать процедуру восстановления до тех пор, пока она не будет проходить без сучка без задоринки, как и резервное копирование.
- Снятие резервной копии обычно происходит безо всякого стресса, а восстановление, как правило, в кризисной ситуации. Важность этого замечания трудно переоценить.

- Часто в дело вмешиваются соображения безопасности. Если резервное копирование производится за пределы рабочей площадки, то, скорее всего, данные на копии шифруются или защищаются как-то иначе. Очень легко рассуждать о том, какой ущерб нанесет компрометация данных, и совсем упустить из виду, что произойдет, если никто не сможет прочитать зашифрованный том, чтобы восстановить с него данные, или позабыть о том, чего стоит извлечь один файл из монолитного зашифрованного файла-архива.
- Один человек может спланировать, спроектировать и реализовать процедуру резервного копирования, особенно при наличии подходящих инструментов. Но его может не оказаться на месте, когда грянет гром. Необходимо обучить нескольких людей и позаботиться о том, чтобы кто-нибудь из них постоянно присутствовал, но не поручать восстановление данных неквалифицированному сотруднику.

Вот пример из реальной жизни: некий заказчик сообщил, что резервное копирование стало выполняться с молниеносной скоростью, стоило указать при запуске *mysqldump* флаг *-d*, и поинтересовался, почему никто не сообщил ему, что этот флаг может настолько ускорить процедуру. Если бы он попытался вернуть данные с таких копий, то очень быстро понял бы причину: при наличии флага *-d* данные вообще не копируются! Заказчика интересовало только резервное копирование, а не восстановление, поэтому он даже не ведал о грозящей беде.

Начав задумываться о восстановлении, первым делом сформулируйте свои требования. Вот что стоит принять во внимание:

- Сколько данных вы можете потерять без серьезных последствий? Понадобится ли восстановление на конкретный момент времени в прошлом или допустимо потерять всю работу с момента снятия последней резервной копии? Есть ли какие-то законодательные требования?
- Насколько быстрым должно быть восстановление? Каково допустимое время простоя? С чем будут готовы смириться приложение и пользователи (например, с частичной недоступностью) и как вы собираетесь реализовать продолжение функционирования в таких условиях?
- Что необходимо восстанавливать? Обычно речь идет о всем сервере, об отдельной базе данных, об отдельной таблице или о конкретных транзакциях и командах.

Запишите свои ответы на эти вопросы, включите их в документацию по системе и держите в уме по ходу чтения настоящей главы. Это упражнение поможет вам помнить о восстановлении при планировании резервного копирования. А документация пригодится, когда позже возникнет необходимость повторить тот же путь.

Миф о резервном копировании № 1: я использую репликацию как резервное копирование

Ох, как часто приходится сталкиваться с этой ошибкой! Подчиненный сервер не может служить заменой резервной копии. И RAID-массив тоже. Чтобы понять почему, задумайтесь над таким вопросом: помогут ли они восстановить данные после случайного выполнения команды `DROP DATABASE` для промышленной базы данных? Ни RAID-массив, ни репликация не пройдут этого простейшего теста. Они не только не являются технологией резервного копирования, но даже заменить резервную копию не могут. Ничто, кроме самого резервного копирования, не способно реализовать функции резервного копирования.

О чем мы не расскажем

Резервное копирование MySQL во многих отношениях не более чем частный случай общей задачи резервного копирования и восстановления. Мы хотели бы сосредоточиться только на вопросе об обеспечении высокой производительности MySQL, но довольно трудно не касаться множества других тем, особенно учитывая, как часто мы встречали людей, упорно пытающихся решить одни и те же проблемы резервного копирования и восстановления.

Но все же перечислим темы, которые мы решили не включать в настоящее издание:

- Безопасность (кто может иметь доступ к резервным копиям, кто обладает привилегиями для возврата данных с копий, следует ли шифровать файлы с резервными копиями).
- Где хранить резервные копии, в частности, насколько они должны быть удалены от источника (на другом диске, на другом сервере, на другой площадке), и как доставлять данные из источника в место назначения.
- Установление срока хранения, аудит, законодательные требования и связанные с этим вопросы.
- Системы хранения и носители, сжатие и инкрементное копирование.
- Форматы хранения (однако мы твердо говорим: избегайте патентованных форматов резервного копирования).
- Мониторинг и отчеты о резервном копировании.
- Средства резервного копирования, встроенные в системы хранения или конкретные устройства, например специализированные файловые сервера.

Все это важные темы. Если вы с ними не знакомы, прочитайте какую-нибудь книгу о резервном копировании.

Общая картина

Прежде чем начинать подробное описание имеющихся вариантов, выскажем свое мнение о том, что большинству людей нужно от решения по резервному копированию и восстановлению. Можете считать эти рекомендации либо отправной точкой, либо направлением, в котором следует работать дальше.

- Для больших баз данных физическое резервное копирование совершенно необходимо: этот механизм работает быстро, что крайне важно. Лично мы предпочитаем резервные копии на основе снимков, но технология InnoDB Hot Backup – вполне приемлемая альтернатива, если вы работаете только с таблицами типа InnoDB.
- Включайте в резервную копию двоичные журналы, чтобы оставалась возможность восстановления на конкретный момент времени в прошлом.
- Сохраняйте несколько поколений резервных копий, а файлы двоичных журналов храните достаточно долго для того, чтобы из них можно было вернуть данные.
- Периодически проверяйте процедуру резервного копирования и восстановления, проходя весь процесс восстановления от начала до конца.
- Периодически создавайте логические резервные копии (возможно, из физических копий для эффективности). Следите за тем, чтобы было достаточно двоичных журналов для восстановления из последней логической копии.
- Если возможно, проверяйте физические резервные копии, чтобы точно знать, можно ли будет восстановиться с них. Старайтесь осуществлять такую проверку еще в процессе резервного копирования, до отправки в место хранения.
- Не упускайте из виду безопасность. Что случится, если кто-то скомпрометирует сервер, – сможет ли он получить доступ также и к серверу резервных копий? А наоборот?
- Ведите мониторинг процесса резервного копирования и самих резервных копий независимо от тех инструментов, которые применяются для резервного копирования. Необходимо внешнее подтверждение их пригодности.
- Обращайте внимание на то, как файлы копируются с одной машины на другую. Существуют куда более эффективные методы копирования, чем команды *scp* или *rsync*. Дополнительную информацию об этом см. в приложении А.

Зачем нужны резервные копии?

Если вы создаете высокопроизводительную систему на основе MySQL, то без резервных копий не обойтись. Назовем лишь несколько причин.

Восстановление после аварии

Это то, что вы делаете в случае, если вышло из строя оборудование, когда из-за какой-то нелепой ошибки оказались поврежденными данные, либо когда по той или иной причине сервер и хранящаяся на нем информация стали недоступны или непригодны к использованию (потенциальных причин множество, включите воображение). Вероятность какой-то одной конкретной аварии довольно мала, но ведь они суммируются. Нужно быть готовым ко всему: случайному подключению не к тому серверу и выполнению команде ALTER TABLE[1], пожару в здании, злонамеренным действиям хакера, ошибке в коде MySQL.

Человек передумал

Вы удивитесь, узнав, как часто нам встречалась ситуация, когда необходимо хотя бы частично восстановить состояние данных на какой-то момент в прошлом. В некоторых приложениях это встречается даже чаще, чем аварии (например, когда важный клиент сознательно удаляет какие-то данные, а затем хочет вернуть их обратно).

Аудит

Иногда нужно знать, как выглядели данные или схема в какой-то момент в прошлом. Например, вы можете оказаться участником судебного процесса или обнаружить в приложении ошибку и задаться вопросом, что когда-то делала программа (иногда версии кода, хранящейся в системе управления версиями, бывает недостаточно).

Тестирование

Один из самых простых способов протестировать приложение на реальных данных состоит в том, чтобы периодически «заливать» на тестовый сервер свежую информацию с промышленного. Если имеется резервная копия, то достаточно просто восстановить с нее данные.

Проверьте справедливость своих допущений. Например, считаете ли вы, что поставщик совместного хостинга регулярно делает резервные копии сервера MySQL, который предоставляется вам по договору? Хотя разделяемый хостинг имеет мало общего с высокой производительностью, мы хотим подчеркнуть, что подобные непроверенные предположения могут привести к печальным последствиям. Если вам интересно,

[1] Бэйрон никак не может забыть подобный случай, случившийся с ним во время работы над сайтом электронной торговли, когда он по ошибке набрал команду не в том окне. Это был недосмотр администраторов; они не должны были давать разработчикам повышенные привилегии на промышленных серверах. Я не шучу!

сообщаем, что многие поставщики хостинга не занимаются резервным копированием MySQL, тогда как другие просто копируют файлы на работающем сервере, поэтому созданная копия, скорее всего, окажется бесполезной.

Различные факты и компромиссы

Резервное копирование MySQL сложнее, чем кажется на первый взгляд. Очень упрощая, можно сказать, что резервная копия – это просто копия данных, но из-за требований приложения, архитектуры подсистем хранения в MySQL и конфигурации вашей системы получить такую копию может оказаться совсем нелегко.

Что вы можете позволить себе потерять?

Если известно, сколько данных можно относительно безболезненно потерять, то к выработке стратегии резервного копирования можно подойти более осмысленно. Нужна ли возможность воссоздания баз на конкретный момент времени или достаточно восстановить данные с копии, снятой прошлой ночью, смирившись с потерей всей проделанной с тех пор работы? Если восстановление на конкретный момент времени необходимо, то, вероятно, придется осуществлять регулярное копирование и включить режим записи в двоичный журнал, чтобы можно было вернуть данные, а затем накатить на них журналы до требуемого момента.

Вообще говоря, чем больше данных вы готовы потерять, тем проще процедура резервного копирования. Если же требования очень жесткие, то гарантировать воссоздание всей информации гораздо сложнее. Существуют даже разные варианты восстановления на конкретный момент времени. «Мягкие» требования означают допустимость воссоздания данных в состоянии, «достаточно близком» к моменту сбоя. «Жесткие» требования подразумевают необходимость восстановить все зафиксированные транзакции, даже если произошло что-то ужасное (например, сервер в буквальном смысле сгорел). Для этого применяют специальные методы, например, хранят двоичные журналы на отдельном SAN-томе или используют DRBD-репликацию диска. Подробнее об этом написано в главе 9.

Оперативное или автономное резервное копирование?

Если останов сервера MySQL на время резервного копирования допустим, то это самый безопасный и вообще наилучший способ получить копию данных с минимальными шансами внести искажения или несогласованность. Если сервер остановлен, то информацию можно копировать, не заботясь о таких неприятностях, как «грязные» буферы в пуле буферов InnoDB или в других кэшах. Не нужно беспокоиться о том, что данные модифицируются в процессе резервного копирования, а по-

скольку сервер не испытывает никакой нагрузки со стороны приложения, то копирование происходит быстрее.

Однако вывод сервера из эксплуатации обходится дороже, чем может показаться. Даже если удается свести к минимуму само время простоя, останов и запуск MySQL могут занять довольно много времени, если нагрузка и объем данных велики:

- Если в пуле буферов InnoDB много «грязных» буферов, то есть велик объем данных, которые в памяти уже модифицированы, но на диск еще не записаны, то для их сохранения у InnoDB может уйти много времени. Время останова InnoDB можно контролировать с помощью конфигурационной переменной `innodb_fast_shutdown`, которая определяет режим работы с пулом буферов и буфером вставки[1] в ходе процедуры останова, но это лишь переносит работу на другое время, а не устраняет ее необходимость. Поэтому существенно сократить общее время останова и запуска таким образом не удастся. Иногда это можно сделать путем настройки других аспектов InnoDB, но такие изменения конфигурации оказывают гораздо более широкое влияние на производительность. Дополнительную информацию по этому поводу см. в разделе «Настройка ввода/вывода в MySQL» в главе 6, стр. 351.
- Перезапуск тоже может занять много времени. Открытие всех таблиц и прогрев кэшей – длительный процесс, особенно если таблиц и данных много. Если присвоить параметру `innodb_fast_shutdown` значение 2, то останов InnoDB происходит быстро, зато во время запуска подсистемы хранения инициирует полную процедуру восстановления. И даже после того, как сервер запустился, пройдет еще немало времени, прежде чем он как следует прогреется и будет готов к полноценной работе.

Следовательно, стремясь к достижению высокой производительности, вы почти наверняка захотите спроектировать процедуру резервного копирования так, чтобы не переводить промышленный сервер в автономный режим. Но в зависимости от требований, предъявляемых к согласованности данных, резервное копирование в оперативном режиме все равно может означать прерывание обслуживания на заметное время.

Например, один из наиболее часто упоминаемых методов резервного копирования начинается с выполнения команды `FLUSH TABLES WITH READ LOCK`. Тем самым мы говорим MySQL, чтобы он сбросил[2] и заблокировал все таблицы, а также опустошил кэш запросов. На это требуется время:

---

[1] Буфер вставки хранится в виде файлов в табличном пространстве InnoDB вместе с другими данными; рано или поздно фоновый поток помещает вставленные записи в те таблицы, где им положено находиться.

[2] Операция сброса таблиц сбрасывает данные на диск в случае использования подсистемы хранения MyISAM, но не InnoDB.

сколько конкретно, предсказать невозможно; оно окажется большим, если для получения глобальной блокировки чтения придется ждать завершения выполнения длительной команды или если количество таблиц велико. Пока блокировки не будут освобождены, изменение данных на сервере невозможно. Команда `FLUSH TABLES WITH READ LOCK` обходится не так дорого, как останов сервера, потому что большая часть данных остается кэшированной в памяти, и сервер «прогрет», но все равно она мешает нормальной работе.

Если это проблема, то придется искать альтернативу. Например, один из применяемых нами методов состоит в том, чтобы делать резервную копию на подчиненном сервере репликации, который входит в пул серверов, так что его можно останавливать и перезапускать со сравнительно небольшими издержками. Мы еще вернемся к этому вопросу и к другим соображениям, касающимся оперативного и автономного резервного копирования. А пока скажем лишь, что в современных версиях MySQL реализовать оперативное резервное копирование без прерывания обслуживания затруднительно.

Логическое и физическое резервное копирование?

Существует два основных способа резервного копирования данных в MySQL: *логическое* (такая копия называется также «дампом») и путем копирования *исходных файлов*. В логической резервной копии данные представлены в формате, который MySQL может интерпретировать как SQL-команды или как текст с разделителями[1]. Исходные файлы – это просто файлы в том виде, в каком они находятся на диске.

У каждого способа копирования данных есть свои плюсы и минусы.

Логические резервные копии

У логических резервных копий есть следующие достоинства:

- Это обычные файлы, которые можно обрабатывать с помощью стандартных текстовых редакторов и таких инструментов командной строки, как *grep* или *sed*. Это очень полезно, когда требуется вернуть информацию или просто просмотреть ее глазами.
- Из них легко восстанавливать данные. Достаточно просто подать файл по конвейеру на вход программы *mysql* или воспользоваться программой *mysqlimport*.

[1] Логические резервные копии, создаваемые программой *mysqldump*, не всегда являются текстовыми файлами. SQL-дамп может содержать текст в различных кодировках и даже двоичные данные, которые вообще не соответствуют печатаемым символам. Да и строки могут оказаться слишком длинными для многих редакторов. И все же, как правило, такие файлы можно открыть и прочитать в текстовом редакторе, особенно если *mysqldump* запускалась с флагом *--hex-blob*.

- Резервное копирование и возврат данных можно выполнять по сети, то есть не на той же машине, где работает сервер MySQL.
- Процедуру можно очень гибко настраивать, потому что *mysqldump* – инструмент, которым многие предпочитают пользоваться для снятия логических копий, – принимает множество параметров, например фразу `WHERE`, позволяющую указать, какие строки включать в резервную копию.
- Они не зависят от подсистемы хранения. Поскольку для создания логической копии данные запрашиваются у сервера MySQL, то различия между подсистемами хранения нивелируются. Следовательно, очень просто снять копию таблицы типа InnoDB и восстановить ее в таблицу типа MyISAM. С физическими резервными копиями такой фокус не пройдет.
- Задав подходящие флаги при вызове *mysqldump*, во многих случаях можно даже импортировать логическую копию в другую СУБД, например PostgreSQL.
- Они могут помочь избежать повреждения данных. Если диски сбоят, то при копировании физических файлов получится поврежденная резервная копия, и если вы специально не проверите ее по окончании резервного копирования, то узнаете об этом только тогда, когда не сможете воспользоваться ей в целях восстановления. Но если данные MySQL в памяти *не* повреждены, то иногда можно получить заслуживающую доверия логическую копию, когда снять хорошую физическую не удается.

Но у логических резервных копий есть и недостатки:

- Для их генерации требуется работа сервера, так что процессор загружается сильнее.
- В некоторых случаях логические копии оказываются объемнее исходных физических файлов[1]. ASCII-представление данных не всегда так же эффективно, как внутреннее представление в подсистеме хранения. Например, для хранения целого числа необходимо 4 байта, но для записи его в виде ASCII-текста может понадобиться до 12 символов. Зачастую логические копии хорошо поддаются сжатию, но для этого нужно дополнительное процессорное время.
- Потеря точности в представлении чисел с плавающей точкой может помешать правильному восстановлению данных из файлов дампа (в состав заплаты Google для MySQL входит доработка программы *mysqldump*, решающая эту проблему).

[1] Наш опыт показывает, что обычно логические резервные копии меньше физических, но так бывает не всегда.

- Для восстановления данных из логической копии необходимо загружать и интерпретировать команды и перестраивать индексы, что возлагает на сервер еще больше работы.

Но самый главный недостаток – стоимость выгрузки данных из MySQL и обратной их загрузки с помощью команд SQL.

Физические резервные копии

У физических резервных копий есть следующие достоинства:

- Для получения физической копии нужно просто скопировать требуемые файлы в другое место. Никакой дополнительной работы для их генерации выполнять не придется.
- Трудоемкость возврата данных из физической копии может быть проще и зависит от подсистемы хранения. В случае MyISAM достаточно просто скопировать файлы в исходное место, тогда как для InnoDB требуется остановить сервер и, возможно, предпринять еще какие-то действия.
- Вообще говоря, физические копии можно переносить между платформами, операционными системами и версиями MySQL.
- Восстановление с физических копий может оказаться быстрее, потому что серверу не нужно выполнять SQL-команды и строить индексы. При наличии таблиц InnoDB, которые не помещаются целиком в память сервера, восстановление данных из физических файлов может быть *гораздо* быстрее.

А вот перечень недостатков:

- Объем физических файлов InnoDB, как правило, гораздо больше соответствующих логических копий. Обычно в табличном пространстве InnoDB очень много неиспользуемого места. К тому же какое-то пространство отведено под цели, не связанные с хранением табличных данных (буфер вставки, сегмент отката и т. д.).
- Не всегда физическую копию можно перенести на другую платформу, операционную систему или версию MySQL. В частности, препятствием может стать чувствительность к регистру букв и формат чисел с плавающей точкой. Перенос файлов в систему с другим форматом чисел с плавающей точкой вообще невозможен (впрочем, в большинстве современных процессоров применяется формат IEEE).

Работать с физическими копиями обычно проще и эффективнее. Но не следует полностью полагаться на них для долговременного хранения или удовлетворения требований законодательства; время от времени следует делать и логические копии.

Не считайте резервную копию (особенно физическую) пригодной для использования, пока не проверили ее. В случае InnoDB это означает, что нужно запустить экземпляр MySQL, дать InnoDB завершить процедуру

восстановления, а затем выполнить команду CHECK TABLES. Можно опустить этот шаг или просто проверить файлы утилитой *innochecksum*, но мы не советуем так делать. Для MyISAM следует выполнить команду CHECK TABLES или воспользоваться утилитой *myisamchk*.

Можно также пойти на хитрость и скомбинировать оба подхода: снять физические копии, а потом запустить экземпляр MySQL и с его помощью создать из них логические копии. В результате вы получаете все лучшее от обоих подходов, не создавая на промышленном сервере излишнюю нагрузку по формированию дампа. Это особенно удобно, когда имеется возможность делать мгновенные снимки файловой системы, – вы создаете снимок, копируете его на другой сервер, разворачиваете, а затем проверяете физические файлы и выполняете логическое резервное копирование.

Что копировать?

Требования к восстановлению диктуют, что нужно копировать. Простейшая стратегия состоит в том, чтобы скопировать данные и определения таблиц, однако это лишь необходимый минимум. Обычно для полного восстановления промышленного сервера требуется гораздо больше. Перечислим кое-что из того, что стоит включать в состав резервной копии MySQL.

Неочевидные данные

Не забудьте о данных, которые не бросаются в глаза, например: двоичные журналы и журналы транзакций InnoDB.

Код

В современном сервере MySQL может содержаться много программного кода, в частности, триггеры и хранимые процедуры. Если вы выполняете резервное копирование базы данных mysql, то большая часть этого кода в нее войдет. Однако тогда становится трудно восстановить единственную базу из всего множества, поскольку часть «данных» в этой базе, к примеру хранимые процедуры, на самом деле содержится в базе mysql.

Конфигурация репликации

Для восстановления сервера, участвующего в репликации, следует включать в резервную копию все необходимые для репликации файлы: двоичные журналы, журналы ретрансляции, индексные файлы журналов и info-файлы. Как минимум, следует добавить результаты выполнения команды SHOW MASTER STATUS и/или SHOW SLAVE STATUS. Полезно также выполнить команду FLUSH LOGS, чтобы MySQL начала новый двоичный журнал. Восстановление на конкретный момент времени проще делать, когда отсчет ведется от начала, а не от середины журнала.

Конфигурация сервера

Если потребуется восстановить данные после настоящей катастрофы, скажем, выстроить сервер с нуля в новом центре обработки данных после землетрясения, то вы оцените полезность хранения конфигурационных файлов сервера в составе резервной копии.

Отдельные файлы операционной системы

Как и в случае с конфигурацией сервера, очень важно сохранить все внешние конфигурационные файлы, существенные для работы операционной системы. На UNIX-сервере это могут быть таблицы *cron*, конфигурация пользователей и групп, административные сценарии и правила *sudo*.

Подобные рекомендации во многих случаях расцениваются как совет: «Копируйте всё». Но, если информации очень много, то это может оказаться слишком дорогим удовольствием, поэтому при выполнении резервного копирования стоит проявить изворотливость. Например, данные, двоичные журналы и конфигурационные файлы операционной системы и сервера можно копировать по отдельности.

Инкрементное резервное копирование

При наличии большого объема данных общепринятой стратегией является регулярное инкрементное копирование. Поделимся некоторыми идеями по этому поводу.

- Делайте резервную копию двоичных журналов. Это самый простой, наиболее распространенный и в общем случае наилучший способ создания инкрементных резервных копий.
- Не копируйте таблицы, которые не изменялись. Некоторые подсистемы хранения, например MyISAM, записывают время последней модификации каждой таблицы. Узнать это время можно, заглянув в файл на диске или воспользовавшись командой `SHOW TABLE STATUS`. При использовании InnoDB можно написать триггер, который поможет отследить время последнего изменения, занося его в специальную маленькую таблицу. Но это нужно делать только для таблиц, которые изменяются редко, чтобы накладные расходы были минимальны. Специальный сценарий резервного копирования сможет легко определить, какие таблицы были изменены.
- Справочные таблицы, например содержащие названия месяцев на разных языках или сокращенные названия государств или регионов, имеет смысл поместить в отдельную базу данных, которая не копируется каждый раз.
- Не копируйте строки, которые не изменялись. Если записи в таблицу только добавляются (командой `INSERT`), как, например, в таблицу, где хранится протокол посещения страниц веб-сайта, то можно включить в нее столбец типа `TIMESTAMP` и включать в резервную ко-

пию только строки, добавленные с момента последнего копирования. Можно также применить подсистему хранения Merge, что позволит хранить старые данные в статических таблицах.

- Некоторую информацию не копируйте вовсе. Иногда это вполне оправдано – например, если имеется хранилище, которое создается из других данных и, строго говоря, является избыточным, то можно включать в копию лишь те элементы, которые нужны для воссоздания хранилища, а его само не копировать. Эта идея может оказаться здравой, даже если воссоздавать хранилище из исходных данных очень долго. Отказ от копирования всех данных может со временем принести многократно большую экономию, чем удалось бы получить при наличии полной копии. Кроме того, можно не копировать некоторые временные значения, например, таблицы, в которых хранятся данные о веб-сеансах.
- Включайте в резервную копию только изменения в двоичном журнале. Чтобы вычислить разницу между двоичными журналами, можно воспользоваться программой *rdiff* и помещать только изменения с момента снятия последней копии (но при этом периодически делайте полные резервные копии). Еще один полезный инструмент, которым мы часто пользуемся, – программа *rdiffbackup*, которая объединяет функциональность *rdiff* и *rsync* для формирования законченного решения по резервному копированию. Или можете просто выполнять после снятия каждой копии команду `FLUSH LOGS`, чтобы начать новый журнал; тогда вычислять двоичные дельты вообще не понадобится.
- Включайте в резервную копию только изменения в файлах данных. Этот совет аналогичен предыдущему. Для данной цели в UNIX применяются все те же программы *rdiff* и *rdiffbackup*. Такая стратегия особенно полезна при работе с очень большими базами, которые изменяются несильно. Предположим, что из терабайта данных ежедневно изменяется только 50 Гбайт. Тогда имеет смысл копировать каждый день только двоичные дельты, а изредка делать полную копию. Выигрыш состоит в том, что применить двоичные дельты к полной копии последовательной операцией дискового чтения/записи гораздо быстрее, чем накатывать двоичный журнал. Впрочем, собственно вычисление двоичной дельты может потребовать больше времени, чем снятие полной копии.

Недостаток инкрементного резервного копирования – повышенная сложность восстановления. Если вы вынуждены проводить восстановление в условиях стресса, то оцените, насколько проще вернуть данные из единственной копии по сравнению с последовательным накатыванием инкрементных копий. Так что, если имеется возможность снимать полные резервные копии, то из соображений простоты мы рекомендуем ей воспользоваться.

В любом случае время от времени выполнять полное копирование необходимо – мы советуем делать это раз в неделю. Вряд ли стоит рассчитывать на то, что можно будет восстановиться, имея только инкрементные копии за год. Даже неделя – это трудоемко и рискованно.

Подсистемы хранения и согласованность

Наличие в MySQL различных подсистем хранения может существенно осложнить процедуру резервного копирования. Проблема заключается в том, как получить согласованную копию базы данных при наличии в ней таблиц произвольных типов.

На самом деле существуют два вида согласованности: *согласованность данных* и *согласованность файлов*.

Согласованность данных

При снятии резервной копии необходимо гарантировать, что данные в копии согласованы по времени. Например, в базе данных сайта электронной торговли счета-фактуры и платежи должны быть согласованы друг с другом. Восстановление платежа без соответствующего счета или наоборот – прямая дорога к неприятностям!

При оперативном резервном копировании (без остановки сервера) необходимо получить согласованные дубликаты всех взаимосвязанных таблиц. Это означает, что нельзя просто блокировать и копировать таблицы по одной, то есть процедура резервного копирования должна знать о структуре базы больше, чем хотелось бы. Если используется нетранзакционная подсистема хранения, то не остается другого выбора, как выполнить команду `LOCK TABLES` для всех таблиц, которые должны копироваться вместе, и освободить блокировки только после того, как процедура будет полностью завершена.

Встроенная в InnoDB технология MVCC способна помочь в этом отношении. Вы можете начать транзакцию, скопировать группу взаимосвязанных таблиц и потом зафиксировать транзакцию. Не следует одновременно с этим использовать команду `LOCK TABLES`, если вы хотите получить согласованную копию, поскольку она неявно фиксирует транзакцию, – детали см. в руководстве по MySQL. Если установлен уровень изоляции `REPEATABLE READ`, то этот метод позволит получить идеально согласованный по времени снимок данных, не блокируя при этом работу сервера на время снятия копии.

Однако такой подход не защитит от неудачно спроектированной логики приложения. Предположим, что интернет-магазин вставляет запись о платеже, фиксирует транзакцию, а затем вставляет счет-фактуру уже в другой транзакции. Процедура резервного копирования может начаться между двумя этими операциями, и тогда платеж войдет в копию, а счет-фактура – нет. Поэтому к использованию транзакций сле-

дует подходить со всей ответственностью, объединяя взаимосвязанные операции вместе.

Получить согласованную логическую копию таблиц InnoDB позволяет также программа *mysqldump*, которая поддерживает флаг *--single-transaction*, делающий в точности то, что было описано выше. Однако при этом могут возникать очень длинные транзакции, что при некоторых характеристиках рабочей нагрузки приводит к неприемлемо высоким издержкам.

Помочь в резервном копировании групп взаимосвязанных таблиц могут инструменты, поддерживающие «групповое резервное копирование», например ZRM (будет рассмотрен ниже), или *mkparallel-dump*, входящий в комплект Maatkit.

Согласованность файлов

Не менее важна внутренняя согласованность каждого файла. Например, в резервную копию не должен попасть файл, состояние которого отражает частичное выполнение большой команды UPDATE. Кроме того, необходимо, чтобы все скопированные файлы были согласованы друг с другом. Если внутренняя согласованность нарушена, то при восстановлении вас поджидают неприятные сюрпризы (скорее всего, данные будут повреждены). А если взаимосвязанные объекты копируются в разные моменты времени, то они не будут согласованы между собой. Примерами могут служить файлы MyISAM с расширениями *.MYD* и *.MYI*.

В случае нетранзакционной подсистемы хранения, в частности MyISAM, единственный вариант – заблокировать и сбросить таблицы. Иными словами, нужно выполнить команды LOCK TABLES и FLUSH TABLES, чтобы сервер сохранил на диск накопившиеся в памяти изменения, или команду FLUSH TABLES WITH READ LOCK. По завершении сброса можно безопасно копировать физические файлы, в которых хранятся таблицы MyISAM.

В случае InnoDB гарантировать согласованность файлов на диске несколько сложнее. Даже после команды FLUSH TABLES WITH READ LOCK InnoDB не прекращает работу в фоновом режиме: потоки буфера вставки, журнала и записи продолжают сохранять изменения в журнал и файлы табличного пространства. Эти потоки задуманы асинхронными – именно выполнение работы в фоновых режимах позволяет InnoDB достигать высокого уровня конкуренции, – поэтому от команды LOCK TABLES они не зависят. Следовательно, нужно гарантировать не только внутреннюю согласованность каждого файла, но и то, что файлы журналов и табличного пространства копируются в один и тот же момент времени. Если резервная копия снимается, когда некоторый поток изменяет файл, или файлы журналов копируются не одновременно с файлами табличного пространства, то после восстановления данные могут оказаться поврежденными. Избежать такого развития событий можно двумя способами.

- Дождаться, пока потоки вытеснения и объединения с буфером вставки завершат работу. Можно следить за тем, что выводит команда `SHOW INNODB STATUS`, и начинать копирование, когда не останется «грязных» или ожидающих операций записи буферов. Но на это может уйти много времени, а кроме того, приходится гадать на кофейной гуще, и все равно из-за наличия фоновых потоков безопасность не гарантируется. Поэтому мы не рекомендуем такой подход.
- Делать согласованный снимок файлов данных и журналов с помощью такой системы, как LVM. *Обязательно* следует снимать файлы данных и журналов согласованно относительно друг друга; делать снимки по отдельности бессмысленно. Ниже в этой главе мы еще будем обсуждать снимки LVM.

После того как файлы куда-то скопированы, можно снять блокировки и продолжить эксплуатацию сервера MySQL в обычном режиме.

Репликация

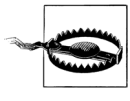
Расхожая мудрость гласит, что механизм репликации в MySQL идеален для резервного копирования. У использования репликации в качестве составной части общей стратегии резервного копирования есть свои достоинства, но это отнюдь не конец и начало всего, как некоторые пытаются представить.

Основное преимущество резервного копирования на подчиненном сервере состоит в том, что не нужно прерывать работу и дополнительно нагружать главный. Это само по себе основательная причина для организации подчиненного сервера, даже если он не нужен с целью балансирования нагрузки или обеспечения высокой доступности. Если бюджет ограничен, то всегда можно использовать сервер резервного копирования и в других целях, например, для генерации отчетов, при условии, что вы не будет производить на нем никаких операций записи, изменяющих резервируемые данные. Подчиненный сервер необязательно должен быть целиком выделен только для копирования; важно лишь, чтобы он успевал догнать главный к моменту снятия следующей копии, если другие занятия не позволяют ему реплицировать информацию без задержек постоянно.

Делая резервные копии на подчиненном сервере, не забывайте сохранять всю информацию о процедуре репликации, например позицию подчиненного сервера относительно главного. Это полезно для клонирования новых подчиненных серверов, для повторного применения двоичных журналов к главному серверу с целью восстановления на конкретный момент времени, для повышения подчиненного сервера до уровня главного и для многих других целей. Кроме того, убедитесь, что в момент останова подчиненного сервера нет открытых временных таблиц, поскольку это может помешать возобновлению репликации. Под-

робнее об этом рассказано в разделе «Отсутствующие временные таблицы» на стр. 488.

Для восстановления после некоторых сбоев бывает очень полезна намеренная задержка репликации. Предположим, что вы производите репликацию с паузой в один час. Если на главном сервере была выполнена нежелательная команда, то у вас есть час, чтобы это заметить и остановить подчиненный сервер до того, как он воспроизведет это событие из своего журнала ретрансляции. Затем можно назначить подчиненный сервер главным и выполнить несколько событий из журнала, обойдя ненужные команды. Это может оказаться намного быстрее, чем техника восстановления на конкретный момент времени, которую мы обсудим ниже. Сценарий *mk-slave-delay* из комплекта Maatkit поспособствует в решении этой задачи.



Информация на подчиненном и главном сервере может различаться. Часто думают, что подчиненные сервера – точные копии главных, но наш опыт показывает, что расхождения в данных – обычное дело, и у MySQL нет средств для обнаружения этой проблемы. Резервное копирование некорректной или поврежденной информации на подчиненном сервере не приведет ни к чему хорошему. О том, как убедиться в идентичности данных на главном и подчиненном сервере, см. раздел «Как узнать, согласованы ли подчиненные серверы с главным» главы 8 на стр. 471. В той же главе приведены рекомендации, как предотвратить расхождение подчиненного и главного сервера.

Наличие реплицированной копии может защитить от таких проблем, как расплавление диска на главном сервере, но гарантий никаких нет. Репликация – это *не* резервное копирование.

Резервное копирование двоичных журналов

Двоичные журналы сервера – одна из важнейших вещей, которые следует включать в резервную копию. Они совершенно необходимы для восстановления на конкретный момент времени, а поскольку их размер обычно меньше, чем сами данные, то копировать их можно чаще. Если существует резервная копия информации, снятая в какой-то момент времени, и все двоичные журналы, накопившиеся с этого момента, то эти журналы можно воспроизвести, то есть «накатить» изменения, произведенные с момента последней полной резервной копии.

В MySQL двоичный журнал также используется для репликации. Это означает, что стратегии резервного копирования и восстановления тесно связаны с конфигурацией репликации.

Двоичные журналы – это «нечто особенное». Если вы потеряли данные, то утратить еще и журналы было бы совсем некстати. Чтобы уменьшить риск потери журналов, их можно хранить на отдельном томе. Это нор-

мально, даже если вы хотите делать снимки двоичных журналов средствами LVM. Для пущей безопасности журналы можно разместить на SAN-хранилище или реплицировать на другое устройство с помощью DRBD. Подробнее об этом см. в главе 9.

Мы рекомендуем копировать журналы почаще. Если потеря данных более чем за 30 минут недопустима, то копируйте их, по крайней мере, раз в 30 минут. Можно также использовать подчиненный сервер репликации, работающий в режиме только чтения, задав флаг *--log_slave_updates*. Позиции в журналах подчиненного и главного сервера, конечно, не будут совпадать, но обычно найти нужную для восстановления точку не составляет труда.

Ниже приведена рекомендуемая конфигурация для записи в двоичный журнал:

```
log_bin           = mysql-bin
sync_binlog       = 1
innodb_support_xa = 1 # MySQL 5.0 и позже
innodb_safe_binlog    # Только MySQL 4.1, грубый эквивалент innodb_support_xa
```

Есть и еще несколько конфигурационных параметров, относящихся к двоичному журналу, например настройки для ограничения размера одного журнала. Дополнительную информацию по этому поводу см. в руководстве по MySQL.

Формат двоичного журнала

Двоичный журнал содержит последовательность событий. У каждого события имеется заголовок фиксированной длины, в котором хранится разнообразная информация, в частности, текущая временная метка и имя подразумеваемой по умолчанию базы данных. Для просмотра содержимого двоичного журнала можно воспользоваться инструментом *mysqlbinlog*, который распечатывает часть сведений из заголовка. Ниже приведен пример вывода:

```
1 # at 277
2 #071030 10:47:21 server id 3 end_log_pos 369 Query thread_id=13
  exec_time=0 error_code=0
3 SET TIMESTAMP=1193755641/*!*/;
4 insert into test(a) values(2)/*!*/;
```

В строке 1 указано смещение в байтах от начала файла журнала (в данном случае, 277).

В строке 2 приведены следующая информация:

- Дата и время события, MySQL использует их для генерации команды `SET TIMESTAMP`.
- Идентификатор исходного сервера, который необходим для предотвращения зацикливания репликации и других проблем.

- Величина `end_log_pos`, то есть смещение начала следующего события в байтах. Для большинства событий в транзакции из нескольких команд это значение некорректно. Во время выполнения транзакции MySQL копирует события в буфер на главном сервере, но позиция следующего события в журнале в этот момент неизвестна.
- Тип события. В примере выше это `Query`, но существует много других типов.
- Идентификатор потока, обработавшего событие на исходном сервере; важно для аудита и для выполнения функции `CONNECTION_ID()`.
- Величина `exec_time`, истинное назначение которой неясно даже отдельным разработчикам MySQL, которым мы задавали вопрос. Обычно здесь хранится время, затраченное на выполнение команды, но при некоторых условиях демонстрируются какие-то странные показатели. Например, вы увидите очень большое значение в журнале ретрансляции на подчиненном сервере, если его поток ввода/вывода далеко отстал от главного сервера, пусть даже на главном сервере команды были выполнены очень быстро. Мы советуем не полагаться на эту величину.
- Код ошибки, возникшей при обработке события на исходном сервере. Если при воспроизведении на подчиненном сервере возникнет иная ошибка, репликация на всякий случай прекратится.

В остальных строках печатаются SQL-команды, необходимые для воспроизведения события. Здесь же вы найдете пользовательские переменные и прочие специальные установки, например временную метку на момент начала обработки команды.

При использовании построчной репликации, появившейся в версии MySQL 5.1, событие не содержит текста SQL-команд, а является «образом» модификаций, произведенных данной командой в таблице. Эти события не предназначены для чтения человеком.

Безопасное удаление старых двоичных журналов

Необходимо продумать стратегию удаления двоичных журналов, чтобы MySQL не заполнила ими весь диск. До какого размера вырастет журнал, зависит от его формата и от рабочей нагрузки (при построчной репликации, появившейся в версии MySQL 5.1, размер одной записи больше). Мы рекомендуем по возможности хранить журналы до тех пор, пока они не станут бесполезны. Двоичные журналы могут пригодиться для настройки подчиненных серверов репликации, для анализа рабочей нагрузки, для аудита и для восстановления на конкретный момент времени с помощью полной резервной копии. Учитывайте все это, когда будете решать, насколько долго хранить журналы.

Обычно используют конфигурационную переменную `expire_logs_days`, которая говорит MySQL, по истечении какого времени следует удалять

журналы. До выхода версии MySQL 4.1 этой переменной не существовало, поэтому журналы приходилось удалять вручную. Еще с тех времен осталась рекомендация уничтожать старые двоичные журналы такой командой в списке заданий *cron*:

```
0 0 * * * /usr/bin/find /var/log/mysql -mtime +N -name "mysql-bin.[0-9]*" | xargs rm
```

Хотя до версии MySQL 4.1 это был единственный способ удалить двоичные журналы, в более поздних версиях так делать ни в коем случае не следует! Если стереть журналы командой *rm*, то индексный файл *mysql-bin.index* рассинхронизируется с файлами на диске, и некоторые команды, например `SHOW MASTER LOGS`, начнут работать неправильно, но что самое опасное – пользователю не будет продемонстрировано никаких сообщений об ошибках. Ручная корректировка файла *mysql-bin.index* не поможет. Поэтому вставляйте в таблицу *cron* такую команду:

```
0 0 * * * /usr/bin/mysql -e "PURGE MASTER LOGS BEFORE CURRENT_DATE - INTERVAL N DAY"
```

Значение параметра `expire_logs_days` считывается на этапе запуска сервера или в момент, когда MySQL ротирует двоичный журнал, поэтому если он никогда не заполняется до конца и не ротируется, то сервер и не будет удалять старые записи. Решение о том, какие файлы уничтожить, сервер принимает исходя из даты модификации файла, а не из его содержимого.

Резервное копирование данных

Как обычно, существуют хорошие и плохие способы резервного копирования, причем самые очевидные из них не обязательно лучшие. Хитрость состоит в том, чтобы по максимуму задействовать пропускную способность сети, диска и процессора и постараться закончить копирование как можно быстрее. Для того чтобы найти «золотую середину», придется поэкспериментировать.

Дать конкретный совет в этом случае трудно, поэтому мы расскажем о нескольких общих приемах.

Снятие логической резервной копии

Прежде всего нужно отчетливо понимать, что существует две разновидности логических копий: SQL-дампы и файлы с разделителями.

SQL-дампы

SQL-дамп лучше знаком пользователям, так как это именно то, что по умолчанию создает программа *mysqldump*. Например, при выгрузке дампа небольшой таблицы с параметрами по умолчанию получится такой результат (мы кое-что опустили):

```
$ mysqldump test t1
-- [Комментарий: номер версии и информация о сервере]
```

```
/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
-- [Дополнительные комментарии, зависящие от версии, с
-- параметрами, необходимыми для восстановления]

--
-- Структура таблицы `t1`
--

 DROP TABLE IF EXISTS `t1`;
 CREATE TABLE `t1` (
   `a` int(11) NOT NULL,
   PRIMARY KEY (`a`)
 ) ENGINE=MyISAM DEFAULT CHARSET=latin1;

--
-- Дамп данных таблицы `t1`
--

 LOCK TABLES `t1` WRITE;
 /*!40000 ALTER TABLE `t1` DISABLE KEYS */;
 INSERT INTO `t1` VALUES (1);
 /*!40000 ALTER TABLE `t1` ENABLE KEYS */;
 UNLOCK TABLES;
 /*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

 /*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
-- [Другие параметры восстановления]
```

Файл дампа содержит как описание таблицы, так и данные, причем то и другое представлено в виде SQL-команд. Файл начинается с комментариев, в которых устанавливаются значения различных параметров MySQL. Они включены для того, чтобы помочь вам при восстановлении, а также ради совместимости и корректности. Затем идет описание таблицы и данные. В самом конце сценарий воссоздает параметры, которые были изменены в начале дампа.

Созданный дамп может быть выполнен для восстановления данных. Это удобно, но подразумеваемые по умолчанию параметры *mysqldump* не годятся для снятия очень больших резервных копий (ниже мы рассмотрим параметры *mysqldump* подробнее).

Программа *mysqldump* не единственный инструмент, позволяющий создать логическую резервную копию. Это можно сделать, к примеру, с помощью phpMyAdmin. Но мы хотим отметить не столько проблемы, свойственные тому или иному инструменту, а недостатки монолитной логической копии как таковой. Перечислим их:

Схема и данные хранятся вместе

Хотя это удобно, когда нужно восстановить все из одного файла, но усложняет задачу, если требуется восстановить только одну таблицу

или только данные. Эту трудность можно устранить, произведя выгрузку дампа дважды – один раз для данных, другой для схемы, но от следующих проблем все равно никуда не деться.

Огромные SQL-команды

Разбор и выполнение всех SQL-команд в дампе требует от сервера значительных усилий. Поэтому загрузка данных происходит довольно медленно.

Один гигантский файл

Большинство текстовых редакторов не умеют обрабатывать файлы с очень длинными строками. Хотя в некоторых случаях для извлечения нужных данных можно воспользоваться потоковыми редакторами, например *sed* или *grep*, лучше, чтобы файлы были поменьше.

Создание логической копии обходится дорого

Существуют более эффективные способы извлечь из MySQL данные помимо передачи их по протоколу взаимодействия между клиентом и сервером в виде результирующих наборов.

Все эти ограничения означают, что SQL-дамп становится непригодным для работы по мере роста таблиц. Однако существует и другая возможность: экспортировать информацию в файлы с разделителями.

Резервные копии в виде файлов с разделителями

Для создания логической копии данных в формате файла с разделителями можно воспользоваться командой `SELECT INTO OUTFILE SQL`. Ту же команду выполнит программа *mysqldump*, запущенная с флагом *--tab*. Файлы с разделителями содержат значения, представленные в кодировке ASCII, без SQL-команд, комментариев и имен столбцов. Следующая команда выводит результат в формате CSV (с разделителями-запятыми) – общепринятом для представления табличных данных.

```
mysql> SELECT * INTO OUTFILE '/tmp/t1.txt'
    -> FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    -> LINES TERMINATED BY '\n'
    -> FROM test.t1;
```

Результирующий файл компактнее и им проще манипулировать с помощью утилит командной строки, чем SQL-дампом. Но самое главное достоинство кроется не в этом, а в скорости резервного копирования и последующего возврата данных. Для загрузки информации назад в таблицу предназначена команда `LOAD DATA INFILE`, которой следует указать те же параметры, что при выгрузке:

```
mysql> LOAD DATA INFILE '/tmp/t1.txt'
    -> INTO TABLE test.t1
    -> FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    -> LINES TERMINATED BY '\n';
```

Ниже приведены результаты неофициального теста для демонстрации различий в скорости создания SQL-дампа и файла с разделителями. Мы воспользовались слегка адаптированными данными из промышленной базы. Выгружаемая таблица имеет следующую структуру:

```
CREATE TABLE load_test (
   col1 date NOT NULL,
   col2 int NOT NULL,
   col3 smallint unsigned NOT NULL,
   col4 mediumint NOT NULL,
   col5 mediumint NOT NULL,
   col6 mediumint NOT NULL,
   col7 decimal(3,1) default NULL,
   col8 varchar(10) NOT NULL default '',
   col9 int NOT NULL,
   PRIMARY KEY (col1,col2)
) ENGINE
```

Таблица содержит 15 миллионов строк и занимает примерно 700 Мбайт на диске. В табл. 11.1 приведены результаты сравнения двух методов резервного копирования и восстановления. Как видите, разница в скорости восстановления весьма значительна.

Таблица 11.1. Время резервного копирования и восстановления для SQL-дампа и файла с разделителями

Метод	Размер файла, Мбайт	Время выгрузки, сек	Время восстановления, сек
SQL-дамп	727	102	600
Файл с разделителями	669	86	301

Однако у метода `SELECT INTO OUTFILE` есть ограничения:

- Копировать данные можно только в файл, находящийся на том же компьютере, где работает MySQL. Но можно написать собственный вариант команды `SELECT INTO OUTFILE` в виде программы, которая читает результат `SELECT` и пишет его на диск; мы знаем людей, которые так и поступали.
- У MySQL должно быть разрешение на запись в каталог, где создается файл, поскольку именно сервер MySQL, а не пользователь, запустивший команду, пишет в этот файл.
- Из соображений безопасности MySQL отказывается перезаписывать существующий файл вне зависимости от имеющихся разрешений.
- Невозможно писать дамп в сжатый файл.

Параллельная выгрузка и загрузка

Зачастую гораздо быстрее выгружать и загружать данные параллельно на машине с несколькими процессорами. Под словом «параллельно» мы здесь понимаем, что производится выгрузка либо загрузка нескольких таблиц сразу, а не синхронную работу нескольких программ над одной и той же таблицей. Когда два приложения одновременно загружают данные в одну таблицу, ничего хорошего, как правило, не получается.

Для параллельной выгрузки или загрузки не нужны какие-то особые инструменты; все можно сделать вручную, запустив несколько экземпляров программы резервного копирования. Тем не менее, существуют специальные приложения и сценарии, предназначенные специально для этой цели, например *mk-parallel-dump* из комплекта Maatkit и *mysqlpdump* (*http://www.fr3nd.net/projects/mysqlpdump/*). На момент работы над этой книгой эти инструменты были еще сравнительно новыми. Однако эталонные тесты показывают, что *mk-parallel-dump* выгружает данные в несколько раз быстрее, чем стандартная *mysqldump*.

В версии MySQL 5.1 программа *mysqlimport* поддерживает одновременный импорт несколькими потоками. Вариант *mysqlimport* из версии 5.1 может работать и в более ранних версиях MySQL.

Но если задать слишком высокую степень параллелизма, то параллельная выгрузка и загрузка данных может занять даже больше времени. Кроме того, иногда это приводит к фрагментации информации, что негативно сказывается на производительности.

Снимки файловой системы

Снимок файловой системы – прекрасный способ сделать оперативную резервную копию. Файловые системы, поддерживающие снимки, способны мгновенно создать согласованный образ содержимого, который затем можно использовать в качестве резервной копии. К числу таких файловых систем и устройств относятся FreeBSD, ZFS, GNU/Linux Logical Volume Manager (LVM), а также многие SAN-системы и файловые хранилища, например NetApp.

Не путайте снимок с резервной копией. Снимок лишь позволяет уменьшить время удержания блокировок; после того как блокировки сняты, файлы необходимо поместить в резервную копию. На самом деле, можно даже делать снимки таблиц InnoDB без захвата блокировок. Мы покажем два способа использования LVM для резервного копирования базы, состоящей только из таблиц InnoDB: с минимальной блокировкой и вообще без блокировки.

Ленц Гриммер (Lenz Grimmer) написал Perl-сценарий *mylvmbackup* для создания резервных копий MySQL с помощью LVM. Дополнительную информацию см. в разделе «Инструменты резервного копирования» на стр. 629.

Как работают снимки LVM

В LVM для создания снимка применяется технология копирования при записи, а это означает, что *логическая* копия всего тома снимается мгновенно. В какой-то мере это напоминает технологию MVCC в СУБД с тем отличием, что хранится лишь одна версия устаревшей информации.

Обратите внимание, что мы не говорим о *физической* копии. Может казаться, что логическая копия содержит данные, которые были на томе в момент снимка, но на самом деле изначально в ней нет вообще никаких данных. Вместо того чтобы копировать информацию в снимок, LVM лишь запоминает момент времени, в который снимок был создан. Позже, когда вы будете запрашивать данные из снимка, LVM начнет читать информацию из ее исходного местоположения. Поэтому-то операция первоначального копирования производится мгновенно вне зависимости от размера тома, для которого создается снимок.

Когда на исходном томе что-то изменяется, LVM копирует прежнее содержимое блоков в область, зарезервированную для снимка, и лишь затем вносит модификации. LVM не хранит несколько «старых версий» данных, поэтому последующая запись в уже измененные блоки не требует никакой дополнительной работы. Иными словами, только при первой записи блок копируется в зарезервированную область.

Когда кто-то запрашивает эти блоки из снимка, LVM читает данные из копий в зарезервированной области, а не с рабочего тома. Это позволяет вам неизменно видеть в снимке одну и ту же информацию, хотя блокировки тома ни в какой момент не производилось. Эта схема изображена на рис. 11.1.

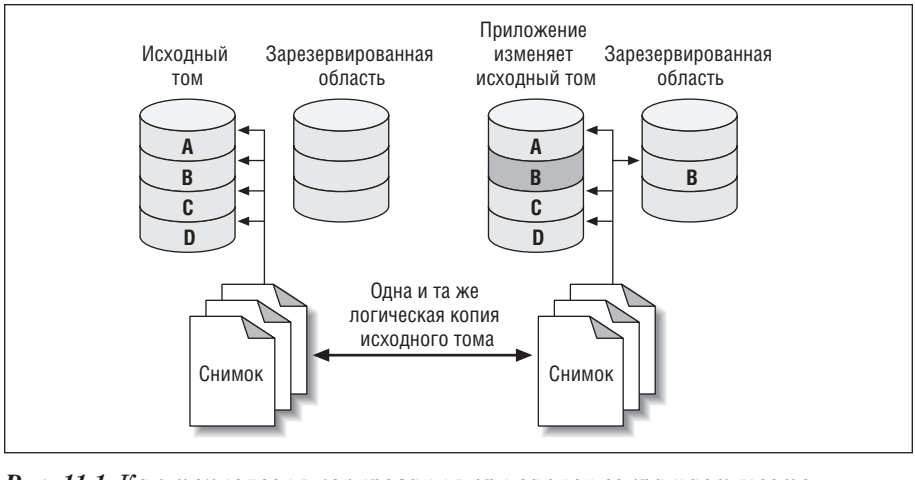


Рис. 11.1. *Как технология копирования при записи сокращает место, необходимое для хранения снимка тома*

Снимку соответствует новое логическое устройство в каталоге */dev*, которое можно смонтировать, как и любое другое.

Теоретически можно сделать снимок огромного тома, затратив на это очень немного физического пространства. Однако необходимо заранее оценить, сколько блоков может обновиться, пока снимок открыт, и отвести для них достаточно места. Если для копирования при записи не хватит дискового пространства, то снимок переполнится и устройство станет недоступно. Выглядит это как отсоединение внешнего диска: любая операция чтения с устройства, в том числе резервное копирование, завершается с ошибкой ввода/вывода.

Необходимые условия и конфигурация

Создание снимка – почти тривиальная задача, но нужно сконфигурировать систему так, чтобы получать согласованный дубликат *всех* файлов, которые планируется включить в резервную копию, в один и тот же момент времени. Прежде всего, убедитесь, что система удовлетворяет следующим условиям:

- Все файлы InnoDB (файлы табличного пространства и журналы транзакций InnoDB) должны находиться на одном физическом томе (разделе). Необходима абсолютная согласованность во времени, а LVM не умеет одномоментно делать согласованные снимки нескольких томов. Это ограничение LVM; некоторые другие системы не испытывают таких трудностей.
- Если необходимо включить в резервную копию также определения таблиц, то на том же томе должен находиться каталог данных MySQL. Если для резервного копирования определений таблиц применяется какой-то другой метод, например хранение схемы в системе управления версиями, то об этом можно не беспокоиться.
- В группе томов должно быть достаточно свободного места для создания снимка, сколько именно – зависит от рабочей нагрузки. Поэтому при конфигурировании системы оставьте нераспределенное пространство для последующих снимков.

В LVM существует понятие *группы томов*, в которую входит один или несколько логических томов. Посмотреть, какие имеются группы томов, позволяет следующая команда:

```
# vgs
VG #PV #LV #SN Attr   VSize   VFree
vg   1   4   0 wz--n- 534.18G 249.18G
```

Здесь мы видим, что существует одна группа, состоящая из четырех логических томов, которые находятся на одном физическом томе и занимают примерно 250 Гбайт. При необходимости можно получить более подробную информацию с помощью команды *vgdisplay*. Теперь посмотрим на сами логические тома:

```
# lvs
LV    VG Attr   LSize   Origin Snap% Move Log Copy%
home  vg -wi-ao 40.00G
mysql vg -wi-ao 225.00G
tmp   vg -wi-ao 10.00G
var   vg -wi-ao 10.00G
```

Мы видим, что размер тома `mysql`составляет 225 Гбайт. Данное устройство называется `/dev/vg/mysql`. Это всего лишь имя, хотя выглядит оно, как путь в файловой системе. Еще сильнее запутывает ситуацию тот факт, что существует символическая ссылка с этого имени на узел реального устройства */dev/mapper/vg-mysql*, которую показывают команды *ls* и *mount*:

```
# ls -l /dev/vg/mysql
lrwxrwxrwx 1 root root 20 Sep 19 13:08 /dev/vg/mysql -> /dev/mapper/vg-mysql
# mount | grep mysql
/dev/mapper/vg-mysql on /var/lib/mysql type reiserfs (rw,noatime,notail)
```

Вооружившись этой информацией, вы можете приступить к созданию снимка файловой системы.

Создание, монтирование и удаление снимка LVM

Создать снимок можно одной командой. Нужно лишь решить, куда его поместить и сколько места отвести под копирование при записи. Не бойтесь указать больше места, чем может реально потребоваться. LVM не займет этот объем немедленно, а просто зарезервирует для использования в будущем, поэтому резервирование очень большого пространства не принесет никакого вреда, если только вам одновременно не нужно оставлять место для других снимков.

Давайте попрактикуемся в создании снимка. Отведем для копирования при записи 16 Гбайт и назовем снимок `backup_mysql`:

```
# lvcreate --size 16G --snapshot --name backup_mysql /dev/vg/mysql
Logical volume “backup_mysql” created
```

Мы сознательно назвали том `backup_mysql`, а не `mysql_backup`, чтобы не возникало неоднозначности при автоматическом завершении команды по нажатию клавиши Tab. Это позволяет подстраховаться от случайного удаления тома `mysql` при автодопонении имени файла нажатием кнопки Tab. Подобные мелочи помогают избежать катастрофы. По крайней мере один из авторов этой книги попадал в неприятную ситуацию, поспешно нажав Tab при работе со снимками LVM.

Теперь посмотрим на состояние только что созданных томов:

```
# lvs
LV          VG Attr    LSize   Origin Snap% Move Log Copy%
backup_mysql vg swi-a-  16.00G  mysql   0.01
```

```
home         vg -wi-ao  40.00G
mysql        vg owi-ao 225.00G
tmp          vg -wi-ao  10.00G
var          vg -wi-ao  10.00G
```

Обратите внимание, что атрибуты снимка и исходного устройства различаются, и что показана кое-какая дополнительная информация: источник снимка и сведения о том, какая часть из отведенных 16 Гбайт в данный момент уже израсходована на копирование при записи. Мы настоятельно рекомендуем следить за состоянием во время снятия резервной копии, чтобы не пропустить момент, когда снимок близок к заполнению и грозит отказом. Вести мониторинг состояния устройств позволяют такие системы, как Nagios. А вот как это можно делать с помощью команды watch:

```
# watch 'lvs | grep backup'
```

Как видно из представленной выше выдачи команды *mount*, на томе `mysql` смонтирована файловая система ReiserFS. Следовательно, она же будет и на томе снимка и ее можно монтировать и использовать, как и любую другую файловую систему:

```
# mkdir /tmp/backup
# mount /dev/mapper/vg-backup_mysql /tmp/backup
# ls -l /tmp/backup/mysql
total 5336
-rw-r----- 1 mysql mysql     0 Nov 17 2006  columns_priv.MYD
-rw-r----- 1 mysql mysql  1024 Mar 24 2007  columns_priv.MYI
-rw-r----- 1 mysql mysql  8820 Mar 24 2007  columns_priv.frm
-rw-r----- 1 mysql mysql 10512 Jul 12 10:26 db.MYD
-rw-r----- 1 mysql mysql  4096 Jul 12 10:29 db.MYI
-rw-r----- 1 mysql mysql  9494 Mar 24 2007  db.frm
... опущено ...
```

Поскольку мы лишь практикуемся, размонтируем и удалим снимок командой *lvremove*:

```
# umount /tmp/backup
# rmdir /tmp/backup
# lvremove --force /dev/vg/backup_mysql
Logical volume "backup_mysql" successfully removed
```

Применение снимков LVM для оперативного резервного копирования

После того как мы научились создавать, монтировать и удалять снимки, воспользуемся ими для снятия резервных копий. Сначала посмотрим, как сделать резервную копию базы данных с таблицами InnoDB, не останавливая сервер MySQL. Подключитесь к серверу и сбросьте таблицы на диск, установив глобальную блокировку чтения, а затем получите координаты в двоичном журнале:

```
mysql> FLUSH TABLES WITH READ LOCK; SHOW MASTER STATUS;
```

Запишите результаты, выданные командой SHOW MASTER STATUS, и оставьте соединение с MySQL открытым, чтобы не освобождать блокировку. Теперь можно сделать снимок LVM и сразу же снять блокировку чтения, выполнив команду UNLOCK TABLES либо закрыв соединение. И, наконец, смонтируйте снимок и поместите файлы в резервную копию. Если все эти действия оформить в виде сценария, то время блокировки можно сократить до нескольких секунд.

Основная проблема при таком подходе заключается в том, что на получение блокировки чтения может уйти достаточно много времени, особенно если выполняется какой-то длинный запрос. Пока соединение ожидает глобальной блокировки чтения, все запросы также блокируются и невозможно заранее сказать, как долго это будет продолжаться.

Снимки файловой системы и InnoDB

Фоновые потоки InnoDB продолжают работать даже после блокировки всех таблиц, поэтому они могут вести запись в файлы, когда делается снимок. Кроме того, поскольку подсистема InnoDB не выполняла корректную последовательность останова, файлы в снимке будут выглядеть так, будто сервер аварийно завершил работу.

Это не проблема, поскольку InnoDB – транзакционная подсистема хранения. В любой момент (в том числе и в момент снятия снимка) любая зафиксированная транзакция находится либо в файлах данных InnoDB, либо в файлах журналов. После того как MySQL будет запущен на восстановленном снимке, InnoDB инициирует свою процедуру восстановления точно так же, как в случае отключения питания сервера. Она найдет в журнале все зафиксированные транзакции, которые еще не попали в файлы данных, и применит их, поэтому потери транзакций не произойдет. Вот почему так важно включать файлы данных и журналов InnoDB в один снимок.

И по той же причине следует тестировать резервные копии сразу после снятия. Запустите экземпляр MySQL, сообщите ему, где находится новая копия, позвольте InnoDB завершить восстановление и проверьте все таблицы. Тогда вы будете уверены, что в резервную копию незаметно для вас не просочились поврежденные данные (файлы могут оказаться испорченными по самым разным причинам). У такого подхода есть и еще один плюс: в будущем воссоздание информации с этой копии пройдет быстрее, так как InnoDB уже завершила процесс восстановления.

Описанные действия можно проделать над снимком еще до того, как он будет помещен в резервную копию, хотя это несколько увеличивает издержки. Но так или иначе обязательно запланируйте его – подробнее об этом ниже.

Применение снимков LVM для резервного копирования InnoDB без блокировок

Техника резервного копирования без блокировок отличается от описанной лишь незначительно. Различие в том, что не выполняется команда FLUSH TABLES WITH READ LOCK. Это означает, что для файлов MyISAM на диске нет никаких гарантий согласованности, но если все таблицы имеют тип InnoDB, это не имеет существенного значения. В системной базе данных mysql все равно имеются кое-какие таблицы типа MyISAM, но в условиях типичной рабочей нагрузки маловероятно, что они будут изменяться в момент снятия снимка.

Если вы полагаете, что системные таблицы в базе mysql все же могут изменяться, то заблокируйте и сбросьте их на диск. К таким таблицам не предъявляются сколько-нибудь длительные запросы, поэтому обычно все происходит очень быстро:

```
mysql> LOCK TABLES mysql.user READ, mysql.db READ, ...;
mysql> FLUSH TABLES mysql.user, mysql.db, ...;
```

Поскольку глобальная блокировка на чтение не ставилась, то ничего полезного от команды SHOW MASTER STATUS не получить. Однако при запуске MySQL на снимке (для проверки целостности копии) вы увидите в журнале примерно такие сообщения:

```
InnoDB: Doing recovery: scanned up to log sequence number 0 40817239
InnoDB: Starting an apply batch of log records to the database...
InnoDB: Progress in percents: 3 4 5 6 ...[omitted]... 97 98 99
InnoDB: Apply batch completed
InnoDB: Last MySQL binlog file position 0 3304937, file name /var/log/mysql/
        mysqlbin.000001
070928 14:08:42 InnoDB: Started; log sequence number 0 40817239
```

InnoDB протоколирует позицию в двоичном журнале, соответствующую точке, до которой он был восстановлен. Именно эту позицию можно использовать при восстановлении на конкретный момент времени.

У описанного подхода к резервному копированию без блокировки с помощью снимков в версии MySQL 5.0 и более поздних есть один дефект. В этих версиях для координации между InnoDB и двоичным журналом применяется технология XA (распределенные транзакции). Если восстановить копию на сервер с идентификатором server_id, отличным от того, на котором снималась копия, то сервер может обнаружить транзакции, подготовленные сервером с идентификатором, не совпадающим с его собственным. В таком случае сервер рискует запутаться, и после восстановления транзакция может «зависнуть» в состоянии PREPARED. Хоть и редко, но такое случается. Поэтому обязательно проверяйте копию, не считайте, что с ней априори все нормально. Может оказаться, что восстановиться с нее не удастся!

Если снимок делался на подчиненном сервере, то процедура восстановления InnoDB напечатает также такие строки:

```
InnoDB: In a MySQL replication slave the last master binlog file
InnoDB: position 0 115, file name mysql-bin.001717
```

В некоторых версиях MySQL эти строчки содержат координаты в двоичном журнале главного (а не подчиненного) сервера той точки, до которой прошло восстановление. Это может оказаться очень полезным при создании резервных копий на подчиненных серверах или для клонирования одних подчиненных серверов из других. Однако в версии MySQL 5.0 и более поздних доверять указанным значениям нельзя.

Планирование резервного копирования с помощью LVM

Резервное копирование с помощью снимков LVM обходится не даром. Чем больше сервер пишет на исходный том, тем больше накладные расходы. Когда сервер модифицирует много блоков, расположенных в случайном порядке, головка диска должна постоянно перемещаться в зарезервированную для копирования область, чтобы записать туда старую версию данных. Чтение из снимка также сопряжено с издержками, так как большую часть информации LVM читает с исходного тома, а из зарезервированной области – только по необходимости. Таким образом, логически последовательное чтение из снимка на самом деле приводит к хаотичному перемещению головки.

Это необходимо учитывать при планировании. По существу, получается, что как при чтении, так и при записи исходный том и снимок функционируют хуже, чем обычно, а, если в зарезервированной области оказалось много блоков, то, вероятно, даже гораздо хуже. В результате может замедлиться как работа сервера MySQL, так и процесса копирования файлов в резервную копию.

Кроме того, важно выделить достаточно места для снимка. Мы рекомендуем следующую тактику.

- Помните, что LVM должен скопировать каждый измененный блок в снимок только один раз. Когда MySQL записывает блок на исходный том, LVM копирует блок в снимок, а затем вносит пометку о скопированном блоке в таблицу исключений. При последующих изменениях того же блока он уже не копируется.
- Если в базе имеются только таблицы типа InnoDB, то примите во внимание, как InnoDB сохраняет информацию. Поскольку любой элемент данных записывается дважды, то по меньшей мере половина всех операций попадает в буфер двойной записи, файлы журналов и сравнительно компактные области на диске. Тем самым одни и те же блоки перезаписываются снова и снова, так что при первой модификации они оказывают влияние на снимок, но потом сохранение туда прекращается.
- Затем оцените, какая часть операций ввода/вывода сопряжена с записью в блоки, которые еще не копировались в снимок, по сравне-

нию с повторной модификацией уже измененных данных. Не бойтесь завысить оценку. Это тот дополнительный объем ввода/вывода, которого следует ожидать от использования снимка (добавьте еще небольшой запас на работу самого LVM).

- С помощью *vmstat* или *iostat* соберите статистику о количестве записываемых блоков в секунду. Дополнительную информацию об этих инструментах см. в главе 7.
- Измерьте (или оцените), сколько времени требуется для копирования резервной копии в другое место; иными словами, как долго нужно будет держать снимок LVM открытым.

Предположим, вы прикинули, что половина операций записи пойдет в область, зарезервированную для копирования при записи, и что сервер сохраняет 10 Мбайт в секунду. Если для копирования снимка на другой сервер требуется один час (3600 секунд), то для снимка понадобится ½ × 10 Мбайт × 3600 = 18 Гбайт. На всякий случай оставьте побольше.

Иногда бывает легко вычислить, сколько данных изменится, пока снимок открыт. Вернемся к примеру, который уже неоднократно встречался ранее. Система поиска по форумам BoardReader хранит на каждом узле примерно 1 Тбайт данных в виде таблиц InnoDB.

Однако мы знаем, что наибольшие расходы обусловлены загрузкой новых данных. Каждый день добавляется примерно 10 Гбайт информации, поэтому для снимка вполне хватит 50 Гбайт. Впрочем, эта оценка не всегда корректна. Как-то раз мы запустили длительную команду `ALTER TABLE`, которая последовательно изменяла каждую секцию, а суммарно было модифицировано гораздо больше, чем 50 Гбайт; пока эта команда работала, мы не могли снять резервную копию.

Другие применения и альтернативы

Снимки можно использовать не только для резервного копирования. Например, они полезны для записи «контрольной точки» непосредственно перед выполнением опасной операции. Некоторые файловые системы, например ZFS, позволяют вернуть данные из снимка на основную файловую систему. Это позволяет без труда откатиться к тому состоянию, которое имело место перед созданием снимка.

Снимки файловой системы – не единственный способ получить мгновенную копию данных. Еще один вариант – это расщепление RAID-массива: если имеется зеркалированный программно реализованный RAID-массив из трех дисков, то можно исключить один диск из зеркала и смонтировать его отдельно. Тогда никаких издержек на копирование при записи не будет, а в случае необходимости очень легко сделать такой «снимок» главной копией.

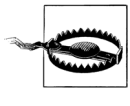
Восстановление из резервной копии

На самом деле, наиболее важно восстановление. В этом разделе мы сосредоточимся на аспектах этого процесса, характерных именно для MySQL, в предположении, что вы знаете, как настраивать прочие части окружения. Регулярно практикуйте «учебные тревоги», чтобы точно знать, как восстанавливать данные, когда случится беда. И обязательно проверяйте резервные копии.

Порядок восстановления зависит от того, как снималась резервная копия. Вам может понадобиться выполнить все или некоторые из перечисленных ниже шагов:

- Остановить сервер MySQL
- Записать куда-нибудь конфигурацию сервера и права доступа к файлам
- Скопировать данные с резервной копии в каталог данных MySQL
- Внести изменения в конфигурационные файлы
- Изменить права доступа к файлам
- Запустить сервер в режиме ограниченного доступа и подождать, пока он перейдет в состояние готовности
- Загрузить логические файлы резервной копии
- Проверить и воспроизвести двоичные журналы
- Убедиться, что все восстановлено
- Перезапустить сервер в режиме полного доступа

В следующих разделах мы покажем, как выполняется каждый из этих шагов. А позже сделаем некоторые замечания о конкретных методах резервного копирования и соответствующих инструментах.



Если есть шанс, что текущие версии файлов еще понадобятся, *не замещайте их файлами из резервной копии*. Например, если копия не содержит двоичных журналов, а они необходимы для восстановления на конкретный момент времени, то не затирайте текущие журналы устаревшими версиями из копии. При необходимости переименуйте их или скопируйте в какое-то другое место.

Ограничение доступа к MySQL

Во время восстановления часто бывает важно, чтобы к MySQL не мог обращаться никто, кроме процесса восстановления. В сложных системах дать такую гарантию трудно. Для уверенности в том, что сервер будет недоступен приложениям, пока мы все не проверим, мы запускаем его с флагами *--skip-networking* и *--socket=/tmp/mysql_recover.sock*. Это особенно важно для логических копий, которые загружаются по частям.

Восстановление из физических файлов

Процедура возврата данных из физических файлов довольно прямолинейна, иначе говоря, вариантов здесь немного. Хорошо это или плохо, зависит от требований к восстановлению. Обычно все сводится к простому копированию файлов туда, где им надлежит находиться.

Нужно ли останавливать MySQL, определяется подсистемой хранения. Файловые объекты MyISAM обычно не зависят друг от друга, поэтому простого копирования файлов с расширениями *.frm, .MYI* и *.MYD* для каждой таблицы достаточно, даже на работающем сервере. Сервер найдет таблицу, как только к ней поступит запрос или будет выполнена иная команда, для которой необходима данная таблица (например, `SHOW TABLES`). Если во время копирования этих файлов таблица была открыта, то вполне возможны неприятности, поэтому предварительно удалите или переименуйте ее либо заблокируйте командами `LOCK TABLES` и `FLUSH TABLES`.

С InnoDB дело обстоит иначе. Если восстанавливается традиционно сконфигурированная база InnoDB (когда все таблицы хранятся в одном табличном пространстве), то нужно будет остановить MySQL, скопировать или переместить файлы на место, а затем перезапустить сервер. Кроме того, нужно убедиться, что файл журнала транзакций соответствует файлам, содержащим табличное пространство. Если эти файлы не соответствуют друг другу, – например, табличное пространство вы заменили, а про журнал транзакций забыли, – то InnoDB откажется запускаться. Поэтому так важно включать в резервную копию не только файлы данных, но и журнал транзакций.

Если используется недавно появившаяся возможность размещать по одной таблице в каждом файле (режим `innodb_file_per_table`), то InnoDB хранит данные и индексы для каждой таблицы в файле с расширением *.ibd*, представляющем собой нечто вроде комбинации MYI и MYD-файлов в подсистеме MyISAM. Резервное копирование и восстановления отдельных таблиц в этом случае может выполняться простым копированием этих файлов, и делать это можно на работающем сервере, но не так просто, как в случае MyISAM. Отдельные файлы не являются независимыми от InnoDB в целом. В каждом IBD-файле записана информация, говорящая InnoDB о том, как этот файл связан с основным (общим) табличным пространством. При возврате такого файла необходимо попросить InnoDB «импортировать» его.

У этой процедуры много ограничений, о которых можно прочитать в разделе руководства по MySQL, посвященном табличным пространствам с раздельным хранением таблиц. Самое серьезное заключается в том, что восстановить таблицу можно только на тот сервер, с которого она была скопирована. Не то чтобы резервное копирование и восстановление таблиц в этой конфигурации было вообще невозможно, но это несколько сложнее, чем могло бы показаться.

Наличие таких сложностей означает, что восстановление физических файлов способно оказаться очень трудоемким процессом и можно наделать ошибок. Эвристическое правило состоит в том, что чем сложнее и труднее становится процедура восстановления, тем важнее оградить себя от неприятностей, создавая также логические копии. Всегда полезно ее иметь на случай, если что-то пойдет не так, и вы не сможете заставить MySQL использовать физическую копию.

Запуск MySQL после восстановления физических файлов

Перед тем как запускать восстановленный сервер MySQL, нужно сделать несколько вещей.

Первое и самое важное, о чем легче всего забыть, – проверить конфигурацию сервера и убедиться в том, что для восстановленных файлов задан правильный владелец и права доступа. Это нужно делать *до* попытки запустить сервер. Если эти атрибуты хоть чем-то отличаются от требуемых, MySQL может не запуститься. В различных системах они задаются по-разному, поэтому посмотрите в своих записях, как они должны быть установлены. Обычно для файлов и каталогов указывается владелец и группа *mysql*, причем владельцу и группе должны быть разрешены чтение и запись, а всем остальным запрещен всякий доступ.

Советуем также следить за журналом ошибок MySQL в ходе запуска. В UNIX-подобных системах для этого можно выполнить такую команду:

```
$ tail -f /var/log/mysql/mysql.err
```

Точное местоположение журнала ошибок зависит от системы. Начав мониторинг этого файла, запускайте сервер MySQL и наблюдайте за тем, что будет появляться в журнале. Если ошибок не возникнет, значит, сервер восстановился нормально и может работать.

Наблюдение за журналом ошибок особенно важно в последних версиях MySQL. В прежних реализациях сервер просто не запускался, если InnoDB обнаруживала ошибку, но теперь он запускается, однако InnoDB отключается. Даже если кажется, что сервер стартовал без проблем, следует в каждой базе данных выполнить команду `SHOW TABLE STATUS` и снова проверить журнал ошибок.

Восстановление из логической копии

Если восстановление производится из логической резервной копии, а не из физических файлов, то для загрузки данных в таблицы понадобится сам сервер MySQL. Копирования файлов на уровне операционной системы недостаточно.

Прежде чем загружать файл дампа, посмотрите на его размер и подумайте, сколько времени он будет обрабатываться и не надо ли что-то сделать перед началом процедуры, например уведомить пользователей или деактивировать часть приложения. Полезно на этот период отклю-

чить запись в двоичный журнал, если только вы не хотите реплицировать восстанавливаемые данные на подчиненный сервер: загружать огромный дамп серверу и так-то нелегко, а запись в двоичный журнал только увеличивает издержки (возможно, без всякой необходимости). Кроме того, для некоторых подсистем хранения загрузка очень больших файлов требует особых предосторожностей. Например, загружать файл размером 100 Гбайт в таблицу InnoDB одной транзакцией – безумная затея, поскольку в результате образуется огромный сегмент отката. Необходимо разбить файл на порции и фиксировать транзакцию после загрузки каждой из них.

Существует две разновидности восстановления в соответствии с двумя видами логических копий.

Загрузка SQL-файлов

SQL-дамп содержит исполняемые команды SQL. Вам остается только запустить его. В предположении, что демонстрационная база данных Sakila выгружена в один файл вместе со схемой, для восстановления данных обычно используется такая команда:

```
$ mysql < sakila-backup.sql
```

Можно также загрузить файл из клиента *mysql* командой `SOURCE`. Хотя это, в общем-то, просто другой способ сделать то же самое, некоторые вещи при этом оказываются проще. Например, если вы обладаете административными правами в MySQL, то можете отключить запись в двоичный журнал команд, выполняемых в текущем соединении, а затем загрузить файл без перезапуска сервера:

```
mysql> SET SQL_LOG_BIN = 0;
mysql> SOURCE sakila-backup.sql;
mysql> SET SQL_LOG_BIN = 1;
```

Но при использовании команды `SOURCE` имейте в виду, что ошибка не прерывает выполнение пакета команд, как это происходит по умолчанию в случае чтения из стандартного ввода *mysql*.

Если резервная копия была сжата, то не нужно перед загрузкой отдельно распаковывать ее. Разархивирование и загрузку можно объединить в одной операции, что будет гораздо быстрее:

```
$ gunzip -c sakila-backup.sql.gz | mysql
```

О том как загрузить сжатый файл командой `SOURCE`, см. обсуждение именованных каналов в следующем разделе.

А вдруг нам понадобилось вернуть данные только для одной таблицы (например, `actor`)? Если данные не содержат символов перехода на новую строку и схема таблицы уже присутствует, то восстановить такую таблицу нетрудно:

```
$ grep 'INSERT INTO `actor`' sakila-backup.sql | mysql sakila
```

Или для сжатого файла:

```
$ gunzip -c sakila-backup.sql.gz | grep 'INSERT INTO `actor`'| mysql sakila
```

Если нужно не только восстановить информацию, но и создать таблицу, а вся база данных находится в одном файле, то этот файл придется отредактировать. Именно поэтому многие предпочитают выгружать каждую таблицу в отдельный файл. Большинство редакторов не умеют работать с очень большими файлами, особенно сжатыми. Но вам, собственно, и не требуется редактировать файл, достаточно лишь извлечь из него нужные строки, поэтому подойдут утилиты командной строки. Довольно просто воспользоваться инструментом *grep* для выборки только команд `INSERT`, относящихся к данной таблице (как показано выше), но извлечь команду `CREATE TABLE` сложнее. Ниже приведен сценарий *sed*, извлекающий ровно то, что необходимо:

```
$ sed -e '/./{H;$!d;}' -e 'x;/CREATE TABLE `actor`/!d;q' sakila-backup.sql
```

Да, признаем, выглядит загадочно. Если для восстановления данных вам приходится идти на подобные трюки, значит, процедура резервного копирования плохо продумана. Когда все распланировано заранее, можно будет избежать ситуаций, при которых вы в панике пытаетесь разобраться, как работает *sed*. Просто копируйте каждую таблицу в отдельный файл, а еще лучше – помещайте данные и схему в разные файлы.

Загрузка файлов с разделителями

При выгрузке данных командой `SELECT INTO OUTFILE` загружать их обратно следует командой `LOAD DATA INFILE` с теми же параметрами. Можно также воспользоваться программой *mysqlimport*, которая является не чем иным, как оберткой вокруг `LOAD DATA INFILE`. Куда загружать данные из файла, утилита *mysqlimport* определяет на основании соглашений об именовании.

Мы надеемся, что вы выгрузили не только данные, но и схему. В таком случае это SQL-дамп, и для его загрузки можно воспользоваться приемами из предыдущего раздела.

Для команды `LOAD DATA INFILE` существует замечательная оптимизация. Поскольку эта директива читает данные непосредственно из файла, то может возникнуть мысль предварительно его распаковать, а это очень длительная операция, которая к тому же активно обращается к диску. Но существует обходной путь, по крайней мере, в системах, поддерживающих «именованные каналы» (или FIFO-очереди), к каковым относится и GNU/Linux. Сначала создайте именованный канал и направьте в него распакованные данные:

```
$ mkfifo /tmp/backup/default/sakila/payment.fifo
$ chmod 666 /tmp/backup/default/sakila/payment.fifo
$ gunzip -c /tmp/backup/default/sakila/payment.txt.gz >
  /tmp/backup/default/sakila/payment.fifo
```

Обратите внимание на употребление знака > для перенаправления распакованного потока в файл *payment.fifo* – нужен именно он, а не знак '|', который создает анонимный канал между программами. *Payment.fifo* – именованный канал, поэтому анонимный нам ни к чему.

Канал ждет, пока какая-нибудь программа откроет его и начнет читать данные с другого конца. В этом-то и заключается изюминка: сервер MySQL может получать распакованные данные из канала точно так же, как из обычного файла. Не забудьте отключить запись в двоичный журнал, если она не нужна:

```
mysql> SET SQL_LOG_BIN = 0; -- Необязательно
    -> LOAD DATA INFILE '/tmp/backup/default/sakila/payment.fifo'
    -> INTO TABLE sakila.payment;
Query OK, 16049 rows affected (2.29 sec)
Records: 16049 Deleted: 0 Skipped: 0 Warnings: 0
```

После того как MySQL закончит загрузку данных, программа *gunzip* завершится, и именованный канал можно будет удалить. Такую же технику можно применять для загрузки сжатых файлов из командного клиента *mysql* директивой SOURCE.

Зачем проверять резервные копии?

Недавно один из авторов этой книги поменял тип столбца с DATETIME на TIMESTAMP, чтобы сэкономить место и ускорить обработку, как рекомендовано в главе 3. В результате определение таблицы стало выглядеть так:

```
CREATE TABLE tbl (
   col1 timestamp NOT NULL,
   col2 timestamp NOT NULL default CURRENT_TIMESTAMP
        on update CURRENT_TIMESTAMP,
   ... прочие столбцы ...
);
```

Такое определение таблицы приводит к синтаксической ошибке в версии MySQL 5.0.40, на которой оно было создано. Выгрузить данные удастся, а загрузить обратно – уже нет. Вот из-за таких странных, непредвиденных ошибок и надо проверять резервные копии. Никогда заранее не знаешь, что помешает вернуть данные!

Восстановление на конкретный момент времени

Самый распространенный способ восстановления на конкретный момент времени в MySQL заключается в том, чтобы вернуть данные с последней полной резервной копии, а затем воспроизвести двоичные журналы, накопившиеся за этот период (иногда это называется «накатом

журналов»). Имея двоичный журнал, можно восстановиться на любой момент времени. Можно даже без особого труда восстановить только одну базу данных.

Нередко возникает задача отменить действие необдуманно выполненной команды, например DROP TABLE. Рассмотрим упрощенный пример, на котором покажем, как это сделать, когда имеются только таблицы типа MyISAM.

Предположим, что в полночь задание резервного копирования выполнило следующие команды, которые копируют базу данных в другое место на том же сервере:

```
mysql> FLUSH TABLES WITH READ LOCK;
    -> server1# cp -a /var/lib/mysql/sakila /backup/sakila;
mysql> FLUSH LOGS;
    -> server1# mysql -e "SHOW MASTER STATUS" --vertical > /backup/master.info;
mysql> UNLOCK TABLES;
```

Позже в тот же день кто-то по ошибке выполнил такую команду:

```
mysql> USE sakila;
mysql> DROP TABLE sakila.payment;
```

Для простоты допустим, что мы можем корректно восстановить эту базу данных отдельно от других (то есть, в этой базе не существует таблиц, которые были вовлечены в запросы, затрагивающие несколько баз данных). Предположим еще, что ошибочная команда была обнаружена спустя некоторое время. Наша задача – восстановить все, что происходило с этой базой данных, за исключением вышеупомянутой команды. Иными словами, мы должны сохранить все обновления в других таблицах, в том числе имевшие место после злополучной команды.

Это не так уж трудно сделать. Прежде всего остановим MySQL, чтобы предотвратить дальнейшие модификации, и вернем из резервной копии только базу данных sakila:

```
server1# /etc/init.d/mysql stop
server1# mv /var/lib/mysql/sakila /var/lib/mysql/sakila.tmp
server1# cp -a /backup/sakila /var/lib/mysql
```

Запретим обычные соединения, временно добавив в файл *my.cnf* такие строчки:

```
skip-networking
socket=/tmp/mysql_recover.sock
```

Теперь можно без опаски запустить сервер:

```
server1# /etc/init.d/mysql start
```

Следующая задача – найти в двоичном журнале те команды, которые мы хотим воспроизвести, и те, которые следует пропустить. Как выясняется, после полуночи сервер создал только один двоичный журнал.

Мы можем обработать его утилитой *grep* и найти ошибочно выполненную команду:

```
server1# mysqlbinlog --database=sakila /var/log/mysql/mysql-bin.000215 |
grep -B 3 -i 'drop table sakila.payment'
# at 352
#070919 16:11:23 server id 1 end_log_pos 429 Query thread_id=16 exec_time=0
error_code=0
SET TIMESTAMP=1190232683/*!*/;
DROP TABLE sakila.payment/*!*/;
```

Итак, команда, которую мы хотим пропустить, начинается с позиции 352 в файле журнала, а следующая за ней – с позиции 429. Воспроизведем журнал до позиции 352 и далее с позиции 429 до конца:

```
server1# mysqlbinlog --database=sakila /var/log/mysql/mysql-bin.000215
--stop-position=352 | mysql -uroot -p
server1# mysqlbinlog --database=sakila /var/log/mysql/mysql-bin.000215
--start-position=429 | mysql -uroot -p
```

Теперь осталось только еще раз проверить данные, на всякий случай остановить сервер, вернуть файл *my.cnf* в исходное состояние и снова запустить сервер.

Более сложные методы восстановления

В основе репликации и восстановления на конкретный момент времени лежит один и тот же механизм: двоичный журнал. Это означает, что репликация может оказаться полезным, хотя и не вполне очевидным, инструментом восстановления. В этом разделе мы продемонстрируем некоторые возможности. Список не полон, но содержит некоторые идеи о том, как спроектировать процедуру восстановления с учетом своих потребностей. Еще раз напомним – оформляйте все, что может понадобиться в процессе восстановления, в виде сценария и прогоняйте его на тестовом сервере.

Быстрое восстановление с помощью отложенной репликации

Выше в этой главе мы уже отмечали, что наличие подчиненного сервера, реплицирующего данные с задержкой, может существенно упростить и ускорить восстановление на конкретный момент времени, если вы успели заметить случайную ошибку до того, как подчиненный сервер выполнил породившую ее команду.

Эта процедура немного отличается от описанной в предыдущем разделе, но идея та же самая. Останавливаем подчиненный сервер, а затем применяем команду `START SLAVE UNTIL`, чтобы воспроизвести события, предшествующие команде, которую нужно пропустить. Затем выполняем команду `SET GLOBAL SQL_SLAVE_SKIP_COUNTER=1`, чтобы пропустить ошибочную директиву. Если нужно проигнорировать несколько событий, задайте значение, большее 1 (или просто воспользуйтесь командой

CHANGE MASTER, чтобы продвинуть вперед позицию, с которой подчиненный сервер читает журнал).

Осталось только выполнить команду START SLAVE и дать подчиненному серверу возможность закончить обработку своих журналов повтора. В результате подчиненный сервер проделает за вас всю утомительную работу по восстановлению на конкретный момент времени. Затем делаем подчиненный сервер главным – и восстановление закончено, причем почти без прерывания обслуживания.

Даже если подчиненного сервера с отложенной репликацией нет, все равно такие серверы могут быть полезны, потому что копируют двоичные журналы главного сервера на другую машину. Если диск на главном сервере выйдет из строя, то журналы повтора на подчиненном могут оказаться единственным местом, где имеются сравнительно актуальные двоичные журналы главного сервера (еще безопаснее хранить двоичные журналы в SAN или реплицировать их с помощью технологии DRBD, как было рассказано в главе 9).

Восстановление с помощью сервера журналов

Существует еще один способ применить репликацию для восстановления: настроить сервер журналов (подробнее об этом см. в разделе «Создание сервера журналов» главы 8 на стр. 465).

Использование сервера журналов – более гибкий и простой путь восстановления, чем работа с программой *mysqlbinlog*, не только из-за наличия команды START SLAVE UNTIL, но и вследствие имеющихся правил репликации (например, replicate-do-table). При наличии сервера журналов можно реализовать гораздо более сложные правила фильтрации, чем без него.

Например, сервер журналов позволяет без труда восстановить одну таблицу. С помощью *mysqlbinlog* и утилит командной строки сделать это значительно труднее – настолько трудно, что мы не советуем даже пытаться.

Предположим, что наш беспечный разработчик ухитрился удалить таблицу, а нам нужно восстановить ее, не возвращая весь сервер в то состояние, которое имело место на момент снятия ночной копии. Вот как можно это сделать с помощью сервера журналов:

1. Пусть сервер, подлежащий восстановлению, называется server1.
2. Восстановим ночную копию на другом сервере, server2. Запустим на этом сервере процедуру восстановления, чтобы еще больше не ухудшить ситуацию в случае, если в ходе этого процесса будет допущена ошибка.
3. Настроим сервер журналов, так чтобы он обслуживал двоичные журналы сервера server1, следуя указаниям из раздела «Создание сервера журналов» в главе 8 на стр. 465 (было бы неплохо для страхов-

ки скопировать журналы на какой-нибудь другой сервер и настроить сервер журналов там).

4. Изменим конфигурационный файл `server2`, включив в него следующую строку:

```
replicate-do-table=sakila.payment
```

5. Перезапустим `server2` и сделаем его подчиненным серверу журналов с помощью команды `CHANGE MASTER TO`. Сконфигурируем его так, чтобы он читал двоичный журнал с позиции, соответствующей ночной копии. Но пока не будем выполнять команду `START SLAVE`.
6. Выполним на сервере `server2` команду `SHOW SLAVE STATUS` и убедимся, что все правильно. Семь раз отмерь, один отрежь!
7. Найдем в двоичном журнале место, с которого начинается ошибочная команда, и выполним команду `START SLAVE UNTIL`, чтобы воспроизвести все события вплоть до этой позиции.
8. Остановим процесс репликации на сервере `server2` командой `STOP SLAVE`. Теперь на этом сервере таблица существует в виде, непосредственно предшествующем удалению.
9. Скопируйте таблицу с `server2` на `server1`.

Все это возможно только в том случае, если данная таблица не участвует в командах `UPDATE`, `DELETE` или `INSERT`, которые выполняют многотабличные обновления. Любая такая команда будет выполняться в состоянии базы данных, отличном от того, которое существовало в момент записи событий в двоичный журнал, поэтому, скорее всего, после восстановления таблица будет содержать не ту информацию, что должна бы.

Восстановление InnoDB

InnoDB проверяет файлы данных и журналов при каждом запуске, чтобы понять, нужно ли начинать процедуру восстановления. Однако восстановление InnoDB – не то же самое, о чем мы говорим на протяжении всей этой главы. Речь идет не о воссоздании данных из резервной копии, а о применении к файлам данных тех транзакций, которые хранятся в журнале, и об откате незафиксированных транзакций.

Детали процедуры восстановления InnoDB слишком сложны, чтобы описывать их здесь. Вместо этого мы расскажем о том, как выполнить эту процедуру на практике, когда в работе InnoDB возникает серьезная проблема.

Обычно InnoDB прекрасно исправляет ошибки самостоятельно. Если только в самом коде MySQL нет дефекта и оборудование исправно, то ничего экстраординарного вам делать не придется, даже после отключения питания сервера. InnoDB просто произведет штатное восстановление при запуске, и все будет хорошо. В журнале ошибок вы увидите примерно такие сообщения:

```
InnoDB: Doing recovery: scanned up to log sequence number 0 40817239
InnoDB: Starting an apply batch of log records to the database...
```

InnoDB выводит в журнал сообщения о ходе восстановления (сколько уже выполнено в процентах). Некоторые пользователи жаловались, что не видят никаких сообщений, пока процесс не завершится. Но будьте терпеливы, ускорить эту процедуру невозможно. Если вы принудительно завершите ее и начнете заново, то будет только дольше.

Если имеет место серьезная неисправность оборудования, скажем, поврежден диск или память, или же вы наткнулись на ошибку в коде MySQL или InnoDB, то, возможно, придется вмешаться и либо принудительно запустить восстановление, либо отменить штатную процедуру.

Причины повреждения InnoDB

Вообще говоря, подсистема InnoDB очень надежна. Она проектировалась именно с этой целью, в нее встроено множество проверок, направленных на то, чтобы предотвратить порчу, а также найти и исправить повреждение данных, – гораздо больше, чем в некоторых других подсистемах хранения. Однако она не может защитить себя от всего на свете.

Как минимум, InnoDB полагается на то, что обращения к небуферизованному вводу/выводу и системному вызову `fsync()` не возвращают управление, пока данные не будут окончательно записаны на физический носитель. Если оборудование не гарантирует этого, то InnoDB не сможет защитить информацию и сбой приведет к ее повреждению.

Многие проблемы, связанные с порчей данных в таблицах InnoDB, обусловлены работой оборудования (например, неправильной записью в страницу из-за сбоя питания или ошибки памяти). Но наш опыт показывает, что гораздо чаще причиной является некорректная конфигурация устройств. К числу типичных ошибок можно отнести включение кэша записи на карте RAID-контроллера без резервного питания или включение кэша записи в самих накопителях на жестких дисках. Из-за таких ошибок контроллер «лжет», сообщая, что вызов `fsync()` завершен, тогда как на самом деле данные еще находятся в кэше записи, а не на диске. Другими словами, оборудование не дает гарантий, необходимых InnoDB для безопасного хранения данных.

Иногда компьютеры конфигурируются так по умолчанию, поскольку этот режим обеспечивает более высокую производительность, – для каких-то целей это, может быть, хорошо, но только не для транзакционной СУБД. Всегда необходимо тщательно проверять машину, которую вы настраивали не сами.

Повреждения возможны и тогда, когда InnoDB работает с хранилищем, подключенным по сети (NAS), поскольку завершение системного вызова `fsync()` в этом случае означает лишь, что устройство получило данные. Сами данные будут в безопасности, если «грохнется» InnoDB, но могут оказаться испорчены в случае отказа NAS-устройства.

Степень повреждения может быть различной. Серьезные ошибки способны вызвать аварийный останов InnoDB или сервера MySQL, а менее серьезные означают просто потерю некоторых транзакций из-за того, что файл журнала не был сброшен на диск.

Как восстановить поврежденные данные InnoDB

Существует три основных типа повреждений InnoDB, и в каждом случае для восстановления применяются различные методы.

Повреждение вторичного индекса

Исправить повреждение вторичного индекса часто удается с помощью команды `OPTIMIZE TABLE`. Как альтернативу можно использовать команду `SELECT INTO OUTFILE`, удалить и заново создать таблицу, а затем загрузить в нее данные командой `LOAD DATA INFILE`. Повреждение устраняется за счет того, что строится новая таблица и, следовательно, испорченный индекс тоже перестраивается.

Повреждение кластерного индекса

Возможно, нужно будет воспользоваться параметром `innodb_force_recovery`, чтобы выгрузить дамп таблицы (подробнее об этом ниже). Иногда в процессе выгрузки InnoDB аварийно завершает работу; тогда приходится выгружать таблицу по частям, чтобы обойти поврежденные страницы, вызывающие аварию сервера. Повреждение кластерного индекса серьезнее, чем вторичного, поскольку затронутыми оказываются сами строки с данными, но во многих случаях поврежденные таблицы все-таки можно исправить.

Повреждение системных структур

К системным структурам относятся журнал транзакций InnoDB, область отмены в табличном пространстве и словарь данных. Для исправления таких повреждений, скорее всего, потребуется произвести полную выгрузку и восстановление, потому что затронутыми могут оказаться многие внутренние механизмы работы InnoDB.

Обычно поврежденный вторичный индекс удается исправить без потери данных. Но в остальных двух случаях утрата какой-то части информации вполне вероятна. При наличии резервной копии лучше вернуть данные с нее, а не пытаться извлечь их из поврежденных файлов.

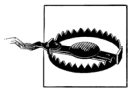
Если никакой альтернативы извлечению данных из поврежденных файлов нет, то сначала нужно запустить InnoDB, а потом выполнить команду `SELECT INTO OUTFILE` для выгрузки информации в файл. Если сервер аварийно завершил работу и не удается даже запустить InnoDB, то можно сконфигурировать его так, чтобы не загружались штатные процессы восстановления и фоновые потоки. Возможно, после этого удастся стартовать и сделать логическую копию с ослабленным контролем целостности или вовсе без него.

Параметр `innodb_force_recovery` управляет тем, какие операции InnoDB выполняет при запуске и во время нормальной работы. Обычно он ра-

вен 0, но может принимать целые значения вплоть до 6. В руководстве по MySQL описано поведение сервера в каждом из режимов; мы не будем здесь дублировать эту информацию, но отметим, что значения вплоть до 4 не представляют особой опасности. При таких настройках можно лишь потерять некоторые данные в поврежденных страницах, но, если пойти дальше, то не исключено извлечение некорректной информации из поврежденных страниц, а также повышается риск аварийного завершения во время выполнения команды `SELECT INTO OUTFILE`. Иными словами, уровни от 0 до 4 не наносят вреда данным, но InnoDB может не воспользоваться некоторыми возможностями их исправления; на уровнях 5 и 6 попытки ликвидировать повреждения становятся более настойчивыми, но это сопряжено с риском нанести ущерб данным.

Если параметр `innodb_force_recovery` больше 0, то InnoDB по существу работает в режиме чтения, но создавать и удалять таблицы все же можно. Это предотвращает развитие повреждений и заставляет InnoDB немного ослабить обычные проверки и не прекращать работу немедленно в случае обнаружения испорченных данных. В обычных условиях принудительное завершение работы при обнаружении испорченной информации служит предохранителем, но во время восстановления это нежелательно. Если нужно заставить InnoDB выполнить восстановление, то имеет смысл сконфигурировать MySQL так, чтобы обычные подключения отвергались до завершения процедуры.

Если данные InnoDB повреждены настолько сильно, что MySQL вообще не запускается, то можно воспользоваться комплектом инструментов InnoDB Recovery Toolkit, чтобы извлечь информацию напрямую из страниц табличного пространства. Эти инструменты были разработаны авторами настоящей книги, их можно бесплатно скачать с сайта *http://code.google.com/p/innodb-tools/*.



Обычно мы не упоминаем конкретных ошибок в коде MySQL, но во многих версиях есть один серьезный дефект, который не дает выполнять восстановление, если параметр `innodb_force_recovery` отличен от 0. Состояние этой ошибки можно посмотреть по адресу *http://bugs.mysql.com/28604*. Если при попытке выгрузить поврежденную таблицу InnoDB вы получаете сообщение «Incorrect key file», то прочитайте отчет об этой ошибке и посмотрите, не в ней ли дело. Если в этом и есть причина, то для выгрузки данных можно воспользоваться версией MySQL 5.0.22. Надеемся, что вам никогда не придется с этим столкнуться.

Скорость резервного копирования и восстановления

После правильности следующим по важности аспектом резервного копирования и восстановления в высокопроизводительных системах является скорость. Ниже перечислены факторы, которые следует учитывать.

Время блокировки

Как долго на протяжении резервного копирования приходится удерживать блокировки, например глобальную блокировку чтения, которую захватывает команда `FLUSH TABLES WITH READ LOCK`?

Время резервного копирования

Сколько времени занимает копирование резервной копии в место назначения?

Нагрузка, воздаваемая резервным копированием

Как отражается на производительности сервера операция копирования резервной копии в место назначения?

Время восстановления

Сколько времени занимает копирование резервной копии из того места, где она хранится, на сервер MySQL, воспроизведение двоичных журналов и т. д.?

Важнее всего соотношение между временем создания и загрузки резервной копии. Часто одно можно улучшить за счет другого; например, существует возможность повысить приоритет резервного копирования ценой снижения производительности сервера.

Можно также при проектировании процедуры резервного копирования учитывать характеристики рабочей нагрузки. Например, если в течение восьми ночных часов сервер загружен только на 50%, то ничто не мешает попытаться настроить резервное копирование так, чтобы оно нагружало сервер не более чем на 50%, и при этом успевало завершиться за 8 часов. Это реализуется разными способами: например, с использованием утилит *ionice* и *nice* для задания приоритета операций копирования или сжатия, с применением разных уровней компрессии или сжатием данных на сервере резервного копирования, а не на сервере MySQL. Можно также установить флаг `O_DIRECT` или воспользоваться системным вызовом *madvice*, чтобы обойти системный кэш при копировании, и тем самым не засорять кэши сервера.

В общем случае значительно быстрее и менее трудоемко создавать физические, а не логические копии. Однако логические копии являются важным дополнением, так как физические не настолько переносимы, с течением времени они могут стать непригодны для восстановления, и, кроме того, в них могут скрываться труднообнаруживаемые повреждения. Если периодически создавать логические копии из физических, то можно получить лучшее из обеих методик за сравнительно низкую плату.

Инструменты резервного копирования

Процедура восстановления, которая не ограничивается остановом сервера, возвратом данных и перезапуском сервера, может оказаться довольно сложной. Такие действия нужно тщательно репетировать и оформ-

лять в виде сценария. В последующих разделах мы познакомим вас с некоторыми инструментами, полезными как для написания сценариев резервного копирования и восстановления, так и для одноразовой выгрузки и загрузки дампа.

В первом издании этой книги было написано: «При наличии сложной конфигурации или необычных требований может оказаться, что какого-то одного из упоминаемых инструментов будет недостаточно. Тогда придется самостоятельно строить нестандартное решение». Но времена изменились, и сегодня мы не рекомендуем писать собственные инструменты резервного копирования, разве что ничего другого не остается. Скорее всего, какой-нибудь готовый инструмент прекрасно вам подойдет, а если нет, то вы сможете модифицировать его под свои потребности.

И все же в некоторых особо сложных ситуациях иногда приходится писать специальный сценарий, поэтому в конце главы мы включили несколько советов о том, как это делать.

Программа mysqldump

Самая популярная программа создания логических резервных копий данных и схемы – это *mysqldump*. Она поставляется вместе с сервером, так что вам даже не придется ее устанавливать. Это инструмент общего назначения, пригодный для решения многих задач, таких как копирование таблицы с одного сервера на другой.

```
$ mysqldump --host=server1 test t1 | mysql --host=server2 test
```

В этой главе мы уже приводили примеры создания логических копий с помощью *mysqldump*. По умолчанию эта программа выводит сценарий, содержащий все команды, необходимые для создания таблицы и заполнения ее данными; существуют также флаги для вывода представлений, хранимых процедур и триггеров. Вот несколько типичных примеров использования.

- Создание логической копии всего, что хранится на сервере, в виде одного файла:
  ```
  $ mysqldump --all-databases > dump.sql
  ```
- Создание логической копии базы данных Sakila:
  ```
  $ mysqldump --databases sakila > dump.sql
  ```
- Создание логической копии таблицы `sakila.actor`:
  ```
  $ mysqldump sakila actor > dump.sql
  ```

Флаг *--result-file* позволяет задать выходной файл, что предотвращает преобразование символов перехода на новую строку на платформе Windows:

```
$ mysqldump sakila actor --result-file=dump.sql
```

Параметры *mysqldump*, подразумеваемые по умолчанию, не годятся для серьезного резервного копирования. Было бы правильно явно задать некоторые флаги. Ниже перечислены флаги, которыми мы часто пользуемся для того, чтобы повысить эффективность *mysqldump* и упростить работу с генерируемыми файлами.

--opt

Относится к группе параметров, отключающих буферизацию (которая может привести к исчерпанию памяти сервера), порождает меньше SQL-команд в дампе, так что они загружаются более эффективно, и делает ряд других полезных вещей. Подробную информацию можно почерпнуть из справки. Если эту группу параметров отключить, то *mysqldump* будет сохранять каждую выгружаемую таблицу в памяти перед тем, как записать ее на диск; для больших таблиц это не практично.

--allow-keywords, *--quote-names*

Позволяет выгружать и загружать таблицы, имена которых совпадают с зарезервированными словами.

--complete-insert

Позволяет перемещать данные между таблицами с неодинаковыми столбцами

--tz-utc

Позволяет перемещать данные между серверами, находящимися в разных часовых поясах.

--lock-all-tables

Выполняет команду `FLUSH TABLES WITH READ LOCK` для получения глобально согласованной резервной копии.

--tab

Формирует файлы дампов с помощью команды `SELECT INTO OUTFILE`, которая позволяет выполнять выгрузку и загрузку очень быстро.

--skip-extended-insert

Выводит каждую строку данных в виде отдельной команды `INSERT`. Это позволит при необходимости избирательно восстанавливать некоторые строки. Но в таком случае большие файлы будут импортироваться в MySQL дольше, поэтому задавать этот флаг нужно с осторожностью.

При задании флагов *--databases* или *--all-databases* сформированные для каждой базы дампы будут содержать согласованные данные, потому что *mysqldump* блокирует и выгружает все таблицы в базе одновременно. Однако таблицы из разных баз все же могут оказаться рассогласованными. Флаг *--lock-all-tables* решает и эту проблему.

Сценарий mysqlhotcopy

mysqlhotcopy – это написанный на языке Perl сценарий, который включен в стандартный дистрибутив MySQL. Он был разработан для таблиц типа MyISAM и, на наш взгляд, *не* предназначен для создания «горячих» резервных копий, так как перед копированием блокирует все таблицы. Когда-то этот сценарий был одним из самых распространенных инструментов резервного копирования на работающем сервере, но в наши дни подрастерял популярность. Многие высокопроизводительные приложения уходят от использования таблиц типа MyISAM и даже если вы по-прежнему работаете только с такими таблицами, то все равно снимки файловой системы удобнее, так как блокируют доступ к данным на более короткое время.

В качестве примера мы создали копию демонстрационной базы данных Sakila, состоящей только из таблиц типа MyISAM. Чтобы выгрузить ее в каталог */tmp*, мы запустили следующую команду:

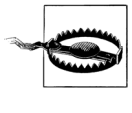
```
$ mysqlhotcopy sakila_myisam /tmp
```

В результате был создан подкаталог *sakila_myisam*, содержащий все таблицы базы данных:

```
$ ls -l /tmp/sakila_myisam/
total 3632
-rw-rw---- 1 mysql mysql 8694 2007-09-28 09:57 actor.frm
-rw-rw---- 1 mysql mysql 5016 2007-09-28 09:57 actor.MYD
-rw-rw---- 1 mysql mysql 7168 2007-09-28 09:57 actor.MYI
... опущено ...
-rw-rw---- 1 mysql mysql 8708 2007-09-28 09:57 store.frm
-rw-rw---- 1 mysql mysql   18 2007-09-28 09:57 store.MYD
-rw-rw---- 1 mysql mysql 4096 2007-09-28 09:57 store.MYI
```

Для каждой таблицы скопированы файлы данных, индексов и определений. С целью экономии места можно задать флаг *--noindices*, тогда будут копироваться только первые 2048 байтов каждого MYI-файла, то есть тот минимум, который необходим серверу MySQL для воссоздания индекса. Если задан этот флаг, то после возврата информации нужно будет перестроить все индексы. Для этой цели можно либо воспользоваться программой *myisamchk* с флагом *--recover*, либо выполнить SQL-команду `REPAIR TABLE`.

Сценарий *mysqlhotcopy* довольно сложен и не отличается большой гибкостью, поэтому многие пользователи писали собственные сценарии для выполнения тех же операций, но немного другим способом.



mysqlhotcopy копирует файлы с расширением *.ibd*, если подсистема InnoDB сконфигурирована в режиме `innodb_file_per_table`, но это бесполезно. Да не вселит это в вас ложное чувство безопасности – так копировать данные InnoDB вовсе небезопасно.

Программа InnoDB Hot Backup

Программа InnoDB Hot Backup, *ibbackup*, – это коммерческий инструмент, распространяемый авторами InnoDB (компания Innobase). Он не требует останавливать MySQL, захватывать блокировки или прерывать нормальную работу базу данных (хотя увеличивает нагрузку на сервер). Кроме того, эта программа умеет сжимать резервные копии.

Для настройки *ibbackup* нужно создать конфигурационный файл, совпадающий с файлом *my.cnf* промышленного сервера, но находящийся в другом каталоге. Программа читает оба файла и копирует файлы InnoDB с промышленного сервера в место, указанное во втором конфигурационном файле:

```
$ ibbackup /etc/my.cnf /etc/ibbackup.cnf
```

Чтобы вернуть данные из резервной копии, остановите MySQL и выполните следующую команду:

```
$ ibbackup --restore /etc/ibbackup.cnf
```

Существует одна небольшая проблема: *ibbackup* копирует только файлы InnoDB, но не определения таблиц и прочие необходимые элементы. Компания Innobase предлагает также вспомогательный сценарий *innobackup*, который включает копирование файлов, блокировку таблиц и вызов *ibbackup*, так что одной командой можно осуществить резервное копирование всех определений таблиц, а также файлов MyISAM наряду с InnoDB. В отличие от самой программы *ibbackup*, этот сценарий все-таки прерывает нормальную работу СУБД, так как выполняет команду `FLUSH TABLES WITH READ LOCK`. Сценарий поставляется бесплатно.

На наш взгляд, снимки LVM более удобны и полезны для резервного копирования InnoDB, чем программа *ibbackup*. Одна из замечательных особенностей LVM состоит в том, что нет нужды создавать вторую копию данных в файловой системе, – вы можете сделать снимок, выполнить на нем резервное копирование InnoDB, а затем отправить копию прямиком в место хранения.

Производительность LVM и *ibbackup* сопоставима и зависит от того, как сконфигурирована процедура резервного копирования, а также от интенсивности операций записи. Если запись ведется интенсивно, то высоки накладные расходы на копирование при сохранении в снимок LVM. С другой стороны, *ibbackup* не всегда линейно масштабируется с ростом объема данных. Принцип ее работы основан на постраничном копировании файлов данных с последующим накатыванием на скопированные файлы журнала, чтобы к моменту завершения процедуры резервного копирования все данные относились к одной и той же временной точке.

Программа mk-parallel-dump

Этот инструмент входит в состав комплекта Maatkit (*http://maatkit.sourceforge.net*). Он выполняет несколько операций резервного копирования.

По умолчанию *mk-parallel-dump* работает как многопоточная обертка вокруг *mysqldump*, но может также выполнять экспорт в файлы с разделителями-табуляторами с помощью команды `SELECT INTO OUTFILE`. По умолчанию создается один поток на каждый имеющийся процессор, так что чем больше процессоров, тем быстрее работает программа. Она также умеет разбивать таблицу на порции заданного размера, что существенно ускоряет восстановление таблиц InnoDB. Преимущество заключается в возможности избежать больших транзакций на этапе возврата данных. Из-за большой транзакции табличное пространство InnoDB может вырасти до необъятных размеров, а время отката в случае ошибки резко возрастет.

У этого инструмента есть и другие приятные особенности, например он умеет делать инкрементные резервные копии и группировать таблицы в логические наборы. Эталонные тесты показывают заметное ускорение при параллельном создании логических резервных копий.

В комплект Maatkit входит также программа *mk-parallel-restore*, позволяющая выполнять многопоточный импорт данных. Оба инструмента широко применяют такие распространенные в системе UNIX средства, как анонимные и именованные каналы, чтобы сократить издержки на сжатие и распаковку файлов.

Сценарий mylvmbackup

Написанный Ленцем Гриммером (Lenz Grimmer) на языке Perl сценарий *mylvmbackup* (*http://lenz.homelinux.org/mylvmbackup/*) призван автоматизировать создание резервных копий MySQL с помощью снимков LVM. Он получает глобальную блокировку чтения, создает снимок, а затем снимает блокировку. После этого данные упаковываются с помощью программы *tar*, и снимок удаляется. Архиву присваивается имя, основанное на времени создания копии.

Программа Zmanda Recovery Manager

Программа Zmanda Recovery Manager для MySQL, или ZRM (*http://www.zmanda.com*), – наиболее полный из всех упоминаемых нами инструментов резервного копирования и восстановления. Она распространяется в виде бесплатной (на условиях лицензии GPL) и коммерческой версий. В состав редакции для предприятия (Enterprise Edition) входит консоль управления, реализующая графический веб-интерфейс конфигурирования, резервного копирования, верификации, восстановления, генерации отчетов и запуска по расписанию. Эта редакция может также выполнять резервное копирование кластера MySQL Cluster и предполагает все обычные бонусы (например, техническую поддержку).

Редакцию с открытым исходным кодом ни в коем случае нельзя назвать сильно урезанной, но в нее все же не включены некоторые дополнительные «приятности», например веб-консоль. Но если вам комфортно работать с командной строкой, то она полностью функциональна. Например, оставлена возможность задавать расписание запуска и проверять резервные копии из командной строки.

ZRM – скорее «координатор резервного копирования», нежели единый инструмент. Его собственная функциональность построена как обертка над такими стандартными инструментами и методиками, как *mysqldump* и снимки LVM, а данные он хранит в стандартных форматах, так что нет необходимости приобретать коммерческое ПО для восстановления из копии. Одна из полезных особенностей – унифицированный механизм восстановления, который работает одинаково вне зависимости от того, как снималась копия.

На рис. 11.2 показан календарь (из корпоративной версии) резервного копирования вместе с анализатором двоичного журнала, которые компания Zmanda называет «Database Events Viewer». По сути дела, это средство поиска в двоичных журналах. Пользуясь обычным синтаксисом поисковых запросов, вы можете искать интересующие события, что упрощает восстановление до конкретного события или на конкретный момент времени.

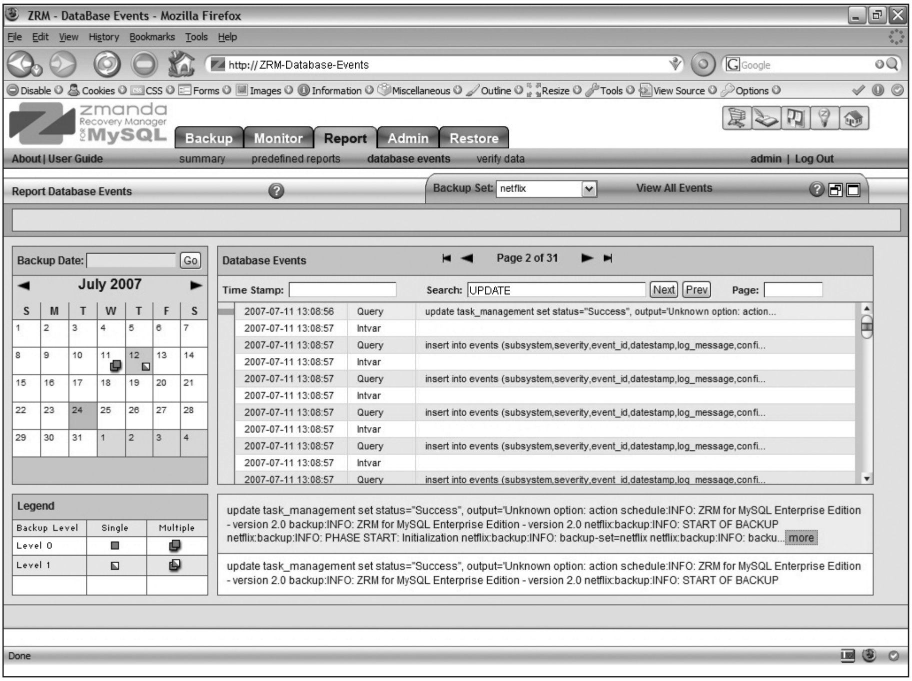


Рис. 11.2. *Интерфейс ZRM к календарю резервного копирования и анализатору двоичных журналов*

Установка и тестирование ZRM

На сайте компании Zmanda утверждается, что для установки программы, снятия и тестирования резервной копии, настройки и проверки ежедневного расписания, а также последующего восстановления из копии требуется примерно 15 минут. В качестве теста мы установили ZRM с нуля на ноутбук под управлением ОС Ubuntu Linux. Сам пакет совсем невелик и для его инсталляции мы выполнили команду *sudo dpkg -i mysql-zrm_1.2.1_all.deb*. Пакету необходимо несколько зависимостей, но они легко устанавливаются командой *sudo apt-get -f install*. Весь процесс занял меньше минуты.

Следуя инструкциям на сайте, мы сконфигурировали набор резервного копирования, который в нашем случае состоял из логической копии демонстрационной базы данных Sakila. На это ушло примерно три минуты. Затем мы ввели следующую команду для снятия резервной копии:

```
# mysql-zrm-scheduler --now --backup-set dailyrun
```

Собственное копирование заняло какие-то секунды, а результирующий файл был помещен в каталог */var/lib/mysqlzrm/dailyrun*. Затем мы снова запустили копирование, но на этот раз намеренно «напакостили», поубивав кое-какие из дочерних процессов ZRM, и задали неправильные учетные данные. Программа корректно обнаружила ошибки и отметила их в отчете о резервном копировании, который отправила по электронной почте. Все подробности были запротоколированы в системном журнале, как и ожидалось.

Наконец, мы удалили базу данных `sakila` и восстановили ее из последней успешно снятой копии, для чего ввели следующие команды:

```
# mysql-zrm-reporter --show restore-info --where backup-set=dailyrun
# mysql-zrm-restore --backup-set dailyrun --source-directory
/var/lib/mysql-zrm/dailyrun/20070930134242/
```

В общем и целом ZRM – хорошо продуманная система с добротным контролем ошибок, которая позволяет автоматизировать значительную часть рутинной работы по резервному копированию и восстановлению. И, как следует из названия, она с самого начала проектировалась с учетом восстановления.

R1Soft

Компания R1Soft (*http://www.r1soft.com*) предлагает продукт под названием Continuous Data Protection, который похож на снимки файловой системы с тем отличием, что при копировании снимка на другой сервер дублируются только изменившиеся блоки. Он позволяет производить откат до любой из нескольких прошлых версий данных. Это коммерческая программа.

MySQL Online Backup

MySQL Online Backup – это не инструмент, а механизм, который разрабатывается для версии MySQL 5.2 (в настоящее время существует альфа-версия) и, скорее всего, будет включен в состав MySQL 6.0.

Интерфейсом к нему служит новая SQL-команда BACKUP DATABASE, которая с очень высокой скоростью записывает согласованные снимки каждой таблицы в файл. При этом используется либо стандартный драйвер, умеющий делать резервную копию для любой подсистемы хранения, либо драйвер, написанный специально для конкретной подсистемы и копирующий ее более эффективно. Стандартный драйвер блокирует другие SQL-команды, тогда как специализированные могут производить резервное копирование без блокировки. Включена также функция восстановления.

На момент работы над этой книгой в дерево исходного кода для версии 5.2 была включена начальная реализация и завершена работа над внушительным перечнем функций, но остается еще немало нереализованных возможностей, например специализированный драйвер для MyISAM и согласованное резервное копирование таблиц разных типов.

Механизм оперативного резервного копирования ожидают с большим нетерпением, и по завершении разработки он, вероятно, станет одним из самых важных способов резервного копирования MySQL.

Сравнение инструментов резервного копирования

В табл. 11.2 приведена краткая сводка некоторых из обсуждавшихся в этой главе методов резервного копирования.

Таблица 11.2. Характеристики инструментов резервного копирования

	mylvmbackup	**mysqldump**	**mk-parallel-dump**	**mysqlhotcopy**	**ibbackup**
Поблочная обработка	Дополнительно	Да	Да	Да	Нет
Логическое или физическое	Физическое	Логическое	Логическое	Физическое	Физическое
Подсистемы хранения	Все	Все	Все	MyISAM/ Archive	InnoDB
Скорость	Очень высокая	Низкая	Высокая	Очень высокая	Очень высокая
Удаленное снятие копии	Нет	Да	Да	Нет	Нет
Доступность	Бесплатно	Бесплатно	Бесплатно	Бесплатно	Коммерческая
Лицензия	GPL	GPL	GPL	GPL	Специальная

Сценарии резервного копирования

Мы рекомендуем не изобретать велосипед, если уже есть готовая система, но иногда все же приходится придумывать собственный сценарий или модифицировать существующий. Ниже перечислено несколько конфигураций резервного копирования, с которыми мы встречались в реальной практике.

- Копирование нескольких серверов на некоторое количество серверов резервного копирования, оснащенных емкими дешевыми дисками без RAID-массива. Сценарий выделяет различные тома для каждой резервной копии, исходя из того, сколько места имеется на каждом сервере. Кроме того, он гарантирует, что разные поколения резервных копий записываются на разные серверы, так чтобы выход из строя одного любого сервера не привел к катастрофе.
- Разбиение архива резервной копии на части, шифрование и сохранение их вне центра обработки данных, в Amazon S3 или другой службе предоставления больших хранилищ.
- Интеграция восстановления с репликацией, так чтобы можно было заново клонировать подчиненный сервер из копии.

Вместо того чтобы приводить пример демонстрационной программы, что только заняло бы место на страницах книги, мы перечислим составные части типичного сценария и покажем несколько фрагментов на языке Perl. Можете считать их кирпичиками, из которых можно собрать ваш собственный сценарий. Мы приведем их примерно в том порядке, в котором они используются.

Проверка корректности

Облегчите жизнь себе и своим коллегам – включите строгий контроль ошибок и используйте английские имена переменных:

```
use strict;
use warnings FATAL => 'all';
use English qw(-no_match_vars);
```

Если сценарий пишется на языке оболочки bash, то можно также включить строгий контроль переменных. Следующие команды заставляют интерпретатор выдавать соответствующее сообщение, если в подстановке встретится неопределенная переменная или какая-то программа завершится с ошибкой.

```
set -u;
set -e;
```

Аргументы командной строки

Любой сценарий должен принимать аргументы командной строки. Если вы ловите себя на том, что «зашиваете» в код такие параметры,

как имена и пароли пользователей, то подумайте о том, чтобы работать на более высоком уровне.

```
use Getopt::Long;
Getopt::Long::Configure('no_ignore_case', 'bundling');
GetOptions( .... );
```

Подключение к серверу MySQL

Стандартный Perl-модуль DBI доступен практически везде и обеспечивает необходимую выразительность и гибкость. О том, как с ним работать, прочитайте документацию, сгенерированную программой *perldoc* (имеется на сайте *http://search.cpan.org*).

```
use DBI;
$dbh = DBI->connect(
       'DBI:mysql:;host=localhost', 'user', 'pass', {RaiseError => 1 });
```

О написании командных сценариев прочитайте оперативную справку, которую программа *mysql* выводит при задании флага *--help*. У нее масса параметров, облегчающих использование в составе сценариев. Вот, например, как можно перебрать список баз данных в bash-сценарии:

```
for DB in `mysql --skip-column-names --silent --execute 'SHOW DATABASES'`
do
   echo $DB
done
```

Останов и запуск MySQL

Самый простой способ остановить и запустить сервер MySQL – воспользоваться методом, принятым в вашей операционной системе, например выполнить сценарий инициализации */etc/init.d/mysql* или воспользоваться диспетчером служб (для Windows). Но это не единственный способ. Можно остановить СУБД из Perl-сценария, имея какое-нибудь соединение с базой данных:

```
$dbh->func("shutdown", 'admin');
```

Не следует считать, что MySQL остановлен после того, как эта команда завершится, – возможно, процесс останова все еще идет. Можно также остановить MySQL из командной строки:

```
$ mysqladmin shutdown
```

Получение списков баз данных и таблиц

Любой сценарий резервного копирования запрашивает у MySQL список баз данных и таблиц. Помните, что в этот список могут входить и не настоящие базы данных, например каталог *lost+found* в некоторых журналирующих файловых системах, а также `INFORMATION_SCHEMA`. Убедитесь, что ваш сценарий правильно обрабатывает представления, и не забывайте, что команда `SHOW TABLE STATUS` может за-

нять много времени, когда вы имеете дело с большими объемами данных в таблицах типа InnoDB:

```
mysql> SHOW DATABASES;
 mysql> SHOW /*!50002 FULL*/ TABLES FROM <database>;
 mysql> SHOW TABLE STATUS FROM <database>;
```

Блокировка, сброс и разблокировка таблиц

Иногда необходимо выполнять блокировку и/или сброс одной или нескольких таблиц. Можно либо заблокировать все нужные таблицы, перечислив их имена, либо захватить глобальную блокировку:

```
mysql> LOCK TABLES <database.table> READ [, ...];
mysql> FLUSH TABLES;
mysql> FLUSH TABLES <database.table> [, ...];
mysql> FLUSH TABLES WITH READ LOCK;
mysql> UNLOCK TABLES;
```

Не забывайте о возможной гонке при получении списков таблиц и их блокировке. Могут быть созданы новые таблицы, а старые удалены или переименованы. Если вы блокируете и копируете таблицы по одиночке, то согласованной копии не получится.

Сброс двоичных журналов

Очень полезно попросить сервер начать новый двоичный журнал (делайте это после блокировки таблиц, но перед тем, как начинать резервное копирование):

```
mysql> FLUSH LOGS;
```

Восстановление и инкрементное резервное копирование будут проще, если не придется думать о том, с какого места в середине файла журнала начинать. Но возможны побочные эффекты, связанные со сбросом и открытием новых журналов, а, кроме того, есть шанс уничтожить старые записи, поэтому внимательно следите за тем, чтобы не отбросить данные, которые могут еще понадобиться.

Получение позиции в двоичном журнале

Сценарий должен получить и куда-то записать состояние как главного, так и подчиненного сервера, даже если данный сервер выступает только в роли главного или только в роли подчиненного:

```
mysql> SHOW MASTER STATUS;
mysql> SHOW SLAVE STATUS;
```

Выполните обе команды и игнорируйте возможные ошибки, чтобы сценарий получил всю возможную информацию.

Выгрузка данных

Ваши лучшие друзья – утилита *mysqldump* и команда `SELECT INTO OUTFILE`.

Копирование данных

Используйте какой-либо из методов, рассмотренных в этой главе.

Это кирпичики, из которых состоит любой сценарий резервного копирования. Самая трудная часть – написать сценарий восстановления. Если вам хочется узнать, как это сделать правильно, изучите исходный код системы ZRM. Входящие в ее состав сценарии делают кое-какие интересные вещи, например сохраняют вместе с каждой резервной копией метаданные, чтобы потом было проще произвести с нее восстановление.

12

Безопасность

Безопасность сервера MySQL критична для обеспечения целостности и конфиденциальности данных. Точно так же, как при защите учетных записей в UNIX или Windows, нужно следить за тем, чтобы учетные записи MySQL имели устойчивые к перебору пароли и выдавались лишь те привилегии, которые необходимы конкретному пользователю. Поскольку MySQL часто используется в сети, необходимо также позаботиться о безопасности компьютера, на котором она работает: подумать, кто имеет к нему доступ и что можно узнать путем прослушивания сетевого трафика.

В MySQL применяется нестандартная система защиты и привилегий, которая позволяет решать многие специализированные задачи. В ее основу положен набор простых правил, но существуют также многочисленные исключения и особые случаи, так что с налету разобраться в ней нелегко. В этой главе мы рассмотрим то, как работают разрешения в MySQL, и расскажем, как можно управлять получением доступа к данным. В руководстве по MySQL система привилегий документирована весьма подробно, поэтому мы лишь разъясним запутанные концепции и покажем, как решать некоторые типичные задачи. Мы также поговорим об общих мерах защиты на уровне операционной системы и сети, которые позволят держать «плохих парней» подальше от ваших баз данных. И наконец, мы обсудим вопрос о шифровании и эксплуатации MySQL в условиях строгого ограничения (такие условия иногда называют «песочницей»).

Терминология

Прежде всего, определим некоторые термины, которые иногда понимают по-разному. В этой главе мы будем придавать им описанный ниже смысл.

Аутентификация

Кто вы такой? MySQL аутентифицирует пользователя по имени, паролю и компьютеру, с которого устанавливается соединение. Знание личности пользователя – обязательное условие для назначения ему тех или иных привилегий.

Авторизация

Что вам разрешено делать? Например, для останова сервера необходимо иметь привилегию `SHUTDOWN`. В MySQL авторизация применяется к глобальным привилегиям, не связанным с какими-то конкретными объектами схемы (например, таблицами или базами данных).

Контроль доступа

Какие данные вам разрешено видеть и/или изменять? При попытке прочитать или модифицировать информацию MySQL проверяет, есть ли у пользователя разрешение на просмотр либо изменение запрошенных столбцов. В отличие от глобальных привилегий, правила контроля доступа применяются к конкретным данным, например к отдельной базе, таблице или столбцу.

Привилегии и разрешения

Этими словами обозначается примерно одно и то же. Привилегия или разрешение – это способ представления в MySQL права доступа.

Основы учетных записей

В MySQL учетные записи отличаются от большинства систем, поскольку MySQL считает место, из которого произведена попытка входа в систему, частью аутентификации. Напротив, при входе в ОС UNIX обычно проверяются только имя и пароль пользователя. Иными словами, первичным ключом для учетных записей в UNIX является имя пользователя, а в MySQL – его имя и местоположение (обычно задается в виде имени компьютера, IP-адреса или маски адреса).

Как мы увидим ниже, ассоциирование местоположения с учетной записью привносит сложность в систему, которая иначе была бы довольно простой. Пользователь *joe*, пытающийся войти в систему с компьютера *joe.example.com*, – это не обязательно тот же *joe*, который входит с компьютера *sally.example.com*. С точки зрения MySQL это совершенно различные пользователи с разными паролями и привилегиями. С другой стороны, это может быть один и тот же пользователь. Все зависит от того, как сконфигурированы учетные записи.

Привилегии

Для аутентификации пользователя MySQL использует информацию об учетной записи (имя пользователя, пароль и местоположение). Если аутентификация прошла успешно, то сервер должен решить, что пользо-

вателю разрешено делать. Для этого он анализирует *привилегии*, чьи имена обычно совпадают с названиями SQL-команд, которые разрешено выполнять. Например, для выборки данных из таблицы необходима привилегия SELECT на эту таблицу.

Существует два вида привилегий: ассоциированные и не ассоциированные с объектами (таблицами, базами данных, представлениями). *Объектные привилегии* разрешают доступ к конкретным объектам. Например, с их помощью можно контролировать, разрешена ли выборка из таблицы, изменение структуры таблицы, создание нового представления или триггера. В версии MySQL 5.0 появилось много новых объектных привилегий в связи с добавлением в СУБД таких средств, как представления, хранимые процедуры и т. д.

С другой стороны, *глобальные привилегии* допускают такие действия, как останов сервера, выполнение команд FLUSH, различных команд SHOW и просмотр запросов других пользователей. В общем, глобальные привилегии позволяют что-то делать с самим сервером, а объектные – с содержимым, хранящимся на сервере (хотя это различие не всегда отчетливо). Каждая выданная глобальная привилегия существенно отражается на безопасности, поэтому раздавайте их с особой осторожностью!

Привилегии в MySQL подчиняются булевской логике: они либо выданы, либо нет. В отличие от других СУБД, в MySQL нет понятия «явно запрещенных привилегий». Отзыв привилегии не запрещает пользователю выполнять некоторое действие, а просто удаляет привилегию на его выполнение, если таковая существовала.

Таблицы привилегий

Для хранения пользователей и их привилегий в MySQL используется ряд *таблиц привилегий* (*grant table*). Это обычные таблицы типа MyISAM[1], находящиеся в базе данных mysql. У хранения информации о безопасности в таблицах привилегий много достоинств, но с другой стороны это означает, что если сервер сконфигурирован неправильно, то любой пользователь сможет изменить параметры безопасности, модифицировав данные в этих таблицах!

Таблицы привилегий – сердце системы безопасности MySQL. Современные версии этой СУБД предоставляют администратору практически полный контроль над безопасностью с помощью команд GRANT, REVOKE и DROP USER (ниже мы их рассмотрим). Однако раньше прямые манипуляции с таблицами безопасности были единственным способом решения некоторых задач. Например, в старых версиях MySQL для удаления пользователя нужно было сначала выполнить команду DELETE в таблице user, а затем команду FLUSH PRIVILEGES.

---

[1] И они должны оставаться таблицами типа MyISAM. Не изменяйте для них подсистему хранения.

Мы не рекомендуем изменять таблицы привилегий напрямую, но для того чтобы разобраться в их неожиданном поведении, нужно все же понимать, как они работают. Мы призываем вас исследовать структуру таблиц привилегий с помощью команд `DESCRIBE` или `SHOW CREATE TABLE`, особенно после изменения привилегий командой `GRANT` или `REVOKE`. Практикуясь, вы узнаете гораздо больше.

Ниже описаны таблицы привилегий в том порядке, в котором MySQL просматривает их, когда хочет узнать, разрешено ли пользователю выполнять ту или иную операцию.

`user`

В каждой строке хранятся учетные данные пользователя (имя, местоположение и зашифрованный пароль) и его глобальные привилегии. В версии MySQL 5.0 добавились лимиты, выделенные данному пользователю, например количество соединений, которые он может установить одновременно.

`db`

В каждой строке хранятся привилегии уровня базы данных для одного пользователя.

`host`

В каждой строке хранятся привилегии доступа к одной базе данных для пользователя, подключающегося из заданного местоположения. При проверке возможности доступа к базе данных эти сведения объединяются с записями из таблицы *db*. Хотя мы отнесли эту таблицу к таблицам привилегий, команды `GRANT` и `REVOKE` ее не модифицируют. Записи в нее добавляются и удаляются вручную.

Мы рекомендуем никогда не использовать эту таблицу, если вы не хотите потом столкнуться с непонятным поведением и проблемами при сопровождении.

`tables_priv`

В каждой строке хранятся привилегии доступа одного пользователя к одной таблице. Здесь же содержатся привилегии доступа к представлениям.

`columns_priv`

В каждой строке хранятся привилегии доступа одного пользователя к одному столбцу.

`procs_priv` *(появилось в MySQL 5.0)*

В каждой строке представлены привилегии доступа одного пользователя к одной хранимой подпрограмме (процедуре или функции).

Как MySQL проверяет привилегии

При проверке привилегий MySQL просматривает таблицы привилегий в том порядке, в котором они перечислены выше. Обнаружив первое же

соответствие, разрешающее искомую привилегию, сервер прекращает проверку. Иными словами, если в таблице db найдена запись, разрешающая запрошенный вид доступа, то таблица tables_priv не просматривается вовсе. Этот процесс показан на рис. 12.1.

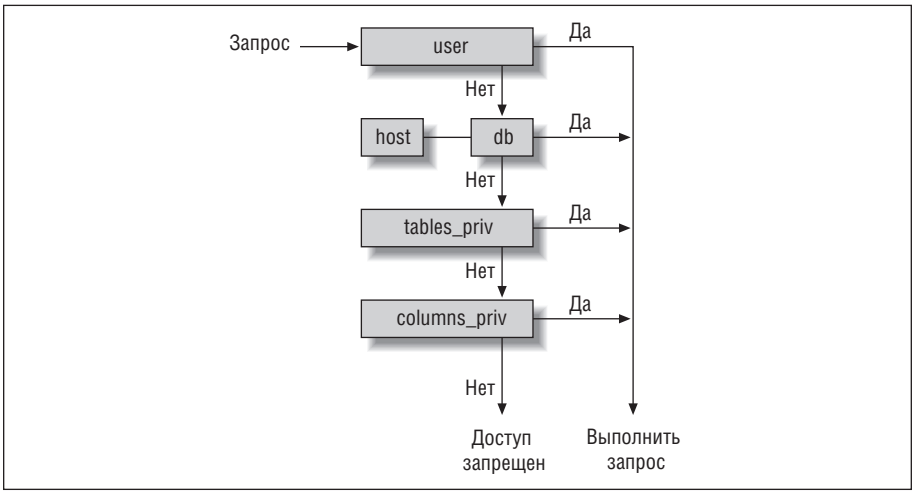


***Рис. 12.1.** Как MySQL проверяет привилегии*

MySQL определяет, какие привилегии применить, выполняя эквивалент команды SELECT для кэшированных таблиц привилегий. Во фразе WHERE этой виртуальной команды указаны столбцы, составляющие первичный ключ таблицы. Для некоторых столбцов допускается сопоставление с шаблоном, а для других определены «магические» значения (например, пустые), требующие специальной обработки. Детали можно посмотреть в руководстве по MySQL.

На изучение структуры и принципов работы таблиц привилегий можно потратить много времени, и иногда эти знания оказываются полезными. Однако мы не рекомендуем отвлекаться на это, если не возникает острой необходимости. Лучше прочитайте следующий раздел. Глубоко вникать в таблицы привилегий имеет смысл только если вы окажетесь в ситуации, которую невозможно (или очень трудно) разрешить с помощью одних лишь команд GRANT и REVOKE.

Добавление, удаление и просмотр привилегий

Рекомендуемый способ создавать учетные записи, а также добавлять и удалять привилегии – команды GRANT и REVOKE, которые хорошо документированы в руководстве по MySQL. Их синтаксис достаточно прост и в большинстве случаев этих команд вполне достаточно, чтобы не вникать в то, как устроены таблицы привилегий и как обрабатываются различные правила. Чтобы добавить новую учетную запись или приви-

легию, используйте директиву GRANT, однако команда REVOKE может удалить только привилегию, но не учетную запись – для удаления последней предназначена команда DROP USER.

Для просмотра привилегий пользователя существует команда SHOW GRANTS. Она показывает команду, которая воссоздает текущие привилегии. Например, вот как выглядит ее результат после входа под именем *root* в систему Debian, настроенную по умолчанию:

```
mysql> SHOW GRANTS;
+---------------------------------------------------------------------+
| Grants for root@localhost                                           |
+---------------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' WITH GRANT OPTION |
+---------------------------------------------------------------------+
```

Эта команда демонстрирует привилегии для пользователя, подразумеваемого по умолчанию, поэтому ее удобно применять с целью выяснить, от чьего имени вы вошли и каковы ваши текущие привилегии. У показанного пользователя имеются вообще все привилегии, но нет пароля, то есть он может войти в систему без пароля[1]. Абсолютно недопустимо с точки зрения безопасности! При настройке нового экземпляра MySQL это первое, на что нужно обратить внимание.

Чтобы просмотреть привилегии какого-то другого пользователя, необходимо указать его имя и адрес хоста. Например, в той же системе Debian в таблице user есть такие записи:

```
+------------------+-----------+
| user             | host      |
+------------------+-----------+
| repl             | %         |
| root             | 127.0.0.1 |
| root             | kanga     |
| debian-sys-maint | localhost |
| root             | localhost |
+------------------+-----------+
```

Обратите внимание, что есть целых три учетных записи *root*! Если вы захотите посмотреть привилегии для какой-то одной из них, то должны будете указать одновременно имя пользователя и имя хоста. По умолчанию подразумевается имя хоста %, так что, если его опустить, произойдет ошибка:

```
mysql> SHOW GRANTS FOR root;
ERROR 1141 (42000): There is no such grant defined for user 'root' on host '%'
```

Команда GRANT для пользователя без указания имени хоста по существу эквивалентна заданию условия *user*@'%' (то есть на любом хосте).

[1] Единственное, что как-то примиряет с этой ситуацией, – тот факт, что пользователь не сможет войти с другого компьютера, но это небольшое утешение.

Ничто не мешает манипулировать таблицами привилегий напрямую с помощью команд INSERT, UPDATE и DELETE, но, работая только с командами GRANT и REVOKE, вы страхуете себя от возможных изменений в структуре этих таблиц в будущих версиях. Например, MySQL позволит записать в них данные, которые не умеет интерпретировать. Команды GRANT и REVOKE — это рекомендуемый способ управления привилегиями и таковыми останутся в обозримом будущем.

Если вы все же решите манипулировать таблицами привилегий вручную, в обход команд GRANT и REVOKE, то должны будете сообщить об этом MySQL, выполнив команду FLUSH PRIVILEGES, которая заново считывает и кэширует информацию об учетных записях и привилегиях. Изменения в таблицах привилегий, сделанные с помощью команды INSERT или любой другой команды общего назначения, могут остаться незамеченными до перезапуска сервера или выполнения FLUSH PRIVILEGES.

Настройка привилегий MySQL

Рассмотрим на примере, как создать учетные записи и привилегии для воображаемой организации *widgets.example.com*. Предположим, что вы вошли в только что установленный экземпляр MySQL и удалили все учетные записи, присутствующие по умолчанию, командой DROP USER. Не забудьте проверить таблицу mysql.user, чтобы убедиться, что в ней действительно ничего не осталось.

MySQL не включает роли и группы, с которыми вы, возможно, знакомы по другим СУБД. MySQL поддерживает только пользователей.

Вся работа строится на применении комбинаций трех основных команд:

```
GRANT [privileges] ON [objects] TO [user];
GRANT [privileges] ON [objects] TO [user] IDENTIFIED BY [password];
REVOKE [privileges] ON [objects] FROM [user];
```

Безопасные пароли

Для иллюстрации мы сознательно используем симпатичное словечко «p4ssword», но в условиях реальной эксплуатации такой пароль не годится. Да, MySQL хранит пароли в зашифрованном виде, но это еще не значит, что можно пренебречь сложностью пароля. Любой, кто имеет возможность подключиться к серверу MySQL, может предпринять атаку перебором, пытаясь взломать пароли, а в этой СУБД не так много хитроумных средств для обнаружения и предотвращения таких действий, как в других программных системах, например, в ОС UNIX. MySQL не дает администратору никакого способа навязать правила выбора строгих

паролей. Невозможно скомпоновать MySQL с библиотекой *libcrack* и потребовать, чтобы пароли отвечали заданным условиям, какой бы привлекательной ни казалась эта идея. Существует много хороших инструментов и сайтов, которые помогают пользователям генерировать строгие пароли, – мы рекомендуем не пренебрегать ими.

Ниже приведен обзор различных видов учетных записей, которые вам могут понадобиться, а также соответствующих им привилегий.

Учетная запись системного администратора

В большинстве крупных организаций есть две важных роли администратора. *Системные администраторы* отвечают за «физический» сервер, в том числе за операционную систему, учетные записи ОС UNIX и т. д., а в обязанности *администраторов базы данных* (АБД) входит только забота о СУБД. Как распределить административные учетные записи, дело ваше – можете упростить себе жизнь и попросить, чтобы каждый, кому нужно решать административные задачи, входил в MySQL как суперпользователь. А можете создать отдельные учетные записи для всех лиц, нуждающихся в административном доступе. Мы для начала решили остановиться на простом варианте и создать пользователя *root* со всеми привилегиями (по аналогии с традиционным именем суперпользователя в системах UNIX):

```
mysql> GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost'
    -> IDENTIFIED BY 'p4ssword' WITH GRANT OPTION;
```

Учетные записи администраторов базы данных(АБД)

Если к серверу MySQL должны иметь доступ несколько АБД, то иногда имеет смысл создать для каждого свою учетную запись, а не общую для всех. При этом упрощается учет и контроль:

```
mysql> GRANT ALL PRIVILEGES ON *.* TO 'john'@'localhost'
    -> IDENTIFIED BY 'p4ssword' WITH GRANT OPTION;
```

Учетные записи для сотрудников

Основная масса сотрудников компании *widgets.example.com* занимается обслуживанием клиентов и отвечает за ввод заявок, принимаемых по телефону, за обновление существующих заказов и т. д. Предположим, что Тера, сотрудница отдела обслуживания клиентов, входит в приложение, которое передает введенное ей имя и пароль серверу MySQL. Для создания учетной записи Теры можно ввести такую команду:

```
mysql> GRANT INSERT,UPDATE PRIVILEGES ON widgets.orders
    -> TO 'tera'@'%.widgets.example.com'
    -> IDENTIFIED BY 'p4ssword';
```

Тера должна ввести свое имя и пароль в приложение, которое затем позволит ей добавлять новые и обновлять существующие заказы. Однако удалять заказы ей запрещено. При таких условиях каждому сотруднику компании *widgets.example.com*, которому разрешено вводить заказы, необходима своя учетная запись. Тогда транзакции, выполняемые каждым сотрудником, регистрируются под его именем, и у пользователей есть только привилегии, необходимые для работы с заказами.

Имитация групп

В MySQL нет групп и ролей, как во многих других СУБД. Иногда имеет смысл создать учетную запись с именем роли приложения или роли сотрудника, например *analyst* или *custserv*, но в данном примере мы этого делать не будем.

Доступ только для протоколирования

Довольно часто MySQL применяется для протоколирования разного рода данных. Неважно, идет ли речь о регистрации всех запросов к Apache или об учете звонков в дверь, протоколирование – это приложение, которому, скорее всего, необходим только доступ для записи в одну базу данных или таблицу. Чтобы создать учетную запись протоколирования, подойдет такая команда:

```
mysql> GRANT INSERT ON logs.* TO 'logger'@'%.widgets.example.com'
    -> IDENTIFIED BY 'p4ssword';
```

Она добавляет одну строку в таблицу `user`, но, поскольку мы не задали глобальных привилегий (*.*), то все относящиеся к привилегиям столбцы в ней содержат `N`. Единственное назначение этой строки – дать возможность пользователю подключиться с любого компьютера и ввести пароль. Поскольку мы указали привилегию, связанную с конкретной базой данных, то все интересное попало в таблицу `db`, в которой все столбцы, кроме `Insert_priv`, будут содержать `N`, а последний – `Y`.

Резервное копирование

Пользователю, который снимает резервные копии с помощью программы *mysqldump*, обычно достаточно привилегий `SELECT` и `LOCK TABLES`. Если же он создает файлы с разделителями, запуская *mysqldump* с флагом *--tab* или выполняя команду `SELECT INTO OUTFILE`, то понадобится еще и привилегия `FILE`. Вот пример создания учетной записи для пользователя, который выполняет резервное копирование, подключаясь только с локального компьютера:

```
mysql> GRANT SELECT, LOCK TABLES, FILE ON *.* TO 'backup'@'localhost'
    -> IDENTIFIED BY 'p4ssword';
```

Чтобы обеспечить согласованность результатов нескольких операций резервного копирования, нужно выполнить команду `FLUSH TABLES WITH`

READ LOCK, для которой необходима также привилегия RELOAD. Она разрешает еще несколько операций, в частности FLUSH LOGS.

Эксплуатация и мониторинг

Бывает так, что необходимо предоставить кому-то или чему-то (например, пользователю или программе мониторинга в центре обслуживания сети, ЦОС) доступ к серверу MySQL для целей мониторинга или устранения неполадок. Такая учетная запись должна иметь возможность подключиться, выполнить команду KILL или SHOW и остановить сервер. Поскольку это очень широкие полномочия, необходимо ограничить подключение единственным компьютером. Следовательно, даже если неуполномоченный пользователь узнает пароль, он сможет что-то сделать, только находясь в центре обслуживания сети. Эту задачу решает такая команда:

```
mysql> GRANT PROCESS, SHUTDOWN on *.*
    -> TO 'noc'@'monitorserver.noc.widgets.example.com'
    -> IDENTIFIED BY 'p4ssword';
```

Может также понадобиться привилегия SUPER, которая позволит выполнять команду SHOW INNODB STATUS.

Изменения системы привилегий в версии MySQL 4.1

В версии MySQL 4.1 появилась новая, гораздо более безопасная схема хеширования паролей (длина свертки была увеличена с 16 байт до 41). Однако и старая схема была оставлена (даже в версии MySQL 5.0 и более поздних). Но мы не рекомендуем ей пользоваться, так как она уязвима для взлома. Если безопасность вам небезразлична, то переходите на версию MySQL 4.1 или более позднюю, и пользуйтесь только новой схемой.

В некоторых дистрибутивах GNU/Linux сервер MySQL по умолчанию сконфигурирован со старой схемой хеширования паролей в целях совместимости с устаревшими клиентами. Проверьте, присутствует ли в конфигурационном файле параметр old_passwords. Если вы хотите, чтобы сервер MySQL отвергал все попытки подключения с небезопасными паролями, то включите в конфигурационном файле сервера режим secure_auth. Аналогичный параметр существует и для клиентских программ, – он не дает им посылать пароль, хешированный по старой схеме, даже если сервер попросит. Это хорошая мысль, поскольку старый формат легко «подслушать» с помощью анализатора трафика и взломать.

Новые пароли начинаются со звездочки, так что их легко опознать визуально. В большинстве случаев учетные записи, унаследованные от старых версий MySQL, аутентифицируются нормально. Однако если клиентская программа, написанная для версии раньше 4.1, попытается соединиться с более современным сервером, на котором пароль учетной записи хранится в новом формате, то у нее ничего не получится.

Чтобы устранить эту проблему, нужно либо вернуть пароль для этой учетной записи к прежнему виду с помощью функции OLD_PASSWORD(), либо обновить клиентскую библиотеку MySQL, с которой компонуется программа.

Изменения системы привилегий в версии MySQL 5.0

В версии MySQL 5.0 добавлено несколько новых привилегий и немного изменено прежнее поведение. В настоящем разделе мы дадим общий обзор этих изменений. Прежде чем переходить на любую новую версию MySQL, всегда следует прочитать замечания к версии, где написано, что в ней появилось нового и что изменилось.

Хранимые подпрограммы

В главе 5 мы уже говорили, что в версии MySQL 5.0 появилась поддержка хранимых подпрограмм. Их можно выполнять в одном из двух контекстов безопасности: как автор (то есть пользователь, создавший подпрограмму) и как вызывающий.

Очень часто хранимые подпрограммы выступают в роли прокси-объектов, позволяющих обращаться к таблицам даже тем пользователям, которым права на доступ к ним явно не предоставлены. Обычно поступают так: заводят привилегированного пользователя, от его имени создают подпрограммы и назначают им характеристику SQL SECURITY DEFINER. В табл. 12.1 показано как хранимая процедура позволяет пользователям выполнять команды с привилегиями другого пользователя.

Таблица 12.1. Контекст безопасности для команды внутри хранимой процедуры

Пользователь, вызывающий эту процедуру	**Контекст безопасности для команд, если включена характеристика...**	
	SQL SECURITY INVOKER	SQL SECURITY DEFINER и DEFINER=*LegalStaff*
LegalStaff	*LegalStaff*	*LegalStaff*
HumanResources	*HumanResources*	*LegalStaff*
CustomerService	*CustomerService*	*LegalStaff*

При таком подходе можно разрешать или запрещать доступ к таблицам, исходя из личности пользователя, и в то же время допускать выполнение строго определенных операций с таблицами – именно тех, которые инкапсулированы в хранимой процедуре, – когда нежелательно давать пользователю прямой доступ к таблицам. Предположим к примеру, что в некоторой таблице хранятся конфиденциальные юридические данные (скажем, состояние контракта со сторонней организацией), которые разрешено просматривать только сотрудникам юридического отдела, однако сотрудники отдела обслуживания клиентов долж-

ны иметь возможность обновлять определенный столбец в этой таблице. Тогда можно отозвать привилегию SELECT на эту таблицу у всех, кроме юридического отдела, а затем написать хранимую процедуру для обновления столбца, которой разрешено пользоваться всем, и запускать ее с привилегиями пользователя *LegalStaff* (для чего достаточно включить в определении процедуры фразу SQL SECURITY DEFINER). Это напоминает привилегии SUID в UNIX-подобных операционных системах.

Пространством имен для хранимых подпрограмм является схема (база данных), поэтому наличие объектов db1.func_1() и db2.func_1() не приводит к конфликту имен.

Привилегия на выполнение подпрограммы еще не дает безусловного права на выполнение содержащихся в ней команд. Для каждой такой команды проверяются привилегии либо автора, либо вызывающего в зависимости от того, была ли при создании процедуры задана характеристика SQL SECURITY DEFINER или SQL SECURITY INVOKER.

Триггеры

В версии MySQL 5.0 появилась также поддержка триггеров, для использования которых требуются специальные привилегии, если при определении не была задана характеристика SQL SECURITY DEFINER[1]. Это может приводить к странным сообщениям об ошибках при попытке выполнить для таблицы команду UPDATE, INSERT или DELETE:

```
mysql> INSERT INTO ...;
ERROR 1142 (42000): Access denied; you need the SUPER privilege for this operation
```

Пользователь, который создает триггер без характеристики SQL SECURITY DEFINER, должен иметь привилегию SUPER, иначе при попытке выполнить триггер появится сообщение, показанное выше. (В MySQL 5.1 появилась привилегия TRIGGER, так что это сообщение перестало быть таким пугающим.)

Сервер MySQL проверяет привилегии для команд внутри триггера точно так же, как для хранимой подпрограммы.

Представления

Подобно процедурам и триггерам представления можно выполнять с привилегиями автора или вызывающего. Привилегии автора дают пользователю доступ к представлению, но не к базовым таблицам.

Это позволяет реализовать защиту на уровне строки, а также ограничить доступ к столбцам. Мы полагаем, что такое решение лучше, чем задание привилегий на уровне столбцов с помощью команды GRANT, так как его существенно проще сопровождать. Если поместить представле-

[1] В синтаксисе команды CREATE TRIGGER отсутствует характеристика SQL SECURITY DEFINER. Скорее всего, авторы имеют в виду фразу: [DEFINER = { user | CURRENT_USER }]. — *Прим. науч. ред.*

ния в отдельную базу данных, то можно просто выдать пользователям привилегии уровня базы данных и не морочить себе голову привилегиями на отдельные таблицы или представления. На рис. 12.2 показана разница между предоставлением доступа к отдельным столбцам с помощью GRANT и выборкой только нужных столбцов через представление.

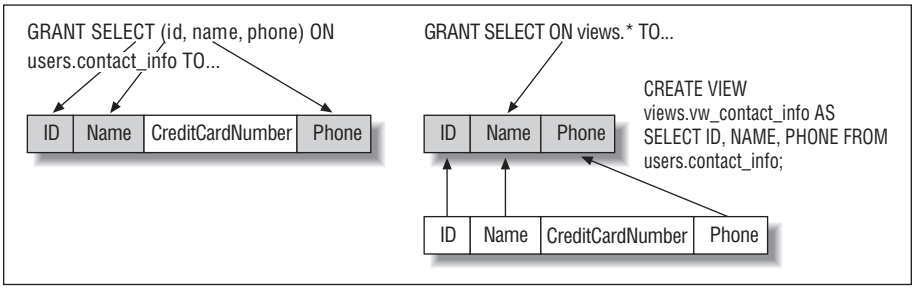


Рис. 12.2. Упрощение доступа к конкретным столбцам с помощью представления

В левой части рисунка показана команда GRANT, которая дает детальные привилегии на доступ к отдельным столбцам. Это замедляет все операции доступа к базе и требует наличия отдельных команд GRANT для каждой таблицы, в которой нужен контроль на уровне столбцов.

В правой части создается новая база данных views, в которой хранятся только представления. Затем создается представление, содержащее лишь те столбцы, к которым разрешен доступ пользователю. Все подобные представления помещаются в базу данных views, так что одной команды GRANT достаточно для разрешения доступа ко всем сразу.

Привилегии для таблиц из базы данных INFORMATION_SCHEMA

В официальных стандартах SQL определен набор представлений, известных под общим названием INFORMATION_SCHEMA. Они дают информацию о базах данных, таблицах и других компонентах СУБД. MySQL пытается максимально точно следовать стандартам. Поэтому сервер управляет привилегиями на таблицы INFORMATION_SCHEMA автоматически и лучше не определять для них никаких привилегий явно. Если пользователь без соответствующих привилегий попытается обратиться к строкам или полям такой таблицы, то MySQL не покажет строки, а вместо значений полей вернет NULL. Например, пользователь не сможет увидеть информацию о таблице в INFORMATION_SCHEMA.TABLES, если вообще не имеет никаких привилегий на эту таблицу. Точно так же ведет себя команда SHOW TABLES: MySQL не демонстрирует таблицы, на которые пользователь не имеет привилегий.

Привилегии и производительность

На первый взгляд, привилегии никак не связаны с производительностью, но в некоторых случаях это не так. Вот какие факторы следует принять во внимание:

Слишком много привилегий

Наличие очень большого количества записей в таблицах привилегий приводит к ощутимым накладным расходам. Каждая привилегия добавляет работы серверу, который должен проверить, разрешено ли пользователю выполнять запрос. Кроме того, на хранение привилегий расходуется память.

Слишком детальные привилегии

Чем ниже в иерархии (пользователь, база данных, хост, таблица, столбец) находится привилегия, тем дороже обходится ее проверка. Проверка глобальных привилегий производится сравнительно быстро, но если определена хотя бы одна привилегия на уровне столбца, то теоретически не исключено, что серверу придется сопоставлять каждый запрос с привилегиями на всех уровнях (напомним, что сервер начинает проверку с самого верхнего уровня и продолжает, пока не найдет первой подходящей привилегии).

Привилегии на доступ к столбцам и кэш запросов

Во время работы над этой книгой MySQL еще не умела использовать кэш запросов для таблиц с привилегиями на уровне столбцов. Чтобы не сталкиваться с этой и другими проблемами, присущими привилегиям уровня столбца, мы рекомендуем применять представления, как описано в предыдущем разделе.

По умолчанию во время аутентификации пользователей MySQL выполняет прямой и обратный DNS-поиск. Это можно запретить, включив в файл *my.cnf* параметр `skip_name_resolve`. Это полезно с точки зрения как безопасности, так и производительности, поскольку увеличивает скорость установки соединений, снижает зависимость от DNS-серверов и уязвимость к атакам типа «отказ от обслуживания».

Побочный эффект такого изменения состоит в том, что вы больше не можете задавать для пользователей доменное имя в столбце `Host`. Такие учетные записи перестанут работать. Вместо доменных имен следует использовать IP-адреса (но метасимволы по-прежнему разрешены, например, *192%*). Кроме того, даже при включенном режиме `skip_name_resolve` допустимо указывать специальное имя *localhost*.

Типичные проблемы и их решения

Поскольку в руководстве по MySQL тема привилегий рассматривается очень подробно, мы решили ограничиться обсуждением типичных задач, подводных камней и неожиданностей, так, чтобы этот раздел мог

служить кратким справочником или руководством по поиску и устранению неполадок. В следующих разделах описываются часто задаваемые вопросы, стандартные задачи и загадочные ситуации, с которыми нам доводилось сталкиваться.

Ошибки при подключении

Списки рассылки, форумы и IRC-каналы изобилуют сообщениями от пользователей, которым не удается подключиться к серверу MySQL. Причин может быть множество, начиная от невозможности установить TCP-соединение из-за того, что в файле *my.cnf* задан параметр `skip_networking`, и кончая ошибками при задании привилегий и вмешательством брандмауэров. Мы не можем рассмотреть здесь все причины, но в руководстве по MySQL этому вопросу посвящен специальный раздел.

Подключение по имени localhost и по адресу 127.0.0.1

Имя *localhost* обычно служит синонимом IP-адреса 127.0.0.1, но в MySQL поведение по умолчанию несколько отличается. Если одним из параметров соединения является имя *localhost*, то клиент пытается подключиться к серверу через UNIX-сокет[1], а не по протоколу TCP/IP. Таким образом, при вводе следующей команды произойдет подключение по UNIX-сокету:

```
$ mysql --host=localhost
```

Это не очень удачное проектное решение, поскольку поведение отличается от ожидаемого, однако теперь передумывать уже поздно, так как это нарушило бы совместимость со старыми приложениями и клиентскими библиотеками. Если вы хотите подключиться по протоколу TCP/IP к серверу, работающему на той же машине, что и клиент, то есть два варианта: задать вместо имени хоста IP-адрес или явно указать протокол. Любая из показанных ниже команд устанавливает соединение по протоколу TCP/IP:

```
$ mysql --host=127.0.0.1
$ mysql --host=localhost --protocol=tcp
```

Попутно отметим, что, попытавшись подключиться к переадресованному TCP-порту на машине *localhost* при настройке SSH-туннеля, вы обнаружите, что ничего не получается. Чтобы подключиться к порту, нужен протокол TCP, поэтому указывайте не имя, а адрес 127.0.0.1. Организацию SSH-туннелей мы обсудим ниже в этой главе.

[1] На платформе Windows имя *localhost* специально не обрабатывается, а подключение по именованному каналу происходит при задании имени . (точка).

Это имя хоста имеет и еще одну особенность: MySQL не пытается сопоставить имя *localhost* с метасимволом %; иными словами, задание разрешений для *user@'%'* и *user@localhost* не является избыточным.

Безопасное использование временных таблиц

MySQL не требует никаких специальных привилегий для работы с временными таблицами за исключением `CREATE TEMPORARY TABLE`. Коль скоро временная таблица создана, для нее действуют обычные привилегии уровня таблиц, заданные для пользователя. Это означает, что у пользователя может получиться создать временную таблицу, но затем не удастся добавить в нее новые столбцы, изменить структуру, построить индексы (и, быть может, даже выполнить команду `SELECT`). Однако предоставление таких привилегий могло бы позволить пользователю нанести ущерб реальным таблицам, что нежелательно.

Решение состоит в том, чтобы оставить привилегии лишь в специальной базе данных, зарезервированной для временных таблиц:

```
mysql> CREATE DATABASE temp;
mysql> GRANT SELECT, INSERT, UPDATE, DELETE, DROP, ALTER, INDEX,
    -> CREATE TEMPORARY TABLES ON temp.* TO analyst@'%';
```

Запрет доступа без пароля

MySQL разрешает вход без пароля. Учетной записи без пароля соответствует такая строка в таблице `user`, в которой столбец `password` содержит пустую строку. Создать такую запись можно командой `GRANT`, в которой отсутствует фраза `IDENTIFIED BY`.

Невозможно полностью запретить доступ без пароля, но если вы контролируете компьютеры, с которых могут подключаться пользователи, то можете добавить такую запись в раздел `[client]` файла *my.cnf*:

```
password
```

В результате все программы, которые по умолчанию читают этот файл (а к ним относятся, в частности, все приложения, входящие в дистрибутив MySQL, если не настроить их специальным образом), будут обязательно запрашивать пароль пользователя. В версии MySQL 5.0 и старше можно установить на сервере режим `NO_AUTO_CREATE_USER`, который запрещает в команде `GRANT` создавать учетные записи без пароля, но решительно настроенный пользователь может обойти это ограничение.

Напомним, что пользователь, для которого в таблице `mysql.user` вместо пароля хранится пустая строка, считается пользователем без пароля, а не пользователем с пустым паролем.

Запрет анонимных пользователей

MySQL допускает также наличие анонимных пользователей: любая запись в таблицах привилегий, для которой в столбце `User` находится пустая строка, определяет привилегии анонимного пользователя. Но будь-

те осторожны: команда SHOW GRANTS не показывает такие привилегии. Мы полагаем, что подобные записи лучше удалить, для этого можно воспользоваться входящей в дистрибутив MySQL программой *mysql_secure_installation*.

Не забывайте отдельно заключать в кавычки имена хостов

Легко позабыть о том, что имена пользователей и хостов следует заключать в кавычки по отдельности. Следующая команда делает не то, что ожидается:

```
mysql> GRANT USAGE ON *.* TO 'fred@%';
```

На первый взгляд, здесь создается учетная запись для пользователя *fred*, который может подключаться из любого места, но на самом деле будет создан пользователь с именем *fred@%*. Правильный синтаксис таков (обратите внимание, что и имя пользователя, и имя хоста заключены в кавычки):

```
mysql> GRANT USAGE ON *.* TO 'fred'@'%';
```

Не используйте имена пользователей повторно

MySQL рассматривает пользователей с одинаковыми именами, но разными хостами, как совершенно различные объекты. Может показаться, что удобно давать одному пользователю разные привилегии в зависимости от того, откуда он подключается, но, по нашему опыту, это редко бывает удачной идеей. Потенциальная путаница намного перевешивает выгоды. Гораздо проще считать, что имена пользователей уникальны, и задействовать столбец Host, чтобы ограничить места, *из которых пользователь может подключаться*, а не *действия, которые разрешены ему после подключения*. Например, можно разрешить подключение только с локальной машины, только из локальной сети или только из конкретной подсети. Это разумная мера предосторожности, но гораздо безопаснее применять для ограничения попыток соединения брандмауэр (мы еще поговорим об этом).

Тот факт, что MySQL обладает большой гибкостью, еще не означает, что всем этим богатством надо пользоваться. Мы полагаем, что чем проще, тем лучше.

Привилегия SELECT разрешает также SHOW CREATE TABLE

Наличие у пользователя привилегии SELECT позволяет ему выполнять команду SHOW CREATE TABLE, которая выводит SQL-команду, необходимую для воссоздания таблицы. Обычно в этом нет ничего плохого, но иногда таким образом раскрываются секретные детали. Самый очевидный случай – таблицы типа Federated в версии MySQL 5.0: SHOW CREATE TABLE покажет имя и пароль пользователя, с которыми подсистема хранения подключается к удаленному серверу. (В MySQL 5.1 добавлен специаль-

ный механизм управления удаленными соединениями для таблиц типа Federated.)

Не давайте привилегий для базы данных mysql

Если вы дадите пользователю привилегии для доступа к базе данных `mysql`, то он сможет изменить собственные привилегии, просмотреть привилегии других пользователей (что широко распахивает дверь для атаки путем подбора пароля) и даже переименовать или изменить таблицы, необходимые MySQL для работы. Обычному пользователю нет нужды обращаться *ни к одной* из этих таблиц – даже для чтения. Следовательно, такую команду выполнять не стоит, поскольку она дает все привилегии глобально:

```
mysql> GRANT ... ON *.* ...;
```

Если у пользователя есть разрешение модифицировать таблицы в базе данных `mysql`, значит, он должен иметь и привилегию `GRANT`. В противном случае он сможет удалить какие-то привилегии посредством стирания строк, но не сумеет вернуть их назад. Один из авторов этой книги однажды по оплошности удалил таким образом всех пользователей, и вынужден был останавливать сервер MySQL и восстанавливать пользователей, запустив сервер с флагом *--skip_grant_tables*.

Не раздавайте бездумно привилегию SUPER

Привилегия `SUPER` разрешает выполнять операции, зарезервированные для суперпользователя (например, изменять данные на сервере, сконфигурированном только для чтения). Это понятно, но есть и еще один аспект: MySQL резервирует для пользователя с привилегией `SUPER` одно соединение даже в том случае, когда достигнут лимит `max_connections`. Это позволяет войти на сервер и заняться администрированием в ситуации, когда запросы на соединение от обычных клиентов уже не принимаются.

Лучше не выдавать привилегию `SUPER` слишком широкому кругу пользователей, но иногда это затруднительно, поскольку она необходима для ряда стандартных задач (например, удаления журналов главного сервера).

Выдача привилегий на несколько баз данных

Механизм сопоставления с шаблоном, принятый в MySQL, не позволяет сказать «все базы данных, кроме этих». Подразумевается, что давать привилегии на все базы, кроме `mysql`, довольно утомительно. Поможет продуманная схема именования: включайте в имена всех баз общий префикс и при выдаче привилегий указывайте имя с метасимволами, например:

```
mysql> GRANT ... ON `analysis%`.* TO 'analyst' ...;
```

К сожалению, в MySQL нет настоящих схем, которые помогли бы решить эту проблему в корне. Впрочем, соглашения об именовании както смягчают ее остроту. Отметим, что имя базы данных в команде GRANT нужно заключать в обратные кавычки (как идентификатор).

Эта техника полезна и при настройке разделяемого хостинга. Обычно в указанном случае пользователю разрешен доступ только к базам данных, имена которых начинаются с его имени и символа подчеркивания. Так как символ подчеркивания – это метасимвол, то в команде GRANT его следует экранировать. Например, при создании учетной записи для пользователя *sunny* можно было бы выполнить такую команду:

```
mysql> GRANT ... ON `sunny\_%`.* TO 'sunny' ...;
```

Вероятно, не стоит выдавать привилегию SHOW DATABASES в системах разделяемого хостинга. Тогда пользователи даже не увидят баз данных, к которым не имеют доступа, а чем меньше они знают, тем лучше.

Отзыв привилегий

Если привилегия выдана глобально, то отозвать ее по-другому невозможно:

```
mysql> GRANT SELECT ON *.* TO 'user';
mysql> SHOW GRANTS FOR user;
+----------------------------------+
| Grants for user@%                |
+----------------------------------+
| GRANT SELECT ON *.* TO 'user'@'%' |
+----------------------------------+
mysql> REVOKE SELECT ON sakila.film FROM user;
ERROR 1147 (42000): There is no such grant defined for user
'user' on host '%' on table 'film'
```

Поскольку привилегия была выдана глобально, то и отозвать ее можно только глобально; попытка отозвать привилегию доступа к конкретной таблице приводит к сообщению о том, что не существует привилегий табличного уровня, удовлетворяющих заданному критерию.

Пользователь может подключиться даже после отзыва привилегий

Предположим, что вы отозвали у пользователя все привилегии:

```
mysql> REVOKE ALL PRIVILEGES ON...;
```

Но подключиться к серверу он все равно сможет, потому что команда REVOKE удаляет не учетную запись, а только предоставленные ей привилегии. Чтобы окончательно удалить учетную запись, нужно выполнить команду DROP USER (или в старых версиях MySQL удалить строку из таблицы mysql.user).

Даже после отзыва всех привилегий команда SHOW GRANTS показывает, что у пользователя осталась привилегия USAGE. Ее отозвать невозможно, поскольку это синоним «нет привилегий». Она просто означает, что пользователь может подключиться к серверу MySQL.

Когда невозможно предоставить или отозвать привилегию

Помимо права предоставлять привилегии (GRANT) вы должны сами обладать привилегией, которую пытаетесь предоставить или отозвать. Эта мера призвана предотвратить взаимное расширение привилегий. Чтобы отозвать все имеющиеся привилегии (ALL PRIVILEGES), необходимо обладать привилегией CREATE USER.

Невидимые привилегии

Команда SHOW GRANTS показывает не все привилегии, имеющиеся у пользователя, а лишь те, которые были предоставлены ему явно. Пользователь может обладать и другими привилегиями, например теми, что были выданы анонимным пользователям. Например, только что установленный экземпляр MySQL по умолчанию настроен так, что у всех пользователей есть привилегии на базу данных test и на базы данных с именами, начинающимися с test! Убедимся в этом на примере. Сначала зайдем от имени *root* и создадим пользователя без каких бы то ни было привилегий:

```
mysql> GRANT USAGE ON *.* TO 'restricted'@'%' IDENTIFIED BY 'p4ssword';
mysql> SHOW GRANTS FOR restricted;
+-----------------------------------------------------+
| Grants for restricted@%                             |
+-----------------------------------------------------+
| GRANT USAGE ON *.* TO 'restricted'@'%' IDENTIFIED BY |
| PASSWORD '*544F2E9C6390E7D5A5E0A508679188BBF7467B57' |
+-----------------------------------------------------+
```

Вроде бы все нормально; этот пользователь сможет подключиться, но и только. Однако это всего лишь видимость. Давайте зайдем от имени этого пользователя и выполним команду SHOW DATABASES:

```
$ mysql -u restricted -pp4ssword
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| test               |
+--------------------+
```

На этом сервере имеется также демонстрационная база данных Sakila, которая не показана, так как пользователь не обладает привилегией SHOW DATABASES, однако база данных test в списке есть. Более того, как

легко убедиться, новый пользователь обладает привилегиями на доступ к этой базе и всем таблицам в ней:

```
mysql> USE test;
mysql> SHOW TABLES;
+---------------+
| Tables_in_test |
+---------------+
| heartbeat      |
+---------------+
mysql> SELECT * FROM heartbeat;
+----+---------------------+
| id | ts                  |
+----+---------------------+
|  1 | 2007-10-28 21:31:08 |
+----+---------------------+
```

И он может не только читать из таблиц, но имеет и большинство прочих привилегий. Даже в состоянии создать новую базу данных:

```
mysql> CREATE DATABASE test_muah_ha_ha;
Query OK, 1 row affected (0.01 sec)
```

Причиной такого поведения являются следующие две строки в таблице `mysql.db`:

```
mysql> SELECT * FROM mysql.db\G
*************************** 1. row ***************************
       Host: %
         Db: test
       User:
Select_priv: Y
... опущено ...
*************************** 2. row ***************************
       Host: %
         Db: test\_%
       User:
Select_priv: Y
... опущено ...
```

Обратите внимание, что столбец `User` в обеих строках пуст, а это означает, что анонимные пользователи, – а по сути все пользователи – обладают указанными привилегиями, даже если команда `SHOW GRANTS`[1] их не показывает. Мораль в том, что `SHOW GRANTS` не всегда сообщает всю информацию. Иногда все же нужно знать, как интерпретировать содержимое таблиц привилегий.

[1] В этом примере иллюстрируется одно из «магических» значений в таблицах привилегий MySQL. Пустая строка в столбце `User` относится к анонимному пользователю. Именно так сервер аутентифицирует пользователя, зашедшего с несуществующим именем, а также обозначает привилегии, предоставляемые всем пользователям.

Но это не единственная странность в поведении привилегий пользователя. Поскольку алгоритм сопоставления с базой данных и именем хоста сначала выбирает наиболее специфичное соответствие, то предоставленные привилегии с менее специфичными условиями оказываются скрыты, даже если дают больше прав.

Рассмотрим такую ситуацию: вместо того чтобы придерживаться предложенного выше соглашения об именовании и соблюдать принцип наименьших привилегий, ленивый администратор решает действовать прямо противоположно. Он выдает пользователю все привилегии, а затем отменяет нежелательные, скрывая привилегии в базе данных `mysql` с помощью строки, в которой все столбцы, относящиеся к привилегиям, содержат `N`:

```
mysql> GRANT USAGE ON *.* TO 'gotcha'@'%' IDENTIFIED BY 'p4ssword';
mysql> GRANT ALL PRIVILEGES ON `%`.* TO 'gotcha'@'%';
mysql> INSERT INTO mysql.db(Host, DB, User) VALUES('%', 'mysql', 'gotcha');
mysql> FLUSH PRIVILEGES;
```

Поскольку образец `mysql` более специфичен, чем `%`, то ленивый администратор заключает, что у пользователя не будет, в частности, привилегии `SELECT` в базе данных `mysql`. И это действительно так:

```
mysql> SELECT * FROM mysql.user;
ERROR 1142 (42000): SELECT command denied to user
'gotcha'@'localhost' for table 'user'
```

Проблема в том, что при такой схеме назначения привилегий администратор расставляет ловушку любому, кто в будущем попытается изменить привилегии данного пользователя. Очень легко по ошибке удалить привилегии, относящиеся к базе `mysql`, и тогда все ранее скрытые привилегии в этой базе станут действующими. Иными словами, удаление привилегии на самом деле предоставляет больше привилегий, чем было! Такая схема не так хитроумна, как может показаться, более того, она просто опасна. К тому же команда `SHOW GRANTS` не показывает скрытые привилегии, поэтому их легко не заметить.

Подобные игры проходят и для сопоставления с именем хоста – с такими же последствиями. Например, чтобы дать пользователю *gotcha* возможность подключаться со всех компьютеров, кроме одного, нельзя задать «исключающий образец имени хоста», поскольку ничего такого MySQL не поддерживает. Единственный способ – создать пользователя с таким же именем, но указать для него запрещенное имя хоста и фиктивный пароль:

```
mysql> GRANT USAGE ON *.* TO 'gotcha'@'denied.com' IDENTIFIED BY 'bOgus';
```

Когда пользователь *gotcha* попытается зайти с этого хоста, MySQL отыщет в таблице `user` строку, содержащую значение `gotcha@denied.com`, и откажет в доступе, так как пароли не совпадают. Однако такое «решение» таит в себе серьезную опасность. Кто-нибудь может подумать, что

эта запись попала в таблицу по ошибке и удалит ее или в борьбе за производительность отключит разрешение доменных имен, или сам пользователь подделает результаты обратного DNS-поиска. В любом случае пользователь сможет подключиться как *gotcha@'%'*.

Мы рекомендуем избегать скрытых привилегий и «хитроумных трюков» наподобие только что рассмотренных. Лучше руководствоваться здравым смыслом и не пытаться «мухлевать» с привилегиями, если в том нет острой необходимости. И вообще следуйте проверенному жизнью принципу наименьших привилегий.

Неактуальные привилегии

MySQL не удаляет привилегии при удалении объектов. Пусть, например, выполнена такая команда:

```
mysql> GRANT ALL PRIVILEGES ON my_db.* TO analyst;
```

А вслед за ней такая:

```
$ mysqladmin drop my_db
```

Было бы неплохо, если бы MySQL аннулировала привилегии, предоставленные командой `GRANT`, но на самом деле они остаются в таблице `db`. Если впоследствии будет вновь создана база данных с таким же именем, то привилегии на нее уже будут существовать, что может стать причиной неприятностей, поскольку администратор, возможно, уже и не помнит, что учетная запись *analyst* когда-то обладала какими-то привилегиями.

Начиная с версии MySQL 5.0 таблицы из схемы `INFORMATION_SCHEMA` способны помочь в отыскании неактуальных привилегий. Например, следующий запрос с внешним соединением находит привилегии, ссылающиеся на несуществующие базы данных:

```
mysql> SELECT d.Host, d.Db, d.User
    -> FROM mysql.db AS d
    -> LEFT OUTER JOIN INFORMATION_SCHEMA.SCHEMATA AS s
    -> ON s.SCHEMA_NAME LIKE d.Db
    -> WHERE s.SCHEMA_NAME IS NULL;
+------+---------+------+
| Host | Db      | User |
+------+---------+------+
| %    | test\_% |      |
+------+---------+------+
```

Аналогичные запросы можно написать и для других таблиц в схеме `INFORMATION_SCHEMA`. В более ранних версиях MySQL неактуальные привилегии приходилось либо искать вручную, либо писать для этого специальный сценарий.

MySQL позволяет создавать привилегии уровня базы данных для несуществующих баз, но не даст создать привилегии табличного уровня для

несуществующей таблицы. Чтобы сделать это, нужно будет вставлять строки непосредственно в таблицу `mysql.tables_priv`.

Безопасность на уровне операционной системы

Даже самые продуманные и безопасные таблицы привилегий окажутся бесполезны, если противнику удастся получить доступ к серверу от имени *root*. Имея неограниченные права, такой пользователь сможет просто скопировать все файлы данных на другую машину с работающим MySQL[1]. Тем самым противник просто получит копию вашей базы данных.

Однако кража данных – не единственная угроза, от которой нужно защищаться. Изобретательный противник может решить, что гораздо интереснее внести незаметные изменения в данные за несколько недель или даже месяцев. В зависимости от того, как долго вы храните резервные копии и сколько времени прошло с момента обнаружения злонамеренных изменений, последствия такой атаки могут быть весьма плачевными.

Общие рекомендации

Рекомендации, приведенные в настоящем разделе, нельзя считать исчерпывающим руководством по защите системы. Если вы серьезно относитесь к безопасности – а так и должно быть, – то мы рекомендуем прочитать книгу Simson Garfinkel и др. «Practical UNIX and Internet Security» (издательство O'Reilly). Тем не менее, следующие мысли могут помочь обеспечению надежной зашиты серверов баз данных.

Не запускайте MySQL от имени привилегированной учетной записи

Пользователь *root* в UNIX и System (Administrator в Windows) наделены полным контролем над системой. Если кто-нибудь обнаружит в MySQL уязвимость, а СУБД работает от имени привилегированного пользователя, то противник сможет получить широкие права на сервере. Инструкция по установке говорит об этом вполне определенно, но мы еще раз повторим: создайте отдельную учетную запись (обычно *mysql*) исключительно для запуска сервера MySQL.

Регулярно обновляйте операционную систему

Все поставщики операционных систем (Microsoft, Sun, Red Hat, Novell и др.) уведомляют о выходе обновлений, касающихся безопасности. Подпишитесь на список рассылки соответствующего поставщика. Обращайте особое внимание на списки рассылки по безопас-

[1] Напомним, что файлы MyISAM переносимы между операционными системами и аппаратными архитектурами (при условии, что формат чисел с плавающей точкой одинаков).

ности самой СУБД, а также всех продуктов, которые могут напрямую взаимодействовать с базой данных, например PHP или Perl.

Ограничивайте количество учетных записей на промышленном сервере

Нужна ли каждому разработчику приложений на основе MySQL учетная запись на сервере? Конечно, нет, это необходимо только администраторам – системным и базы данных. Разработчикам нужна только возможность удаленно отправлять серверу запросы по протоколу TCP/IP.

Отделяйте промышленный сервер от всего остального

Разделяйте промышленное окружение от среды разработки и тестирования. Лучше использовать для этих целей разные физические серверы. К безопасности промышленного сервера предъявляются совершенно другие требования, чем к серверу разработки, поэтому разумно физически разделить их. Заодно это поможет избежать многих ошибок и упростит администрирование и обслуживание. Необходимо с самого начала позаботиться о надлежащих процедурах и инструментах, например, о том, как будут переноситься данные между серверами.

Подвергайте сервер аудиту

Многие крупные организации содержат штат внутренних аудиторов, которые могут оценить защищенность сервера и дать рекомендации по ее улучшению. Если вы не располагаете такой возможностью, то можете хотя бы нанять консультанта по безопасности для выполнения аудита.

Применяйте самые строгие существующие средства защиты

Чтобы максимально изолировать сервер MySQL, можно воспользоваться такими методами, как изменение начальной точки файловой системы (chrooting), организация «тюрем» (jails), зон или виртуальных серверов.

Еще одной важной мерой защиты является хранение резервных копий на другом сервере. Если противник взломает сервер, вы должны будете восстановить операционную систему из нескомпрометированного источника. После этого встанет вопрос о восстановлении всех данных. При наличии времени можно даже сравнить скомпрометированный сервер с заведомо хорошей резервной копией, чтобы понять, как противник получил доступ.

Безопасность на уровне сети

Лучше всего изолировать серверы, запретив к ним всякий доступ из сети, но иногда к серверу MySQL должны обращаться клиенты, работающие на других компьютерах. Мы рассмотрим несколько способов ограничить доступ к серверу в таких условиях.

Даже если сервер используется только во внутренней сети организации, нужно принять все меры, чтобы оградить данные от любопытных глаз. Никогда не следует забывать, что серьезные угрозы безопасности чаще всего исходят от самих сотрудников компании.

Имейте в виду, что изложенные ниже рекомендации – лишь отправная точка в процессе обеспечения надежной защиты серверов MySQL. Есть немало хороших книг по сетевой безопасности, например Elizabeth D. Zwicky и др. «Building Internet Firewalls»[1] и Craig Hunt «TCP/IP Network Administration»[2] (обе вышли в издательстве O'Reilly). Если вы по-настоящему озабочены безопасностью сети, сделайте одолжение – возьмите какую-нибудь книгу на эту тему (но сначала закончите читать эту!).

Как и в случае безопасности на уровне операционной системы, очень полезно нанять стороннего консультанта для проведения аудита сети – пусть лучше он обнаружит слабые места, чем кто-то начнет их эксплуатировать.

Разрешение только локальных подключений

Если MySQL используется только в приложении, которое работает на том же самом компьютере (так часто бывает в случае небольших и средних по размеру веб-сайтов), то может статься, что доступ к серверу по сети можно вообще запретить. А раз запросы на соединение с внешних компьютеров не принимаются, то у противника меньше шансов добраться до вашего сервера.

Запрет доступа по сети ограничивает возможность удаленно производить административные действия (добавлять пользователей, ротировать журналы и т. д.), поэтому придется либо заходить на сервер по протоколу SSH, либо устанавливать специальное веб-приложение. В некоторых системах на платформе Windows вход по сети может оказаться затруднительным, но существуют и другие способы удаленного доступа. Например, можно установить приложение *phpMyAdmin*. Однако имейте в виду, что в нем самом не раз находили уязвимости!

Конфигурационный параметр `skip_networking` говорит MySQL, что принимать соединения по TCP-сокету не следует, но соединения по UNIX-сокету разрешены. Запустить MySQL без поддержки сети просто. Достаточно поместить следующую строку в раздел `[mysqld]` файла *my.cnf*:

```
[mysqld]
skip_networking
```

[1] Цвики Э., Купер С., Чапмен Б. «Создание защиты в Интернете», 2-е изд. – Пер. с англ. – СПб.: Символ-Плюс, 2001.

[2] Хант К. «TCP/IP. Сетевое администрирование», 3-е изд. – Пер. с англ. – СПб.: Символ-Плюс, 2004.

Но у параметра `skip_networking` есть неприятный побочный эффект: он не дает использовать репликации и такие инструменты, как *stunnel*, для организации безопасной связи с другими серверами, и к тому же полностью блокирует подключение приложений, написанных на Java (адаптер Connector/J умеет подключаться только по протоколу TCP/IP). Альтернативно можно сконфигурировать MySQL следующим образом:

```
[mysqld]
bind_address=127.0.0.1
```

В этом случае соединения по протоколу TCP/IP разрешены, но только с локальной машины, так что обеспечивается и безопасность, и удобство. В некоторых популярных дистрибутивах GNU/Linux эта конфигурация принимается по умолчанию.

Интересная конфигурация получается, когда для подчиненного сервера установлен режим `skip_networking`. Поскольку именно подчиненный сервер инициирует соединение с главным, он по-прежнему может получать все обновления. Но так как подключения к нему по протоколу TCP запрещены, то «резервная реплика» оказывается лучше защищенной и не может быть скомпрометирована удаленно. Однако использовать такой подчиненный сервер для аварийного переключения при отказе невозможно, потому что к нему нельзя подключиться удаленно.

Брандмауэры

Как и для любой сетевой службы, очень важно, чтобы соединения были разрешены только с авторизованных компьютеров. Конечно, для ограничения множества машин, с которых разрешено подключаться пользователям, можно воспользоваться командой `GRANT`, но всегда полезно организовывать многоуровневую защиту. При наличии нескольких способов фильтрации соединений одна ошибка, например опечатка в команде `GRANT`, не даст подключиться с неавторизованного компьютера. Применение брандмауэра для фильтрации соединений дает дополнительный уровень защиты[1].

Во многих организациях сетевой безопасностью занимаются не те же люди, которые разрабатывают приложения. Это снижает риск компрометации сервера из-за изменения, внесенного одним человеком.

При настройке брандмауэра безопаснее всего начать с полного запрета всех соединений. Затем постепенно добавляются правила, открывающие доступ к тем службам, которые необходимы другим компьютерам. Если система предоставляет только доступ к серверу MySQL, то

[1] В этой книге можно считать, что брандмауэр представляет собой устройство, которое пропускает через себя весь сетевой трафик с целью фильтрации и, возможно, маршрутизации. Будет ли это «настоящий» брандмауэр, маршрутизатор или старенький ПК на базе процессора Intel 486, не играет роли.

следует разрешить соединения с TCP-портом 3306 (зарезервирован для MySQL по умолчанию) и, быть может, удаленный вход по протоколу SSH (обычно TCP-порт 22).

Отключение маршрута по умолчанию

Подумайте о том, чтобы не конфигурировать маршрут по умолчанию на серверах MySQL, защищенных брандмауэром. Тогда, даже если брандмауэр будет скомпрометирован и кто-то попытается подключиться к серверу MySQL извне, пакеты не будут доставлены обратно отправителю. Они не смогут выйти из локальной сети.

Предположим, что сервер MySQL имеет IP-адрес 192.168.0.10, а маска локальной подсети задана в виде 255.255.255.0. В этом случае любой пакет из сети 192.168.0.0/24 считается «локальным», поскольку он может быть передан по физически подключенному сетевому интерфейсу (обычно *eth0* или его эквиваленту в зависимости от операционной системы). Трафик с любого другого адреса необходимо сначала доставить в шлюз для дальнейшей маршрутизации к пункту назначения, но, поскольку маршрута по умолчанию не существует, то найти шлюз невозможно, а, значит, пакет никогда не дойдет до адресата.

Если необходимо, чтобы к серверу, находящемуся за брандмауэром, могли обращаться некоторые внешние компьютеры, добавьте для них статические маршруты. Тогда сервер будет отвечать минимально возможному количеству внешних хостов.

Отключение маршрута по умолчанию – не стопроцентно надежная методика, и защищает она скорее от ошибок конфигурирования брандмауэра, нежели от полной компрометации. Но даже малостью пренебрегать не следует.

MySQL в демилитаризованной зоне

Одной лишь защиты серверов MySQL брандмауэром часто бывает недостаточно. Если какой-нибудь веб-сервер или сервер приложений скомпрометирован, то противник может прямо с него атаковать сервер MySQL. После того как противник получит доступ хотя бы к одному узлу сети за брандмауэром, все остальные узлы этой сети также становятся ему доступны с относительно малым количеством ограничений[1].

Повысить безопасность можно за счет переноса серверов MySQL в отдельный сегмент сети, недоступный извне. Пусть, например, в локальной сети имеется веб-сервер или другие серверы приложений, а также брандмауэр. За брандмауэром, в отдельном физическом сегменте

[1] Впрочем, это не совсем так. Многие современные сетевые коммутаторы позволяют конфигурировать несколько виртуальных локальных сетей (VLAN) в одной физической. Узлы, находящиеся в разных виртуальных сетях, не могут общаться друг с другом напрямую.

и в другой логической подсети находятся один или несколько серверов MySQL. Серверам приложений разрешен ограниченный доступ к серверам MySQL: весь трафик с них сначала проходит через брандмауэр, на котором можно задать очень строгие ограничительные правила. Если противник получит доступ к серверу приложений, которому разрешен только трафик на порт 3306 на серверах MySQL, то организовать атаку на другие службы, работающие на тех же серверах (например, SSH), у него не выйдет.

Можно даже поместить серверы приложений либо в общую демилитаризованную зону (ДМЗ), либо в отдельную ДМЗ, предназначенную только для них. Или это уже чересчур? Возможно. В вопросах безопасности всегда так: приходится искать компромисс между степенью защищенности и удобством, но нужно четко представлять себе возможные риски.

Шифрование и туннелирование соединений

Если с сервером MySQL приходится взаимодействовать по сети общего доступа (например, Интернет) или иной сети, открытой для прослушивания трафика (таковыми являются многие беспроводные сети), то следует подумать о применении той или иной формы шифрования. В этом случае противнику, перехватившему соединение, будет гораздо сложнее подсмотреть или подделать данные.

К тому же многие алгоритмы шифрования еще и сжимают информацию. Так что и данные оказываются лучше защищены, и пропускная способность сети используется более эффективно.

Хотя наше обсуждение посвящено клиентам, обращающимся к серверу MySQL, надо понимать, что в роли клиента может выступать другой MySQL-сервер. Такая ситуация типична при использовании репликации: подчиненный сервер подключается к главному по тому же протоколу, что и обычные клиенты.

Виртуальные частные сети

Если у компании есть несколько отделений в разных местах, то между ними можно организовать виртуальную частную сеть (VPN). Существует достаточно много технологий, позволяющих это сделать. Как правило, в каждом отделении устанавливают внешние маршрутизаторы, которые шифруют весь трафик, предназначенный другим отделениям. В такой ситуации волноваться особо не о чем. Трафик передается по открытым или частным сетям, расположенным между отделениями, уже в зашифрованном виде.

Верно ли, что VPN делает излишним применение решений, ориентированных исключительно на MySQL? Не обязательно. Если VPN-сеть по какой-то причине должна быть отключена, то хорошо бы, чтобы трафик MySQL оставался засекреченным. Для решения проблемы можно

сконфигурировать MySQL так, чтобы разрешались соединения только с IP-адресов, принадлежащих VPN-сети; тогда при отключении VPN сервера MySQL будет недоступен.

SSL в составе MySQL

Начиная с версии 4.1, MySQL поддерживает протокол Secure Sockets Layer (SSL). Это та самая технология, которая позволяет держать в секрете номер кредитной карточки при покупке книг в интернет-магазине Amazon.com или авиабилетов на вашем любимом туристическом сайте. Конкретно, в MySQL используется бесплатная библиотека yaSSL (в более ранних версиях – OpenSSL).

В некоторых двоичных дистрибутивах MySQL поддержка SSL по умолчанию отключена. Чтобы понять, как обстоит дело с вашим сервером, проверьте значение переменной `have_openssl`:

```
mysql> SHOW VARIABLES LIKE 'have_openssl';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| have_openssl  | NO    |
+---------------+-------+
```

Если оно равно `NO`, то придется либо откомпилировать MySQL самостоятельно, либо найти другой дистрибутив. Если же эта переменная равна `YES`, то перед администратором открывается целый ряд новых возможностей по защите доступа к данным. Как ими воспользоваться, зависит от требований, предъявляемых к безопасности конкретным приложением.

Самое простое – разрешить только зашифрованные сеансы, полагаясь на то, что SSL обеспечит защиту пароля пользователя. Можно потребовать, чтобы пользователь подключался только по протоколу SSL, задав дополнительные аргументы в команде `GRANT`:

```
mysql> GRANT ... IDENTIFIED BY 'p4ssword' REQUIRE SSL;
```

Однако команда `GRANT` не налагает никаких ограничений на SSL-сертификат, которым пользуется подключающийся клиент. Коль скоро клиент и сервер MySQL сумели согласовать параметры SSL-сеанса, MySQL не станет проверять достоверность клиентского сертификата.

Можно потребовать, чтобы выполнялась минимальная проверка клиентского сертификата, задав параметр `REQUIRE x509`:

```
mysql> GRANT ... IDENTIFIED BY 'p4ssword' REQUIRE x509;
```

Тогда хотя бы будет проверено, что клиентский сертификат находится среди тех, которые были прописаны при настройке MySQL.

Следующий шаг в том же направлении – разрешить доступ к базе данных только конкретному клиентскому сертификату. Для этого предназначен параметр `REQUIRE SUBJECT`:

```
mysql> GRANT ... IDENTIFIED BY 'p4ssword'
    -> REQUIRE SUBJECT "/C=US/ST=New York/L=Albany/O=Widgets
Inc./CN=client-ray.example.com/emailAddress=raymond@example.com";
```

Быть может, вас интересует не столько сам клиентский сертификат, сколько то, что он выпущен удостоверяющим центром (УЦ) вашей организации. На такой случай предусмотрен следующий синтаксис:

```
mysql> GRANT ... IDENTIFIED BY 'p4ssword'
    -> REQUIRE ISSUER "/C=US/ST=New+20York/L=Albany/O=Widgets
Inc./CN=cacert.example.com/emailAddress=admin@example.com";
```

И для достижения самой надежной аутентификации можно объединить оба параметра, задав конкретные значения `ISSUER` и `SUBJECT`. Например, можно потребовать, чтобы пользователь Raymond обладал конкретным сертификатом, выпущенным вашим УЦ:

```
mysql> GRANT ... IDENTIFIED BY 'p4ssword'
    -> REQUIRE SUBJECT "/C=US/ST=New York/L=Albany/O=Widgets
Inc./CN=client-ray.example.com/emailAddress=raymond@example.com"
    -> AND ISSUER "/C=US/ST=New+20York/L=Albany/O=Widgets
Inc./CN=cacert.example.com/ emailAddress=admin@example.com";
```

Еще один, не столь важный, аспект поддержки SSL – требование `CIPHER`, с помощью которого администратор может разрешить только «достойные доверия» (стойкие) шифры. Сам протокол SSL не зависит от алгоритма шифрования, поэтому потенциально стойкое шифрование в SSL может быть сведено на нет выбором слабого шифра для защиты передаваемых данных. Чтобы ограничить спектр возможных шифров теми, которые вы считаете безопасными, выполните такую команду:

```
mysql> GRANT ... IDENTIFIED BY 'p4ssword'
    -> REQUIRE CIPHER "EDH-RSA-DES-CBC3-SHA";
```

Управление индивидуальными клиентскими сертификатами на первый взгляд представляется отличной мерой повышения безопасности, но на деле может обернуться административным кошмаром. При создании клиентского сертификата ему нужно назначить дату истечения срока действия – и желательно в не слишком отдаленном будущем. Рекомендуется, чтобы сертификат, с одной стороны, действовал достаточно долго, и не приходилось постоянно выпускать новые, а, с другой стороны, не очень долго, чтобы, попав в руки противника, он не давал доступ к данным в течение длительного времени.

В небольшой организации всего с парой сотрудников следить за индивидуальными сертификатами просто. Но когда в организации сотни или тысячи работников и у каждого свой сертификат, отслеживать сроки их действия и вовремя заменять их новыми – весьма сложная задача.

В некоторых организациях эта проблема решается комбинированием параметра `REQUIRE ISSUER` с ежемесячным генерированием клиентских

сертификатов, которые распространяются по защищенным каналам, например через внутреннюю сеть компании. Клиент может самостоятельно скачать сертификат и подключаться с его помощью к серверу MySQL в течение одного-двух месяцев. В этом случае, если сотрудник потеряет возможность обращения к внутренней сети компании или у партнера не будет доступа к его месячному ключу, то даже при том, что администратору не прикажут закрыть доступ данному лицу, такой работник все равно потеряет возможность подключения к серверу через заранее известное время естественным образом.

SSH-туннель

Если эксплуатируется старая версия MySQL или вы просто не хотите возиться с настройкой SSL, то можно обратиться к протоколу SSH. При работе на платформе UNIX или Linux вы, скорее всего, уже пользуетесь программой SSH для входа на удаленную машину[1]. Но большинству людей неизвестно, что эта программа позволяет организовать зашифрованный туннель между двумя компьютерами.

SSH-туннелирование лучше всего проиллюстрировать на примере. Предположим, что нужно создать зашифрованное соединение между рабочей станцией под управлением GNU/Linux и сервером MySQL, который работает на машине *db.example.com*. На рабочей станции запускаем следующую команду:[2]

```
$ ssh -N -f -L 4406:db.example.com:3306
```

Она организует туннель между TCP-портом 4406 на рабочей станции и портом 3306 на машине *db.example.com*. Теперь по этому туннелю можно следующим образом подключиться к серверу MySQL с рабочей станции:

```
$ mysql -h 127.0.0.1 -P 4406
```

SSH – очень мощный инструмент, способный на гораздо более сложные вещи, чем показано в этом простеньком примере. Существует также программа *stunnel*, которая также позволяет создавать защищенные туннели, но без компонента для удаленного входа в систему. В некоторых случаях она является неплохой заменой VPN-сети.

---

[1] **Для Windows-клиентов имеется версия OpenSSH, а также популярная программа** Putty (*http://www.chiark.greenend.org.uk/~sgtatham/putty/*). Подробное пособие по настройке SSH-туннелей для подключения к MySQL есть на страницах *http://www.vbmysql.com/articles/security/protecting-mysql-sessions-with-ssh-port-forwarding-part-1 и http://www.vbmysql.com/articles/security/protecting-mysql-sessions-with-ssh-port-forwarding-part-2*.

[2] В предположении, что установлена программа SSH версии 2. В версии 1 нет флага *-N*. Подробности см. в документации по SSH.

TCP Wrappers

На платформе UNIX MySQL можно откомпилировать с поддержкой технологии TCP Wrappers. Если полноценный брандмауэр почему-либо не подходит, то эта технология поможет организовать базовый уровень защиты. Не изменяя таблицы привилегий, вы получите возможность контролировать, с какими компьютерами сможет и не сможет общаться сервер MySQL. В некоторых дистрибутивах, например Debian GNU/Linux, MySQL уже откомпилирован подобным образом.

Чтобы воспользоваться TCP Wrappers, нужно собрать MySQL из исходных текстов, передав программе *configure* флаг *--withlibwrap*, чтобы она знала, где найти нужные заголовочные файлы:

```
$ ./configure --with-libwrap=/usr/local/tcp_wrappers
```

В предположении, что в файле */etc/hosts.deny* имеется запись, запрещающая все соединения по умолчанию:

```
# запретить все соединения
ALL: ALL
```

можно явно добавить MySQL в файл */etc/hosts.allow*:

```
# разрешить соединение с mysql с машин в локальной сети
mysqld: 192.168.1.0/255.255.0.0 : allow
```

Нужно также не забыть включить запись, описывающую MySQL, в файл */etc/services*. Добавьте следующую строку, если ее еще нет:

```
mysql 3306/tcp # MySQL Server
```

Если MySQL работает на нестандартном порту, укажите его номер вместо 3306.

Использование библиотеки TCP Wrappers сопряжено с определенными накладными расходами, в частности может выполняться обратный DNS-поиск. Это создает зависимость от системы DNS, которой вы, возможно, хотели бы избежать.

Автоматическая блокировка хоста

MySQL оказывает некоторую помощь в предотвращении сетевых атак: если она замечает слишком много неудачных попыток соединения с определенного компьютера, то начинает отвергать поступающие с него запросы. Серверная переменная `max_connection_errors` определяет, сколько должно быть неудачных попыток, чтобы MySQL приступил к блокированию. «Неудачной» называется любая незавершенная попытка соединения (то есть, не приведшая к созданию корректного сеанса MySQL). Чаще всего причиной является неверный пароль, но неудача может быть обусловлена и сетевыми проблемами.

Заблокировав некий хост, MySQL помещает в журнал такое сообщение:

```
Host 'host.badguy.com' blocked because of many connection errors.
```

```
Unblock with 'mysqladmin flush-hosts'
```

Как видно из текста сообщения, разблокировать хост можно командой *mysqladmin flush-hosts*, но сначала стоит выяснить, с чем была связана проблема, и как-то ее разрешить. Команда *mysqladmin flush-hosts* просто выполняет SQL-команду `FLUSH HOSTS`, которая очищает хранящиеся в памяти сервера таблицы хостов. При этом разблокируются *все* заблокированные хосты; разблокировать только один хост невозможно.

Если такая ситуация начинает часто повторяться, то можно присвоить переменной `max_connection_errors` в файле *my.cnf* большое значение, чтобы воспрепятствовать блокировке хостов:

```
max_connection_errors=999999999
```

Невозможно установить `max_connection_errors` в 0 и тем самым вообще отключить проверку, да и не стоит этого делать. Лучше разобраться в проблеме и устранить ее.

Шифрование данных

Если в приложении хранятся секретные данные, например записи о банковских счетах, то, возможно, имеет смысл хранить их в зашифрованном виде. Тогда неуполномоченному лицу будет очень трудно воспользоваться данными, даже получив физический доступ к серверу. Подробное обсуждение достоинств и недостатков различных алгоритмов и методов шифрования выходит за рамки этой книги, но краткий обзор соответствующей тематики мы все же дадим.

Хеширование паролей

Если приложение не очень секретное, то защищать имеет смысл только небольшую часть информации, например, сведения о паролях. Пароли ни в коем случае не следует хранить в открытом виде, поэтому обычно они шифруются. Однако вместо шифрования можно взять за образец принятую в большинстве UNIX-систем да и в самом MySQL практику: применить к паролю алгоритм хеширования и сохранить в таблице получившийся результат (свертку).

В отличие от традиционного шифрования, которое можно обратить, хорошая функция хеширования необратима. Единственный способ узнать пароль, по которому была сгенерирована свертка, – применить требующий огромных вычислительных ресурсов полный перебор (то есть, испробовать все возможные варианты входных данных).

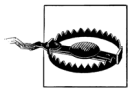
В MySQL есть три функции для хеширования паролей: `ENCRYPT()`, `SHA1()` и `MD5()`[1]. Чтобы посмотреть, что они возвращают, проще всего приме-

[1] Функция `ENCRYPT()` просто вызывает функцию `crypt()` из библиотеки языка C. В некоторых вариантах UNIX эта функция представляет собой реализацияю алгоритма MD5, тогда `ENCRYPT()` ничем не отличается от `MD5()`. В других же вариантах применяется традиционный алгоритм DES.

нить каждую из них к одному и тому же входному тексту. Давайте сделаем это на примере строки p4ssword:

```
mysql> SELECT MD5('p4ssword'), ENCRYPT('p4ssword'), SHA1('p4ssword')\G
*************************** 1. row ***************************
MD5('p4ssword'): 93863810133ebebe6e4c6bbc2a6ce1e7
ENCRYPT('p4ssword'): dDCjeBzIycENk
SHA1('p4ssword'): fbb73ec5afd91d5b503ca11756e33d21a9045d9d
```

Все три функции возвращают строку букв и цифр фиксированной длины, которую можно сохранить в столбце типа CHAR. Поскольку ENCRYPT() может возвращать буквы в разных регистрах, то лучше указать для столбца тип CHAR BINARY.



Никогда не используйте в приложениях внутреннюю функцию MySQL PASSWORD(). Возвращаемые ей результаты зависят от версии MySQL.

Для сохранения свертки данных достаточно такой команды:

```
mysql> INSERT INTO user_table (user, pass) VALUES ('user', MD5('p4ssword') );
```

Чтобы проверить пароль пользователя *user*, можно выполнить запрос SELECT и проверить совпадает ли свертка указанного переданного пароля с тем, что хранится в базе. На языке Perl это можно сделать следующим образом:

```
my $sth = $dbh->prepare('SELECT * FROM user_table '
        . 'WHERE user = ? AND pass = MD5(?)');
$sth->execute($username, $password);
```

Хеширование паролей – простой и относительно безопасный способ хранить пароли в базе данных, не опасаясь взлома. Чтобы усложнить атаку по словарю, можно хешировать комбинацию имени и пароля пользователя, так что результат зависит от большего числа переменных:

```
my $sth = $dbh->prepare('SELECT * FROM user_table '
        . 'WHERE user = ? AND pass = SHA1(CONCAT(?, ?))');
$sth->execute($username, $username, $password);
```

Единственная проблема – это риск, сопряженный с тем, что серверу пароли отправляются в открытом виде; он может быть записан в виде обычного текста в журнал на диске и виден в дампе памяти процесса. Чтобы немного снизить этот риск, можно хранить пароль в пользовательской переменной или вообще переместить хеширование на уровень приложения, чтобы устранить проблему на корню. Для большинства языков программирования имеются функции или библиотеки шифрования. Чуть ниже мы рассмотрим вопрос о шифровании на уровне приложения.

Зашифрованные файловые системы

Поскольку все подсистемы хранения MySQL содержат данные в обычных файлах в той файловой системе, которая для них отведена, то можно использовать и зашифрованную файловую систему. В большинстве популярных ОС имеется по крайней мере одна такая файловая система (бесплатная или коммерческая).

Основное достоинство такого подхода заключается в том, что от MySQL не требуется ничего особенного. Шифрование и дешифрирование производятся вне MySQL, поэтому серверу остается только читать и писать данные, не зная, что с ними происходит дальше. Вам необходимо лишь разместить файлы данных и журналов в подходящей файловой системе. Для приложения такая организация также прозрачна.

У использования зашифрованных файловых систем совместно с MySQL есть и недостатки. Прежде всего, на шифрование и дешифрирование данных, индексов и журналов уходит процессорное время. Если вы подумываете о применении зашифрованной файловой системы, то предварительно прогоните хорошие эталонные тесты, чтобы стало понятно, как поведет себя система под высокой нагрузкой.

Кроме того, позаботьтесь о том, чтобы информация не расшифровывалась при снятии резервной копии. Этому правилу нетрудно следовать, но о нем легко забыть.

И, наконец, имейте в виду, что зашифрованная файловая система не дает защиты от человека, получившего доступ к серверу, на котором хранятся данные. Поскольку сервер, смонтировавший файловую систему, производит дешифрирование прозрачно, то всякий имеющий к нему доступ может прочитать интересующие его сведения и сделать незашифрованную копию.

Шифрование на уровне приложения

Чаще шифрование встраивают непосредственно в приложение (или в ПО промежуточного уровня). Когда приложение собирается сохранить секретные данные, оно сначала шифрует их, а затем отправляет результат серверу MySQL. И наоборот, после извлечения зашифрованных данных из базы приложение должно их расшифровать.

Этому подходу присуща большая гибкость. Вы не привязаны к конкретной файловой системе, ОС и даже СУБД (если код написан инвариантно), и разработчик вправе выбрать тот алгоритм шифрования, который считает наиболее подходящим (с учетом компромисса между быстродействием и стойкостью).

Поскольку данные зашифрованы, создание резервной копии не вызывает никаких сложностей. Куда бы ни копировалась информация, она останется зашифрованной. Однако это означает, что доступ к данным

возможен только при посредничестве приложения, которое знает, как их расшифровать. Не получится просто запустить командный клиент *mysql* и выполнить запрос.

Шифрование на уровне приложения часто оказывается хорошим решением, но и у него есть минусы. Например, эффективно индексировать зашифрованные данные гораздо труднее, а еще труднее в этом случае оптимизировать производительность MySQL.

Вопросы проектирования

Вышеупомянутая свобода и гибкость имеют любопытные последствия для проектирования базы данных. Во-первых, нужно выбирать тип столбца, подходящий для используемого алгоритма шифрования. Некоторые алгоритмы порождают блоки данных фиксированного минимального размера. Следовательно, для хранения зашифрованного элемента необходимо отвести столбец длиной 256 байтов, пусть даже до шифрования он был гораздо короче. Кроме того, многие популярные библиотеки шифрования порождают двоичные данные, поэтому тип столбца надо выбирать соответствующим. В качестве альтернативы можно преобразовать двоичные данные в шестнадцатеричный вид или в представление base64, но на это уходит дополнительное место и время.

Также трудно решить, что следует, а что не следует шифровать. Необходимо соблюдать баланс между безопасностью данных и сложностью выполнения запросов к таблицам. Например, в таблице `account` для хранения информации о банковских счетах могут быть такие столбцы:

- `id`
- `type`
- `status`
- `balance`
- `overdraft_protection`
- `date_established`

Какие из них имеет смысл шифровать? Если зашифровать баланс, что, на первый взгляд, представляется разумным, то будет трудно составлять даже простейшие отчеты. Вот, например, запрос, который находит минимальный, максимальный и средний баланс по всем счетам, группируя их по типам:

```
mysql> SELECT MIN(balance), MAX(balance), AVG(balance)
    -> FROM account GROUP BY type;
```

Однако он вернет бессмысленные результаты. MySQL не знает, что столбец `balance` зашифрован, поэтому попытается просто применить агрегатные функции к криптованным данным.

Чтобы решить эту задачу, приложению придется прочитать все строки из таблицы `account` и самостоятельно выполнить агрегирование. Быть

может, это не слишком сложно, но обидно. Вы не только заново реализуете уже встроенную в MySQL функциональность, но и существенно замедляете процедуру.

Поэтому все сводится к поиску компромисса между безопасностью и преимуществами использования реляционной базы данных. Любой столбец, содержащий зашифрованные данные, бесполезен для встроенных в MySQL агрегатных функций, поскольку они могут оперировать только незашифрованными данными. Аналогичные проблемы возникают и при оптимизации запросов. Например, в отсутствии шифрования очень легко найти все счета с балансом больше $100000:

```
mysql> SELECT * FROM account WHERE balance > 100000;
```

Если по столбцу `balance` построен индекс и этот столбец не зашифрован, то для поиска нужных строк MySQL может воспользоваться индексом. Если же данные зашифрованы, то необходимо выбрать все строки в приложение, расшифровать их, а затем отфильтровать.

Шифрование и дешифрирование внутри MySQL

Можно хранить криптованные данные в базе MySQL, производя шифрование и дешифрирование с помощью встроенных инструментов. Лучше всего для этой цели подходят функции `AES_ENCRYPT()` и `AES_DECRYPT()`, которые преобразуют строки в зашифрованные двоичные последовательности и обратно. Эти функции реализуют симметричное криптование: для шифрования и дешифрирования применяется один и тот же ключ. Например:

```
mysql> SET @key := 's3cret';
mysql> SET @encrypted := AES_ENCRYPT('sensitive data', @key);
mysql> SELECT AES_DECRYPT(@encrypted, @key);
+-------------------------------+
| AES_DECRYPT(@encrypted, @key) |
+-------------------------------+
| sensitive data                |
+-------------------------------+
```

Мы не показываем зашифрованное значение, так как это просто двоичные нули и единицы, которые человеку представляются бессмысленным набором символов.

Но и такой подход не решает все вышеупомянутые проблемы. Прежде всего, сложность с индексированием остается. Кроме того, данные, которые вы пытаетесь зашифровать, в SQL-запросе представляются открытым текстом и будут записаны в журнал сервера (если он, конечно, включен). Однако мы все же показали, как можно снизить риск того, что другие пользователи увидят вашу секретную информацию: храните ключ шифрования в пользовательской переменной. Есть и другие, более безопасные способы присвоить этой переменной значение. Например, можно поместить ключ в хранимую процедуру, вызывать эту

процедуру для вычисления зашифрованного значения и ограничить к ней доступ. Тогда другим пользователям будет сложнее узнать значение ключа.

Модификация исходного кода

Если вы ищете подход более гибкий, чем зашифрованные файловые системы и шифрование на уровне приложения, то всегда можно создать специализированное решение. Исходный код MySQL распространяется бесплатно на условиях лицензии GNU General Public License.

Но для такого рода работы нужно владеть языком C++ или нанять специалиста в этой области. Кроме того, есть два варианта действий: написать собственную подсистему хранения с встроенной поддержкой шифрования или добавить шифрование к уже имеющейся подсистеме.

MySQL в окружении с измененным корневым каталогом

При работе на платформе UNIX можно существенно повысить общую защищенность системы, если запускать сервер в окружении с измененным корневым каталогом (chrooted). При этом в файловой системе создается изолированная среда, не допускающая доступа к файлам вне заданной папки. В результате, даже если будет обнаружена ошибка в коде сервера и для нее создан эксплойт, потенциальный ущерб ограничится только содержимым этого каталога, где должны храниться лишь файлы, относящиеся к конкретному приложению.

Если вы хотите запускать приложение MySQL в окружении с измененной корневой папкой, то начать нужно либо с компиляции сервера из исходных текстов, либо с распаковки и установки двоичного дистрибутива, поставляемого компанией MySQL AB. Многие администраторы считают это само собой разумеющимся, но так или иначе для приложения с измененным корневым каталогом другого пути нет: многие готовые дистрибутивы MySQL устанавливают файлы в каталог */usr/bin*, другие – в */var/lib/mysql*, но в случае инсталляции с измененным корневым каталогом все файлы должны находиться в одной и той же папке.

Мы собираемся создать папку */chroot*, в которой будут «жить» все наши приложения с измененным корневым каталогом. Для этого следует сконфигурировать файлы сборки MySQL следующей командой:

```
$ ./configure --prefix=/chroot/mysql
```

Затем откомпилируйте MySQL как обычно и запустите процедуру установки, которая поместит все файлы в дерево с корнем */chroot/mysql*.

Следующий шаг – небольшое волшебство, от которого все станут счастливы. Имя *chroot* – это сокращение от *change root* (*изменить корень*). Если ввести команду:

```
$ chroot /chroot/mysql
```

то каталог */chroot/mysql* станет корневым /. Поскольку одни и те же файлы использует как сервер с измененным корневым каталогом, так и клиент с неизмененным корневым каталогом, то нужно настроить файловую систему так, чтобы и сервер, и клиенты могли найти необходимые им файлы. Проще всего сделать это следующим образом:

```
$ cd /chroot/mysql
$ mkdir chroot
$ cd chroot
$ ln -s /chroot/mysql mysql
```

При этом создается символическая ссылка */chroot/mysql/chroot/mysql*, которая на самом деле ведет на */chroot/mysql/*. Теперь даже если приложение, работающее в окружении с измененным корневым каталогом, попытается зайти в папку */chroot/mysql/*, оно будет перенаправлено в нужное место. Но и клиентское приложение, работающее вне среды с измененным корневым каталогом, находит необходимые ему файлы.

Последний шаг – дать нужные указания программе *mysqld_safe*, чтобы сервер MySQL мог запуститься и выполнить команду *chroot* с требуемым каталогом. Для этого нужно ввести что-то вроде:

```
$ mysqld_safe --chroot=/chroot/mysql --user=1001
```

Обратите внимание, что мы использовали идентификатор пользователя (UID) MySQL вместо *--user=mysql*. Объясняется это тем, что в окружении с измененным корневым каталогом сервер MySQL уже не может обратиться к библиотекам, которые занимаются поиском UID по имени пользователя.[1]

При работе с сервером MySQL в окружении с измененным корневым каталогом есть некоторые тонкости. Команда `LOAD DATA INFILE` и другие команды, напрямую обращающиеся к файлам, могут работать совсем не так, как вы ожидаете, поскольку сервер больше не считает, что / – это корень файловой системы. Поэтому, когда вы просите загрузить данные из файла */tmp/filename*, позаботьтесь о том, чтобы на самом деле он находился в */chroot/mysql/tmp/filename*, иначе MySQL не сможет его найти.

Окружение с измененным корневым каталогом – лишь один из способов частично изолировать MySQL. Существуют и другие, например «тюрьмы» (jails) в ОС FreeBSD, зоны (Zones) в Solaris и виртуализация.

[1] Наш опыт тестирования такой конфигурации показывает, что достаточно скопировать файлы *libnss** в каталог с библиотеками MySQL в окружении со смещенным корнем, но с практической точки зрения лучше не заниматься такими вещами, а просто ввести UID непосредственно в сценарий запуска.

13

Состояние сервера MySQL

На многие вопросы о сервере MySQL можно ответить, посмотрев на его состояние. MySQL раскрывает информацию о своем внутреннем состоянии двумя способами: самый новый – стандартизированная база данных INFORMATION_SCHEMA и более традиционный – набор команд SHOW (они продолжают поддерживаться, хотя предпочтительным механизмом для новых возможностей считается база данных INFORMATION_SCHEMA). Кое-какая информация, доступная с помощью команд SHOW, пока отсутствует в таблицах базы INFORMATION_SCHEMA.

Вы должны знать, как определить, что именно относится к вашей проблеме, как получить требуемую информацию и как ее интерпретировать. Хотя MySQL предоставляет очень много сведений о том, что происходит внутри сервера, воспользоваться ими не так-то легко. Для понимания того, что есть что, нужно терпение, опыт и свободный доступ к руководству по MySQL.

Существуют инструменты, помогающие интерпретировать состояние сервера в различных контекстах, например в ходе мониторинга или профилирования, и некоторые из них мы упомянем в следующей главе. Однако все равно необходимо понимать смысл переменных хотя бы на верхнем уровне – как минимум, знать, какие существуют категории, – и уметь получать их от сервера.

В этой главе мы рассмотрим многие команды состояния и возвращаемые ими результаты. Если некоторая тема детально рассматривается в другом месте книги, мы дадим соответствующую отсылку.

Системные переменные

MySQL показывает множество серверных системных переменных по команде SHOW VARIABLES. Эти переменные можно использовать в выражениях. Кроме того, их можно применять и в командной строке, запуская

программу *mysqladmin variables*. Начиная с версии 5.1 они также доступны через таблицы в базе данных INFORMATION_SCHEMA.

Эти переменные представляют разнообразную конфигурационную информацию, например подразумеваемую по умолчанию подсистему хранения (storage_engine), поддерживаемые названия часовых поясов, схему упорядочения для соединения и параметры запуска. О том как устанавливать и использовать системные переменные, мы рассказывали в главе 6.

Команда SHOW STATUS

Команда SHOW STATUS выводит переменные состояния сервера в виде таблицы с двумя столбцами: имя и значение. В отличие от серверных переменных, упомянутых в предыдущем разделе, эти предназначены только для чтения. Посмотреть их можно с помощью SQL-команды SHOW STATUS или вызвав программу *mysqladmin extended-status* из командной строки. При использовании SQL-команды можно употреблять фразы LIKE и WHERE для фильтрации результатов; оператор LIKE выполняет стандартное сопоставление имени переменной с шаблоном. В обоих случаях возвращается таблица результатов, но ее нельзя сортировать, соединять с другими таблицами и выполнять иные стандартные операции, применимые к таблицам MySQL.

Мы применяем термин «переменная состояния», говоря о переменных, которые выводит команда SHOW STATUS, и термин «серверная системная переменная», когда речь идет о конфигурационной переменной сервера.

Поведение команды SHOW STATUS в версии MySQL 5.0 существенно изменилось, но это можно не заметить, если специально не присматриваться. Теперь MySQL поддерживает некоторые переменные глобально, а другие – на уровне отдельного соединения. Таким образом, SHOW STATUS выводит и глобальные, и сеансовые переменные. У многих из них двойная область видимости: глобальные и сеансовые переменные называются одинаково. Кроме того, команда SHOW STATUS теперь по умолчанию показывает только сеансовые переменные; если вы привыкли просматривать с ее помощью глобальные, то вас ждет разочарование. Для просмотра глобальных переменных нужно выполнить команду SHOW GLOBAL STATUS[1].

В версии MySQL 5.1 и последующих значения переменных состояния можно выбирать непосредственно из таблиц INFORMATION_SCHEMA.GLOBAL_STATUS и INFORMATION_SCHEMA.SESSION_STATUS. В сервере MySQL 5.0 есть сотни переменных состояния, и с каждой новой версией их количество уве-

[1] Здесь вас подстерегает сюрприз: при использовании старой версии *mysqladmin* с новым сервером команда SHOW GLOBAL STATUS не вызывается, так что отображается «неправильная» информация.

личивается. Большинство представляют собой либо счетчики, либо содержат текущее значение какой-то метрики. Счетчики увеличиваются всякий раз, как MySQL выполняет некоторое действие, например инициирует полное сканирование таблицы (`Select_scan`). Метрики, в частности количество открытых соединений, могут как увеличиваться, так и уменьшаться. Иногда несколько переменных относятся к одному и тому же, например `Connections` (количество попыток соединений с сервером) и `Threads_connected`; в таком случае переменные взаимосвязаны, но из их имен такая связь не всегда очевидна.

Счетчики хранятся в виде целых чисел без знака. В 32-разрядных сборках они занимают 4 байта, а в 64-разрядных – 8 байтов. По достижении максимального значения счетчик обращаются в нуль. Когда ведется непрерывный мониторинг переменных, такой переход через нуль нужно отслеживать и учитывать; следует понимать, что если сервер проработал достаточно долго, то наблюдаемое значение может оказаться меньше ожидаемого просто потому, что когда-то произошел переход через нуль (в 64-разрядных сборках такая проблема возникает гораздо реже).

Изучать многие из этих переменных лучше всего, наблюдая за их изменением на протяжении нескольких минут. Для этого можно воспользоваться программой *mysqladmin extended-status -r -i 5* или *innotop*

Ниже приводится краткий обзор – ни в коем случае не исчерпывающий – различных категорий переменных, которые показывает команда `SHOW STATUS`. Полное описание следует искать в руководстве по MySQL, все переменные документированы на странице *http://dev.mysql.com/doc/en/mysqld-option-tables.html*. Говоря о группе взаимосвязанных переменных, имена которых начинаются с общего префикса, мы будем собирательно называть их «переменными `<prefix>_*`».

Статистика потоков и соединений

В этих переменных отслеживаются попытки соединения, разорванные соединения, статистика сетевого трафика и использования потоков.

- `Connections`, `Max_used_connections`, `Threads_connected`
- `Aborted_clients`, `Aborted_connects`
- `Bytes_received`, `Bytes_sent`
- `Slow_launch_threads`, `Threads_cached`, `Threads_created`, `Threads_running`

Если `Aborted_connects` не равна нулю, то это свидетельствует, скорее всего, о проблемах в сети или о неудавшейся попытке соединения (например, из-за того, что введен неверный пароль или имя базы данных). Если это значение становится чрезмерно большим, то возможны серьезные побочные эффекты: MySQL может заблокировать хост. Подробнее об этом см. главу 12.

Переменная `Aborted_clients` называется похоже, но смысл ее совершенно иной. Увеличение ее значения обычно связано с ошибками в прило-

жении, в частности, из-за того, что программист забыл корректно закрыть соединение с MySQL перед завершением программы. Как правило, это не является признаком серьезной проблемы.

Полезная метрика – количество соединений, создаваемых в секунду (`Threads_created/Uptime`). Если оно сильно отличается от нуля, значит кэш потоков слишком мал, и для новых соединений не удается найти свободного потока в кэше.

Наиболее полезно последить за этими переменными на протяжении нескольких минут, а не за все время с момент запуска сервера.

Состояние записи в двоичный журнал

Переменные состояния `Binlog_cache_use` и `Binlog_cache_disk_use` показывают, сколько транзакций было сохранено в кэше двоичного журнала и размер скольких транзакций оказался настолько велик, что они не поместились в кэш и их пришлось записывать во временный файл. О том, как выбирать размер кэша двоичного журнала, мы рассказывали в главе 6.

Счетчики команд

Переменные `Com_*` подсчитывают, сколько раз вызывались SQL-команды или функции C API каждого вида. Например, `Com_select` – это количество команд `SELECT`, а `Com_change_db` показывает, сколько раз изменялась подразумеваемая по умолчанию база данных – то ли с помощью команды `USE`, то ли в результате вызова функции C API. В переменной `Questions` подсчитывается общее количество запросов и команд, полученных сервером. Однако оно не равно в точности сумме всех переменных `Com_*` из-за попаданий в кэш запросов, закрытых и разорванных соединений и, возможно, других факторов.

Значение переменной состояния `Com_admin_commands` может быть очень велико. Она подсчитывает не только административные команды, но и ping-запросы к самому экземпляру сервера MySQL. Эти запросы выдаются с помощью функции C API и обычно исходят из клиентского кода, например такого:

```
my $dbh = DBI->connect(...);
while ( $dbh && $dbh->ping ) {
   # Что-то сделать
}
```

Такие ping-запросы считаются «мусорными». Быть может, они не слишком нагружают сервер, но все равно впустую транжирят его ресурсы. Нам встречались системы объектно-реляционного отображения, которые «пингуют» сервер перед каждым запросом, что абсолютно бессмысленно. Приходилось нам видеть и библиотеки абстрагирования баз данных, в которых перед каждым запросом изменяется подразумеваемая

база, в результате чего значение переменной Com_change_db оказывается очень велико. Того и другого лучше избегать.

Временные таблицы и файлы

Посмотреть переменные, в которых подсчитывается, сколько раз MySQL создавала временные таблицы и файлы, можно такой командой:

```
mysql> SHOW GLOBAL STATUS LIKE 'Created_tmp%';
```

Операции обработчиков

API обработчиков (handler API) – это интерфейс между сервером MySQL и его подсистемами хранения. Переменные Handler_* подсчитывают, например, сколько раз MySQL просила подсистему хранения прочитать следующую строку из индекса. Изучение переменных Handler_* может пролить свет на то, какие действия сервер выполняет чаще всего. Они полезны и для профилирования запросов. Посмотреть на них можно с помощью такой команды:

```
mysql> SHOW GLOBAL STATUS LIKE 'Handler_%';
```

Буфер ключей MyISAM

Переменные Key_* содержат метрики и счетчики, характеризующие буфер ключей MyISAM. Посмотреть на них можно с помощью такой команды:

```
mysql> SHOW GLOBAL STATUS LIKE 'Key_%';
```

О том, как анализировать и настраивать работу кэшей ключей, подробно рассказано в главе 6.

Дескрипторы файлов

Если вы применяете главным образом подсистему хранения MyISAM, то важно следить за статистикой использования дескрипторов файлов, – она показывает, как часто MySQL открывает файлы с расширениями *.frm*, *.MYI* и *.MYD*. InnoDB хранит все данные в файлах табличного пространства, поэтому эти переменные не столь важны. Посмотреть на переменные Open_* можно с помощью такой команды:

```
mysql> SHOW GLOBAL STATUS LIKE 'Open_%';
```

О том, как настраивать параметр, отражающийся на значениях этих переменных, подробно рассказано в главе 6.

Кэш запросов

Оценить работу кэша запросов позволяют переменные состояния Qcache_*. Если вы полагаетесь на кэш запросов для повышения производительно-

сти, то существенны все переменные из этой группы. Посмотреть на них можно с помощью такой команды:

```
mysql> SHOW GLOBAL STATUS LIKE 'Qcache_%';
```

Подробное описание настройки кэша запросов приведено в главе 5.

Типы команд SELECT

В переменных `Select_*` подсчитывается количество запросов определенных типов `SELECT`. Они полезны, когда нужно оценить соотношение между запросами с разными планами выполнения. К сожалению, для других типов команд, в частности `UPDATE` и `REPLACE`, соответствующих переменных состояния не существует; однако для изучения производительности запросов, отличных от `SELECT`, можно воспользоваться переменными состояния `Handler_*` (см. выше). Чтобы увидеть все переменные `Select_*`, выполните такую команду:

```
mysql> SHOW GLOBAL STATUS LIKE 'Select_%';
```

На наш взгляд, переменные `Select_*` можно расположить в порядке возрастания стоимости запроса следующим образом:

`Select_range`

Количество операций соединения, в которых производился просмотр диапазона индекса для первой таблицы.

`Select_scan`

Количество операций соединения, в которых производилось полное сканирование первой таблицы. В этом нет ничего плохого, если в результат соединения должны войти все строки из первой таблицы, но никуда не годится, если нужны только определенные строки, а для их эффективной выборки нет индекса.

`Select_full_range_join`

Количество операций соединения, в которых использовалось значение из таблицы *n* для выборки строк по диапазону индекса для таблицы *n + 1*. В зависимости от запроса, это может быть как дешевле, так и дороже, чем `Select_scan`.

`Select_range_check`

Количество операций соединения, в которых для каждой строки из таблицы *n* приходилось заново оценивать индексы таблицы *n + 1*, чтобы понять, какой из них окажется самым дешевым. Обычно это означает, что над таблицей *n + 1* нет индексов, полезных для соединения. Накладные расходы для такого плана выполнения очень высоки.

`Select_full_join`

Перекрестное соединение, для которого не заданы критерии отбора строк из таблиц. Количество просматриваемых в этом случае строк

равно произведению количества строк во всех соединяемых таблицах. Обычно это очень плохо.

На качественно настроенном сервере последние две переменные не должны быстро увеличиваться. Иногда индикатором плохо оптимизированной рабочей нагрузки может служить отношение этих двух счетчиков к общему количеству запросов SELECT, обработанных сервером (Com_select). Если хотя бы один из них составляет больше, чем несколько процентов, то, вероятно, следует оптимизировать запросы или схему.

С этими переменными тесно связана переменная Slow_queries. Разработанные нами заплаты для журнала медленных запросов позволяют увидеть, для каких запросов было произведено полное соединение, какие были обслужены из кэша запросов и т. д. Дополнительную информацию см. в разделе «Тонкая настройка протоколирования» главы 2, на стр. 99.

Операции сортировки

В главах 3 и 4 мы много говорили об оптимизации сортировки, поэтому вы уже должны хорошо представлять себе механизм ее работы. Если MySQL не может воспользоваться индексом для выборки строки в нужном порядке, то он вынужден прибегнуть к сортировке (filesort), в результате чего увеличиваются переменные состояния Sort_*. Если не считать переменную Sort_merge_passes, то повлиять на их значения можно только путем добавления индексов, которые MySQL могла бы задействовать для сортировки. Что же касается переменной Sort_merge_passes, то она зависит от серверной переменной sort_buffer_size (не путайте с myisam_sort_buffer_size). MySQL использует буфер сортировки для хранения порции обрабатываемых строк. После сортировки они объединяются с ранее вычисленным результатом, переменная Sort_merge_passes увеличивается на 1 и буфер заполняется следующей порцией строк. Если буфер сортировки слишком мал, то эту последовательность действий придется выполнить много раз, поэтому значение переменной состояния будет велико.

Посмотреть на переменные Sort_* можно с помощью такой команды:

```
mysql> SHOW GLOBAL STATUS LIKE 'Sort_%';
```

MySQL увеличивает переменные Sort_scan и Sort_range, когда читает отсортированные строки из результата файловой сортировки и возвращает их клиенту. Разница между ними в том, что первая переменная увеличивается, когда план запроса привел к увеличению Select_scan (см. предыдущий раздел), а вторая – когда увеличивается Select_range. Оба вида сортировки идентичны с точки зрения реализации и стоимости – переменные просто отражают различие между типами планов запроса, приведших к сортировке.

Блокирование таблиц

Переменные Table_locks_immediate и Table_locks_waited говорят о том, сколько блокировок было выдано немедленно, а скольких пришлось ждать. Если в списке, возвращенном командой SHOW FULL PROCESSLIST, вы видите много потоков в состоянии Locked, проверьте эти переменные. Но имейте в виду, что они отражают лишь статистику блокировки на уровне сервера, а не подсистемы хранения. Об отладке блокировок см. приложение D.

Secure Sockets Layer (SSL)

Переменные Ssl_* показывают то, как сервер сконфигурирован для работы по протоколу SSL (если его поддержка включена). Посмотреть на них можно с помощью команды:

```
mysql> SHOW GLOBAL STATUS LIKE 'Ssl_%';
```

Переменные, относящиеся к InnoDB

Переменные Innodb_* показывают некоторые данные, выдаваемые командой SHOW INNODB STATUS, которая рассматривается ниже в этой главе. Эти переменные можно сгруппировать по имени: Innodb_buffer_pool_*, Innodb_log_* и т. д. Внутренние механизмы InnoDB мы обсудим более подробно, когда будем говорить о команде SHOW INNODB STATUS.

Эти переменные, появившиеся в версии MySQL 5.0, имеют важный побочный эффект: при их просмотре запрашивается глобальная блокировка, которая удерживается в течение времени обхода всего пула буферов InnoDB. И пока это происходит, остальные потоки блокируются и вынуждены ждать. Это искажает некоторые переменные состояния, например Threads_running, значение которой оказывается выше, чем обычно (иногда даже существенно выше, все зависит от того, насколько занят сервер). Тот же эффект проявляется при выполнении команды SHOW INNODB STATUS и доступе к статистике с помощью таблиц INFORMATION_SCHEMA (начиная с версии MySQL 5.0 команды SHOW STATUS и SHOW VARIABLES реализованы с помощью запросов к таблицам INFORMATION_SCHEMA).

Поэтому в упомянутых версиях MySQL такие операции могут обойтись дорого – слишком частые проверки состояния сервера (скажем, раз в секунду) сопряжены с заметными накладными расходами. Фильтрация с помощью команды SHOW STATUS LIKE не помогает, так как сначала извлекаются все переменные состояния, а уже потом отсеиваются ненужные.

Переменные, относящиеся к подключаемым модулям

Начиная с версии MySQL 5.1 поддерживается новая архитектура подключаемых подсистем хранения и предоставляется механизм для регистрации подсистемами собственных переменных состояния и конфигу-

рации. Такие переменные появятся, если вы подключите соответствующую подсистему хранения.

Разное

Упомянем еще некоторые переменные состояния:

`Delayed_*`, `Not_flushed_delayed_rows`

Это метрики и счетчики для запросов типа `INSERT DELAYED`.

`Last_query_cost`

Эта переменная показывает стоимость выработанного оптимизатором плана выполнения последнего запроса. О том, что такое стоимость плана выполнения запроса, мы говорили в главе 4.

`Ndb_*`

Эти переменные содержат конфигурационную информацию о подсистеме хранения NDB Cluster, если она используется.

`Slave_*`

Эти переменные появляются, если сервер выступает в роли подчиненного в процессе репликации. В случае покомандной репликации особенно важна переменная `Slave_open_temp_tables`. В разделе «Отсутствующие временные таблицы» в главе 8 (стр. 488) приведена дополнительная информация о репликации и временных таблицах.

`Tc_log_*`

Эти счетчики имеют смысл для сервера, выступающего в роли координатора XA-транзакций. Подробности см. в разделе «Распределенные (XA) транзакции» в главе 5 (стр. 329).

`Uptime`

Эта переменная содержит время, прошедшее с момента запуска сервера, в секундах.

Чтобы прочувствовать характеристики рабочей нагрузки, полезно сравнить значения взаимосвязанных переменных, принадлежащих одной группе, например всех переменных `Select_*` или `Handler_*`. При использовании программы *innotop* это легко делается в режиме сводки команд (Command Summary), но можно добиться того же результата и вручную с помощью команды типа *mysqladmin extended -r -i60 | grep Handler_*. Вот что показала *innotop* для переменных `Select_*` на проверенном нами сервере.

```
_____________________ Command Summary ____________________ _ _
Name                    Value   Pct    Last Incr   Pct
Select_scan            756582  59.89%         2   100.00%
Select_range           497675  39.40%         0     0.00%
Select_full_join         7847   0.62%         0     0.00%
```

```
Select_full_range_join   1159   0.09%          0      0.00%
Select_range_check          1   0.00%          0      0.00%
```

В первых двух колонках находятся значения с момента запуска сервера, а в последних двух – с момента последнего обновления (в данном случае 10 секунд назад). Процент вычисляется относительно суммы показанных значений, а не относительно суммы по всем запросам.

Несмотря на то, что доля полных соединений сравнительно невелика, стоит поинтересоваться, почему они вообще есть.

Команда SHOW INNODB STATUS

Подсистема хранения InnoDB выдает массу информации о своем внутреннем состоянии по команде `SHOW ENGINE INNODB STATUS`, или сокращенно `SHOW INNODB STATUS`. В отличие от большинства других команд `SHOW`, она выводит одну строку, а не таблицу, состоящую из строк и столбцов. Строка разделена на секции, в каждой из которых представлены сведения о каком-то одном аспекте InnoDB. Часть этой информации представляет интерес в основном для разработчиков InnoDB, но в остальном она весьма полезна – и даже необходима – любому, кто желает разобраться в работе InnoDB и настроить ее для получения максимальной производительности.

InnoDB обычно выводит 64-разрядные числа двумя частями: старшие 32 бита и младшие 32 бита. Примером может служить идентификатор транзакции: `TRANSACTION 0 3793469`. Чтобы получить полное 64-разрядное значение, нужно сдвинуть первое число на 32 бита влево и прибавить к нему второе. Мы продемонстрируем эту технику ниже.

Среди прочего команда `SHOW INNODB STATUS` выводит некоторые усредненные статистики, например количество вызовов `fsync()` в секунду. Усреднение производится за период с момента последнего выполнения команды, поэтому если вы хотите изучить статистику, запрашивайте ее с интервалами примерно 30 секунд, чтобы данные успели накопиться. Не все результаты генерируются строго в один и тот же момент, поэтому средние вычисляются по разным промежуткам времени. Кроме того, в InnoDB есть внутренний интервал сброса, который непредсказуем и меняется от версии к версии; необходимо обращать внимание на то, за какой временной промежуток сгенерированы средние, поскольку в разных выборках он может отличаться.

Результат содержит достаточно информации, чтобы при желании вручную вычислить средние большинства значений. Однако большую помощь может оказать инструмент мониторинга, например программа *innotop*, который сам вычисляет разности между последовательными выборками и производит усреднение.

Заголовок

Первая секция представляет собой заголовок, который знаменует начало вывода, содержит текущую дату и время, а также показывает, какой интервал прошел с момента последней распечатки. В строке 2 мы видим текущую дату и время. В строке 4 указано, за какой период производилось усреднение; это время, прошедшее либо с момента последней распечатки, либо с момента внутреннего сброса.

```
1 ====================================
2 070913 10:31:48 INNODB MONITOR OUTPUT
3 ====================================
4 Per second averages calculated from the last 49 seconds
```

Секция SEMAPHORES

Если рабочая нагрузка характеризуется высокой конкуренцией, то имеет смысл обратить внимание на следующую секцию, SEMAPHORES. В ней приведены данные двух видов: счетчики событий и текущий список ожиданий, который может и отсутствовать. Если вы наблюдаете какие-то узкие места, то эта информация поможет определить их причину. К сожалению, выявить узкие места недостаточно, надо еще както «расшить» их, а это уже сложнее. Но некоторые рекомендации мы дадим ниже. Вот пример распечатки:

```
1  ----------
2  SEMAPHORES
3  ----------
4  OS WAIT ARRAY INFO: reservation count 13569, signal count 11421
5  --Thread 1152170336 has waited at ./../include/buf0buf.ic line 630 for
   0.00 seconds the semaphore:
6  Mutex at 0x2a957858b8 created file buf0buf.c line 517, lock var 0
7  waiters flag 0
8  wait is ending
9  --Thread 1147709792 has waited at ./../include/buf0buf.ic line 630 for
   0.00   seconds the semaphore:
10 Mutex at 0x2a957858b8 created file buf0buf.c line 517, lock var 0
11 waiters flag 0
12 wait is ending
13 Mutex spin waits 5672442, rounds 3899888, OS waits 4719
14 RW-shared spins 5920, OS waits 2918; RW-excl spins 3463, OS waits 3163
```

В строке 4 приводится информация о массиве ожиданий операционной системы; это массив «слотов». InnoDB резервирует слоты в массиве семафоров. Семафор – это примитив операционной системы, с помощью которого она сигнализирует потокам, что ожидание закончилось и они могут продолжить работу. В этой строке показано, сколько раз InnoDB пришлось использовать механизм ожиданий операционной системы. Счетчик резервирования (reservation count) сообщает, как часто InnoDB выделяла слоты, а счетчик сигналов (signal count) – сколь-

ко раз потокам посылались сигналы с помощью семафоров из этого массива. Как мы скоро увидим, ожидание с помощью примитивов ОС обходится дороже активного ожидания (spin wait).

В строках 5–12 показаны потоки InnoDB, которые в данный момент ожидают освобождения мьютекса. В этом примере мы видим два ожидания, каждое из которых начинается словами «-- Thread <num> has waited...». Если бы рабочая нагрузка не характеризовалась высокой конкуренцией, заставляющей InnoDB прибегать к услугам ОС, то эта секция была бы пуста. В том случае если вы знакомы с исходным кодом InnoDB, весьма полезной информацией является имя файла, в котором поток ожидает события. Оно позволяет понять, в каком месте InnoDB возникают «горячие точки». Например, если много потоков ожидают в файле *buf0buf.ic*, то имеет место конкуренция за пул буферов. В распечатке также показано, как долго поток уже ждет, а поле «waiters flag» говорит о том, сколько потоков стоят в очереди к мьютексу.

Фраза «wait is ending» означает, что мьютекс уже освободился, но операционная система еще не запланировала выполнение потока.

Может возникнуть вопрос, чего именно ждет InnoDB. В этой подсистеме мьютексы и семафоры применяются для защиты критических секций кода; они могут разрешать одновременно войти в секцию только одному потоку, запрещать выполнение «писателей», когда есть активные «читатели», и т. д. В коде InnoDB имеется множество критических секций, и при определенных условиях в распечатке может появиться любая из них. Чаще всего речь идет о получении доступа к странице пула буферов.

После списка ожидающих потоков, в строках 13 и 14 мы видим дополнительные счетчики событий. В строке 13 показано несколько счетчиков, относящихся к мьютексам, а в строке 14 – к разделяемым и монопольным блокировкам чтения/записи. И в том, и в другом случае демонстрируется, сколько раз InnoDB прибегала к ожиданию на уровне ОС.

В InnoDB применяется многоступенчатая стратегия ожидания. Сначала она пытается подождать блокировку с помощью активного ожидания. Если после заранее заданного числа итераций (конфигурационный параметр `innodb_sync_spin_loops`) получить блокировку не удалось, то она обращается к более дорогому и сложному массиву ожиданий[1].

Активное ожидание (spin wait) обходится относительно дешево, но на него тратится время процессора, поскольку программа постоянно проверяет, можно ли заблокировать ресурс. Это не так плохо, как может показаться, поскольку обычно у процессора есть свободные циклы, пока он ожидает завершения ввода/вывода. А даже если свободных циклов нет, все равно активное ожидание гораздо дешевле альтерна-

[1] В версии MySQL 5.1 код массива ожиданий был переписан и стал гораздо более эффективным.

тивных способов. Однако опрос в цикле монополизирует процессор, не допуская к нему другие потоки, в которых, возможно, нашлась бы полезная работа.

Альтернативой активному ожиданию является контекстное переключение на уровне операционной системы, в результате которого другой поток получает возможность поработать, пока первый ждет. Спящий поток будет разбужен сигналом от семафора в массиве ожиданий. Сигнализация с помощью семафора реализована эффективно, но вот контекстное переключение – очень дорогая операция. Причем она производит кумулятивный эффект: тысячи контекстных переключений могут привести к ощутимым накладным расходам.

Можно попытаться соблюсти баланс между активными ожиданиями и ожиданиями на уровне ОС, изменив системную переменную `innodb_sync_spin_loops`. Если количество активных ожиданий в секунду не слишком велико (порядка сотен или тысяч), то можно о них особо не беспокоиться. Дополнительные рекомендации по поводу настройки этого аспекта InnoDB приведены в главе 6.

Секция LATEST FOREIGN KEY ERROR

Следующая секция, `LATEST FOREIGN KEY ERROR`, появляется только в том случае, если имели место ошибки внешнего ключа. В исходном коде много мест, где могут генерироваться такие ошибки, причем причины у них бывают разные. Иногда это родительские либо дочерние строки, которые транзакция искала в попытке вставить, обновить или удалить запись. А иногда виновато рассогласование между таблицами, обнаруженное при попытке добавить или удалить внешний ключ. А, быть может, изменение структуры таблицы, в которой уже есть внешний ключ.

Информация в этой секции очень полезна для выяснения точных причин сообщений об ошибках внешнего ключа, которые зачастую звучат туманно. Рассмотрим несколько примеров. Для начала создадим две таблицы, связанные внешним ключом, и вставим в них данные:

```
CREATE TABLE parent (
   parent_id int NOT NULL,
   PRIMARY KEY(parent_id)
) ENGINE=InnoDB;

CREATE TABLE child (
   parent_id int NOT NULL,
   KEY parent_id (parent_id),
   CONSTRAINT child_ibfk_1 FOREIGN KEY (parent_id)
   REFERENCES parent (parent_id)
) ENGINE=InnoDB;

INSERT INTO parent(parent_id) VALUES(1);
INSERT INTO child(parent_id) VALUES(1);
```

Существует два широких класса ошибок внешнего ключа. В первый попадают ошибки, которые возникают при добавлении, обновлении и удалении данных таким образом, что нарушается ограничение внешнего ключа. Например, вот что произойдет, удали мы строку из родительской таблицы:

```
DELETE FROM parent;
ERROR 1451 (23000): Cannot delete or update a parent row:
a foreign key constraint fails[1] (`test/child`, CONSTRAINT `child_ibfk_1`
FOREIGN KEY (`parent_id`) REFERENCES
`parent` (`parent_id`))
```

Текст достаточно понятен, такие сообщения выдаются при любой попытке добавить, обновить или удалить строки так, что это приведет к рассогласованию. Вот что покажет в этом случае команда SHOW INNODB STATUS:

```
 1 ------------------------
 2 LATEST FOREIGN KEY ERROR
 3 ------------------------
 4 070913 10:57:34 Transaction:
 5 TRANSACTION 0 3793469, ACTIVE 0 sec, process no 5488, OS thread id
   1141152064   updating or deleting, thread declared inside InnoDB 499
 6 mysql tables in use 1, locked 1
 7 4 lock struct(s), heap size 1216, undo log entries 1
 8 MySQL thread id 9, query id 305 localhost baron updating
 9 DELETE FROM parent
10 Foreign key constraint fails for table `test/child`:
11 ,
12 CONSTRAINT `child_ibfk_1` FOREIGN KEY (`parent_id`) REFERENCES `parent`
   (`parent_id`)
13 Trying to delete or update in parent table, in index `PRIMARY` tuple:
14 DATA TUPLE: 3 fields;
15 0: len 4; hex 80000001; asc ;; 1: len 6; hex 00000039e23d; asc 9 =;; 2:
   len 7; hex 000000002d0e24; asc - $;;
16
17 But in child table `test/child`, in index `parent_id`, there is a record:
18 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
19 0: len 4; hex 80000001; asc    ;; 1: len 6; hex 000000000500; asc    ;;
```

В строке 4 демонстрируется дата и время последней ошибки, связанной с внешним ключом. В строках 5–9 приведены подробные сведения о транзакции, в которой имела место ошибка, мы поясним их смысл чуть позже. В строках 10–19 точно описывается, что именно пыталась изменить InnoDB, когда обнаружила проблему. В основном, это данные строки, приведенные к печатаемому виду, об этом мы тоже еще поговорим.

[1] Не могу удалить или обновить родительскую строку: нарушено ограничение внешнего ключа. – *Прим. перев.*

Так-то оно так, но есть еще один класс ошибок внешнего ключа, которые отлаживать куда труднее. Вот что произойдет, если мы попытаемся изменить структуру родительской таблицы:

```
ALTER TABLE parent MODIFY parent_id INT UNSIGNED NOT NULL;
ERROR 1025 (HY000): Error on rename of './test/#sql-1570_9' to './test/
parent' (errno: 150)
```

Это сообщение уже не так понятно, но SHOW INNODB STATUS проливает свет на причину:

```
 1 ------------------------
 2 LATEST FOREIGN KEY ERROR
 3 ------------------------
 4 070913 11:06:03 Error in foreign key constraint of table test/child:
 5 there is no index in referenced table which would contain
 6 the columns as the first columns, or the data types in the
 7 referenced table do not match to the ones in table. Constraint:
 8 ,
 9 CONSTRAINT child_ibfk_1 FOREIGN KEY (parent_id) REFERENCES parent
   (parent_id)
10 The index in the foreign key in table is parent_id
11 See http://dev.mysql.com/doc/refman/5.0/en/innodb-foreign-key-constraints.
   html
12 for correct foreign key definition.
```

В данном случае ошибка возникла из-за различных типов данных. Столбцы, связанные внешним ключом, должны иметь *в точности* одинаковый тип данных, включая все модификаторы (в частности, UNSIGNED, из-за которого и произошла ошибка). Если вы видите ошибку 1025 и не понимаете, в чем дело, то начинать расследование следует с изучения распечатки SHOW INNODB STATUS.

Секция LATEST DETECTED DEADLOCK

Как и секция, относящаяся к внешним ключам, секция LATEST DETECTED DEADLOCK присутствует только в случае, если сервер обнаружил взаимоблокировку.

Взаимоблокировка возникает при наличии цикла в графе ожиданий (wait-for graph), где представлены строки, на которые удерживается блокировка, и строки, для которых ожидается освобождение блокировки. Этот цикл может быть произвольно большим. InnoDB обнаруживает взаимоблокировки мгновенно, так как проверяет наличие цикла всякий раз, когда транзакция вынуждена ждать освобождения блокировки строки. Взаимоблокировки могут быть весьма сложными, но в этой секции показаны только последние две участвующие транзакции, завершающая команда в каждой из них, и блокировки, образовавшие цикл в графе. Вы не увидите ни других транзакций, которые, возможно, вносят свой вклад в цикл, ни команды, которая в действительности захватила блокировки, если она была не последней. Тем не

менее изучение этой распечатки обычно помогает выявить причину взаимоблокировки.

На самом деле в InnoDB есть две разновидности взаимоблокировок. Первая, знакомая многим, связана с наличием реального цикла в графе ожиданий. Вторая возникает, когда граф ожиданий настолько велик, что искать в нем циклы слишком дорого. Если требуется проверить граф, содержащий более миллиона блокировок, или во время проверки транзакций глубина рекурсии превысит 200, то InnoDB сдается и считает, что взаимоблокировка имеется. Эти константы «зашиты» в код InnoDB и не конфигурируются (хотя при желании их можно изменить, перекомпилировав исходный код InnoDB). О наличии такой взаимоблокировки свидетельствует сообщение в распечатке «TOO DEEP OR LONG SEARCH IN THE LOCK TABLE WAITS-FOR GRAPH» (слишком глубокий или продолжительный поиск в графе ожиданий блокировок).

InnoDB выводит не только номера транзакций и блокировки, которые они удерживают или ожидают, но также и сами записи. Эта информация полезна, главным образом, разработчикам InnoDB, но в настоящее время отключить ее вывод невозможно. К сожалению, она может заполнить все место, отведенное для распечатки, так что вы не увидите последующие секции. Единственный способ как-то поправить ситуацию – сделать так, чтобы вместо большой взаимоблокировки возникала маленькая, или применить заплату, разработанную одним из авторов этой книги (ее можно скачать со страницы *http://lists.mysql.com/internals/35174*).

Ниже приведен пример сведений о взаимоблокировке:

```
------------------------
LATEST DETECTED DEADLOCK
------------------------
070913 11:14:21
*** (1) TRANSACTION:
TRANSACTION 0 3793488, ACTIVE 2 sec, process no 5488, OS thread id
1141287232 starting index read
mysql tables in use 1, locked 1
LOCK WAIT 4 lock struct(s), heap size 1216
MySQL thread id 11, query id 350 localhost baron Updating
UPDATE test.tiny_dl SET a = 0 WHERE a <> 0
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 3662 n bits 72 index `GEN_CLUST_INDEX` of
table `test/tiny_dl` trx id 0 3793488 lock_mode X waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format;
info bits 0
0: len 6; hex 000000000501 ...[ опущено ] ...

*** (2) TRANSACTION:
TRANSACTION 0 3793489, ACTIVE 2 sec, process no 5488, OS thread id
1141422400 starting index read, thread declared inside InnoDB 500
mysql tables in use 1, locked 1
```

```
19 4 lock struct(s), heap size 1216
20 MySQL thread id 12, query id 351 localhost baron Updating
21 UPDATE test.tiny_dl SET a = 1 WHERE a <> 1
22 *** (2) HOLDS THE LOCK(S):
23 RECORD LOCKS space id 0 page no 3662 n bits 72 index `GEN_CLUST_INDEX` of
   table `test/tiny_dl` trx id 0 3793489 lock mode S
24 Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format;
   info bits 0
25 0: ... [ опущено ] ...
26
27 *** (2) WAITING FOR THIS LOCK TO BE GRANTED:
28 RECORD LOCKS space id 0 page no 3662 n bits 72 index `GEN_CLUST_INDEX` of
   table `test/tiny_dl` trx id 0 3793489 lock_mode X waiting
29 Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format;
   info bits 0
30 0: len 6; hex 000000000501 ...[ опущено ] ...
31
32 *** WE ROLL BACK TRANSACTION (2)
```

В строке 4 показано, когда произошла взаимоблокировка, а в строках 5–10 приведены сведения о первой транзакции, участвующей в этой взаимоблокировке. Назначение отдельных полей мы объясним в следующем разделе.

В строках 11–15 показаны блокировки, которых ожидала транзакция 1 в момент, когда была обнаружена взаимоблокировка. Кое-какую информацию в строке 14, полезную только для отладки InnoDB, мы опустили. Обратите особое внимание на строку 12, в которой говорится, что транзакция запросила монопольную (X) блокировку на индекс `GEN_CLUST_INDEX`[1] по таблице `test.tiny_dl`.

В строках 16–21 показано состояние второй транзакции, а в строках 22–26 – список удерживаемых ей блокировок. В строке 25 было перечислено несколько записей, которые мы для краткости опустили. Одной из них была как раз запись, чьего освобождения ожидала первая транзакция. Наконец, в строках 27–31 показаны блокировки, которых ожидала вторая транзакция. Цикл в графе ожиданий образовался из-за того, что каждая транзакция удерживает блокировку, нужную другой транзакции. InnoDB не показывает все удерживаемые и ожидаемые блокировки, но выводит достаточно информации, чтобы можно было понять, какие индексы использовались при выполнении запросов. Это помогает решить, можно ли что-то сделать для предотвращения взаимоблокировок.

Если удастся добиться того, чтобы оба запроса просматривали один и тот же индекс в одном и том же направлении, то количество взаимоблокировок сократится, так как при запросе блокировок в одном и том же порядке цикл никогда не возникает. Иногда сделать это несложно. Например, если в транзакции нужно обновить несколько записей, то отсортируйте их по первичному ключу в памяти приложения и обнов-

[1] Этот индекс InnoDB создает самостоятельно, если не задан первичный ключ.

ляйте именно в таком порядке – взаимоблокировки не будет. Но в других случаях ничего не получится (например, когда два процесса работают с одной таблицей, используя при этом разные индексы).

В строке 32 показано, какая транзакция выбрана жертвой взаимоблокировки. InnoDB выбирает ту транзакцию, которую, с ее точки зрения, будет проще откатить, т. е. в которой количество обновлений наименьшее.

Эту информацию полезно отслеживать и записывать в журнал для последующего анализа. В этих целях удобен инструмент *mk-deadlock-logger* из комплекта Maatkit. Кроме того, очень помогает просмотреть общий журнал, найти в нем все запросы, выполненные в тех потоках, что участвовали в инциденте, и выяснить истинную причину взаимоблокировки. В следующем разделе рассказано о том, где искать идентификатор потока в распечатке сведений о взаимоблокировке.

Секция TRANSACTIONS

В этой секции содержится сводная информация о транзакциях InnoDB, за которой следует список активных транзакций. Вот первые несколько строчек (заголовок):

```
1 ------------
2 TRANSACTIONS
3 ------------
4 Trx id counter 0 80157601
5 Purge done for trx's n:o <0 80154573 undo n:o <0 0
6 History list length 6
7 Total number of lock structs in row lock hash table 0
```

Состав распечатки зависит от версии MySQL, но в любом случае имеется, по крайней мере, следующая информация.

- Строка 4: текущий идентификатор транзакции; это системная переменная, увеличивающаяся на единицу для каждой новой транзакции.
- Строка 5: идентификатор транзакции, для которой InnoDB удалила старые MVCC-версии строк. Узнать, сколько старых версий еще не удалено, можно, посмотрев на разность между этим значением и текущим идентификатором транзакции. Не существует четкого и однозначного правила, насколько большое значение этой величины можно считать безопасным. Если никто не обновляет никаких данных, то большое значение еще не означает, что скопилось много неудаленных версий, потому что все транзакции на самом деле видят одну и ту же версию базы. С другой стороны, если обновляется много строк, то одна или даже несколько версий каждой строки остается в памяти. Наилучшая стратегия сокращения издержек – фиксировать транзакции сразу после завершения работы, а не оставлять их открытыми на длительное время, поскольку даже если незафиксированная транзакция ничего не делает, она все равно не дает InnoDB удалить старые версии строк.

- Также в строке 5: номер записи в журнале отмены, с которым в настоящий момент работает процесс очистки InnoDB. Если он равен "0 0", как в данном примере, значит, процесс очистки простаивает.
- Строка 6: длина списка истории, то есть количество еще не удаленных транзакций в пространстве отмены в файлах данных InnoDB. Когда транзакция производит обновление и фиксируется, это число увеличивается. Как только процесс очистки удаляет старые версии, оно уменьшается. Процедура очистки также обновляет значение, показанное в строке 5.
- Строка 7: количество lock-структур. В каждой lock-структуре обычно хранится несколько блокировок уровня строки, так что это не то же самое, что количество заблокированных строк.

После заголовка идет список транзакций. В текущих версиях MySQL вложенные транзакции не поддерживаются, поэтому в каждый момент времени в рамках одного соединения может существовать не более одной транзакции, и каждая транзакция принадлежит ровно одному соединению. Под каждую транзакцию в этой распечатке отведено, по меньшей мере, две строчки. Вот пример минимальной информации о транзакции:

```
1 ---TRANSACTION 0 3793494, not started, process no 5488, OS thread id 1141152064
2 MySQL thread id 15, query id 479 localhost baron
```

Первая строчка начинается с идентификатора и состояния транзакции. Эта транзакция «не начата» (not started), то есть зафиксирована и не инициировала никаких команд, которые могли бы повлиять на другие транзакции; скорее всего, она просто ничего не делает. Далее следует информация о процессе и потоке. Во второй строчке показан идентификатор процесса MySQL; это то же самое число, что демонстрируется в колонке `Id` в списке процессов, выдаваемых командой `SHOW FULL PROCESSLIST`. Затем идет внутренний номер запроса и сведения о подключении (также печатаемые командой `SHOW FULL PROCESSLIST`).

Однако о транзакции может быть напечатано гораздо больше информации. Вот более сложный пример:

```
1 ---TRANSACTION 0 80157600, ACTIVE 4 sec, process no 3396, OS thread id
  1148250464, thread declared inside InnoDB 442
2 mysql tables in use 1, locked 0
3 MySQL thread id 8079, query id 728899 localhost baron Sending data
4 select sql_calc_found_rows * from b limit 5
5 Trx read view will not see trx with id>= 0 80157601, sees <0 80157597
```

Из строки 1 видно, что транзакция была активна четыре секунды. Возможны следующие состояния: «not started» (не начата), «active» (активна), «prepared» (подготовлена) и «committed in memory» (зафиксирована в памяти) – после того как транзакция зафиксируется на диске, она переходит в состояние «not started». Дополнительно может быть выведена информация о том, чем сейчас занята транзакция, хотя в дан-

ном примере этого нет. В исходном коде встречается свыше 30 строковых констант, которые могут быть напечатаны в этом месте, например: «fetching rows» (выборка строк), «adding foreign keys» (добавление внешних ключей) и т. д.

Фраза «thread declared inside InnoDB 442» (поток находится внутри InnoDB 442) в строке 1 означает, что поток занят какой-то операцией внутри ядра InnoDB, и у него осталось еще 442 неиспользованных «билета». Другими словами, этому SQL-запросу разрешено войти в ядро InnoDB еще 442 раза.

Система билетов ограничивает количество одновременно выполняющихся потоков внутри ядра с целью предотвратить пробуксовку потоков на некоторых платформах. Даже если показывается состояние потока «inside InnoDB», это еще не означает, что поток выполняет всю работу внутри подсистемы хранения; некоторые операции могут выполняться на уровне сервера и лишь временами тем или иным образом взаимодействовать с InnoDB. Иногда можно увидеть, что транзакция находится в состоянии «sleeping before joining InnoDB queue» (спит перед постановкой в очередь InnoDB» или «waiting in InnoDB queue» (ждет в очереди InnoDB).

В следующей строчке может находиться информация о том, сколько таблиц используется и заблокировано текущей командой. Обычно InnoDB не блокирует таблицы, но для некоторых команд это делается. Заблокированные таблицы могут появляться и тогда, когда сервер MySQL произвел блокировку на более высоком уровне, чем InnoDB. Если транзакция заблокировала какие-то строки, то будет присутствовать информация о количестве lock-структур (еще раз подчеркнем, что это не то же самое, что количество блокировок строк) и о размере кучи (heap size); пример был показан выше в распечатке сведений о взаимоблокировке.

Размер кучи – это объем памяти, отведенной для хранения блокировок строк. В InnoDB блокировки строк реализованы с помощью специальной таблицы битовых векторов, которая теоретически позволяет представлять блокировку всего одним битом. Наши тесты показывают, что в общем случае на блокировку отводится не более четырех битов.

Третья строка в этом примере содержит чуть больше информации, чем вторая строка в предыдущем: в конце мы видим состояние потока, «Sending data» (отправка данных). Это именно то, что показывает команда `SHOW FULL PROCESSLIST` в столбце `Command`.

Если транзакция в данный момент занята выполнением запроса, то далее будет приведен текст этого запроса (в некоторых версиях MySQL только часть его), как в данном случае в строке 4.

В строке 5 показано представление базы (read view), используемое транзакцией (подробнее см. описание секции `ROW OPERATIONS`), то есть диапазоны идентификаторов транзакций, которые наверняка видны и навер-

няка не видны данной транзакции благодаря механизму многоверсионности. В данном случае между двумя диапазонами имеется промежуток в четыре транзакции, относительно которых нельзя точно сказать, видны они или нет. При выполнении запроса InnoDB должна проверить, видны ли строки, для которых идентификаторы транзакций попадают в этот промежуток.

Если транзакция ожидает блокировки, то после запроса будет приведена информация об этой блокировке. Такие примеры встречаются в разделе о взаимоблокировках выше. К сожалению, распечатка ничего не говорит о том, какие транзакции *удерживают* те блокировки, которых ожидает данная транзакция.

Если транзакций много, то InnoDB может ограничить их количество в распечатке, чтобы объем вывода не оказался слишком велик. В таком случае вы увидите строку «...truncated...» (обрезано).

Секция FILE I/O

В секции FILE I/O представлены сведения о состоянии вспомогательных потоков ввода/вывода, а также счетчики производительности:

```
1 --------
2 FILE I/O
3 --------
4 I/O thread 0 state: waiting for i/o request (insert buffer thread)
5 I/O thread 1 state: waiting for i/o request (log thread)
6 I/O thread 2 state: waiting for i/o request (read thread)
7 I/O thread 3 state: waiting for i/o request (write thread)
8 Pending normal aio reads: 0, aio writes: 0,
9 ibuf aio reads: 0, log i/o's: 0, sync i/o's: 0
10 Pending flushes (fsync) log: 0; buffer pool: 0
11 17909940 OS file reads, 22088963 OS file writes, 1743764 OS fsyncs
12 0.20 reads/s, 16384 avg bytes/read, 5.00 writes/s, 0.80 fsyncs/s
```

В строках 4–7 приведена информация о состоянии вспомогательных потоков ввода/вывода. В строках 8–10 показано количество еще не завершенных операций для каждого такого потока, а также количество ожидающих процедур `fsync()` для потоков журнала и пула буферов. Аббревиатура «aio» означает «асинхронный ввод/вывод» (asynchronous I/O). В строке 11 демонстрируется количество выполненных операций чтения, записи и `fsync()`. За этими переменными полезно наблюдать с помощью какой-нибудь системы построения графиков и выявления трендов, например одной из упомянутых в начале следующей главы. Абсолютные значения зависят от рабочей нагрузки, поэтому важнее следить за тем, как они изменяются во времени. В строке 12 приведены значения количества операций в секунду, усредненные по интервалу, который указан в секции заголовка.

Значения «pending» (незавершенные) в строках 8 и 9 – удобный способ выявить приложения, занимающиеся преимущественно вводом/выво-

дом. В этом случае для большинства видов ввода/вывода будут присутствовать незавершенные операции.

На платформе Windows количество вспомогательных потоков ввода/вывода можно регулировать с помощью конфигурационной переменной `innodb_file_io_threads`, поэтому одновременно может присутствовать несколько потоков чтения и записи. Однако на любой платформе будут существовать, по крайней мере, следующие четыре потока:

Insert buffer thread (поток буфера вставки)

Отвечает за объединение с буфером вставки (то есть за перенос записей из буфера вставки в табличное пространство).

Log thread (поток журнала)

Отвечает за асинхронный сброс журнала на диск.

Read thread (поток чтения)

Выполняет операции упреждающего чтения для выборки тех данных, которые, как считает InnoDB, понадобятся в ближайшем будущем.

Write thread (поток записи)

Сбрасывает на диск «грязные» буферы.

Секция INSERT BUFFER AND ADAPTIVE HASH INDEX

В этой секции отображается состояние буфера вставки и адаптивного хеш-индекса:

```
1 -------------------------------------
2 INSERT BUFFER AND ADAPTIVE HASH INDEX
3 -------------------------------------
4 Ibuf for space 0: size 1, free list len 887, seg size 889, is not empty
5 Ibuf for space 0: size 1, free list len 887, seg size 889,
6 2431891 inserts, 2672643 merged recs, 1059730 merges
7 Hash table size 8850487, used cells 2381348, node heap has 4091 buffer(s)
8 2208.17 hash searches/s, 175.05 non-hash searches/s
```

В строке 4 приведена информация о размере буфера вставки, длина его «списка свободных» (free list) и размер сегмента. Фраза «for space 0» вроде бы указывает на возможность наличия нескольких буферов вставки – по одному на каждое табличное пространство, – но это так и не было реализовано, поэтому в последних версиях MySQL эта фраза удалена. Так как существует всего один буфер вставки, то строка 5 излишня. В строке 6 показана статистика работы с буфером. Отношение количества объединений (merges) к количеству вставок (inserts) дает представление об эффективности буфера.

В строке 7 демонстрируется состояние адаптивного хеш-индекса. Из строки 8 мы узнаем, сколько операций с этим индексом было произведено за время, указанное в секции заголовка. Отношение количества поисков по хеш-индексу (hash searches) к количеству поисков без ис-

пользования хеш-индекса (non-hash searches) – еще одна полезная метрика эффективности, так как поиск по хеш-индексу выполняется значительно быстрее. Но эти данные приведены только для справки, поскольку средств для конфигурирования адаптивного хеш-индекса нет.

Секция LOG

В этой секции собрана статистика о подсистеме работы с журналом транзакций в InnoDB:

```
1 ---
2 LOG
3 ---
4 Log sequence number 84 3000620880
5 Log flushed up to 84 3000611265
6 Last checkpoint at 84 2939889199
7 0 pending log writes, 0 pending chkp writes
8 14073669 log i/o's done, 10.90 log i/o's/second
```

В строке 4 показан текущий порядковый номер журнала, а в строке 5 – место, до которого журнал был сброшен на диск. Порядковый номер – это просто количество байтов, записанных в файлы журнала; зная его, можно вычислить, сколько данных в буфере журнала еще не сброшено в файлы. В данном случае эта величина равна 9615 байтов (13000620880 – 13000611265). В строке 6 показана последняя контрольная точка (она определяет момент времени, когда данные и файлы журналов находились в известном состоянии, которое пригодно для восстановления). В строках 7 и 8 отображается количество незавершенных операций с журналом и статистика, – ее можно сравнить с величинами в секции FILE I/O и узнать, какая доля ввода/вывода приходится на подсистему работы с журналом, а не обусловлена какими-то другими причинами.

Секция BUFFER POOL AND MEMORY

В этой секции приведена статистика использования пула буферов и памяти InnoDB (о настройке пула буферов см. главу 6).

```
 1 ----------------------
 2 BUFFER POOL AND MEMORY
 3 ----------------------
 4 Total memory allocated 4648979546; in additional pool allocated 16773888
 5 Buffer pool size 262144
 6 Free buffers 0
 7 Database pages 258053
 8 Modified db pages 37491
 9 Pending reads 0
10 Pending writes: LRU 0, flush list 0, single page 0
11 Pages read 57973114, created 251137, written 10761167
12 9.79 reads/s, 0.31 creates/s, 6.00 writes/s
13 Buffer pool hit rate 999 / 1000
```

В строке 4 демонстрируется общий объем памяти, выделенной InnoDB, и указано, какая часть этой памяти отведена под дополнительный пул.

В строках 5–8 приведены различные метрики пула буферов, измеряемые в страницах: размер пула, количество свободных страниц, количество страниц, отведенных для хранения страниц базы данных, и количество «грязных» страниц базы. Часть страниц в пуле буферов подсистема хранения использует для индексов блокировок, адаптивного хеш-индекса и других системных структур, поэтому количество страниц базы данных в пуле никогда не совпадает с общим размером пула.

В строках 9 и 10 показано число незавершенных операций чтения и записи (то есть операций, которые InnoDB еще предстоит выполнить в пуле буферов). Эти значения не совпадают со значениями в секции `FILE I/O`, так как InnoDB может объединять несколько логических процедур в одну физическую. Аббревиатура LRU означает «least recently used» (наиболее давно использованный); это алгоритм освобождения места для часто используемых страниц пула буферов за счет вытеснения и сброса на диск тех, что задействуются редко. Список сброса (flush list) содержит старые страницы, которые надлежит сохранить на диск при следующей контрольной точке, а метрика «запись одиночных страниц» (single page) относится к независимым операциям записи, которые не удалось объединить.

В строке 8 мы видим, что пул буферов включает 37491 грязную страницу, которые в какой-то момент нужно будет сбросить на диск (то есть они уже модифицированы в памяти, но еще не сохранены). Однако строка 10 говорит, что в данный момент операция сброса не запланирована. Ничего страшного, InnoDB сбросит их, когда ей это будет нужно.

В строке 11 показано, сколько страниц InnoDB прочитала, создала и записала. Количество прочитанных и записанных страниц относится к данным, которые были прочитаны, соответственно, с диска в пул буферов и наоборот. Созданные страницы – это те, для которых InnoDB выделила место в буфере, не читая их содержимое из файла данных, потому что это содержимое ей безразлично (например, они могли принадлежать таблице, которая уже удалена).

В строке 13 показан коэффициент попаданий в кэш, он говорит о том, как часто InnoDB находит нужные страницы в пуле буферов. Эта метрика характеризует эффективность кэширования. Она учитывает попадания с момента последней распечатки состояния InnoDB, поэтому если с тех пор сервер ничего не делал, то вы увидите сообщение: «No buffer pool page gets since the last printout» (с момента последней распечатки не было обращений к страницам из пула буферов). Из-за особенностей устройства InnoDB напрямую сравнивать коэффициенты попаданий в кэш пула буферов InnoDB и в буфер ключей MyISAM не имеет смысла.

Секция ROW OPERATIONS

В этой секции собрана информация об операциях со строками и прочая статистика InnoDB:

```
1 --------------
2 ROW OPERATIONS
3 --------------
4 0 queries inside InnoDB, 0 queries in queue
5 1 read views open inside InnoDB
6 Main thread process no. 10099, id 88021936, state: waiting for server activity
7 Number of rows inserted 143, updated 3000041, deleted 0, read 24865563
8 0.00 inserts/s, 0.00 updates/s, 0.00 deletes/s, 0.00 reads/s
9 ----------------------------
10 END OF INNODB MONITOR OUTPUT
11 ============================
```

В строке 4 показано, сколько потоков сейчас находится в ядре InnoDB (мы уже касались этой темы при обсуждении секции `TRANSACTIONS`). Запросы в очереди (queries in queue) – это те потоки InnoDB, которые пока не допущены в ядро, чтобы ограничить общее количество одновременно работающих потоков. Выше мы также отмечали, что перед постановкой в очередь запрос может некоторое время спать.

В строке 5 демонстрируется количество открытых в настоящий момент представлений базы (read view). Представление базы – это согласованный снимок многоверсионного содержимого базы данных на момент начала транзакции. Посмотреть, имеет ли конкретная транзакция представление базы, можно в секции `TRANSACTIONS`.

В строке 6 отображается состояние главного потока ядра. В версиях MySQL 5.0.45 и 5.1.22 оно может принимать следующие значения:

- `archiving log` (архивирование журналов (если режим архивирования журналов включен))
- `doing background drop tables` (фоновое удаление таблиц)
- `doing insert buffer merge` (объединение с буфером вставки)
- `flushing buffer pool pages` (сброс страниц из пула буферов)
- `flushing log` (сброс журнала)
- `making checkpoint` (запись контрольной точки)
- `purging` (очистка)
- `reserving kernel mutex` (резервирование мьютекса ядра)
- `sleeping` (сон)
- `suspending` (приостановка)
- `waiting for buffer pool flush to end` (ожидание завершения сброса пула буферов)
- `waiting for server activity` (ожидание действия со стороны сервера).

В строках 7 и 8 приведена статистика количества вставленных, обновленных, удаленных и прочитанных строк, а также усредненные значения числа этих операций в секунду. За ними имеет смысл наблюдать, если вы хотите знать, чем занимается InnoDB.

Распечатка, выдаваемая командой `SHOW INNODB STATUS`, заканчивается строками 9–11. Если вы их не видите, то, скорее всего, имела место большая взаимоблокировка, из-за которой пришлось обрезать распечатку.

Команда SHOW PROCESSLIST

Список процессов – это список соединений, или потоков, установленных с MySQL в данный момент времени. Команда `SHOW PROCESSLIST` выводит перечень всех потоков, включая в него сведения об их состоянии.

```
mysql> SHOW FULL PROCESSLIST\G
*************************** 1. row ***************************
     Id: 61539
   User: sphinx
   Host: se02:58392
     db: art136
Command: Query
   Time: 0
  State: Sending data
   Info: SELECT a.id id, a.site_id site_id, unix_timestamp(inserted) AS
inserted,forum_id, unix_timestamp(p

*************************** 2. row ***************************
     Id: 65094
   User: mailboxer
   Host: db01:59659
     db: link84
Command: Killed
   Time: 12931
  State: end
   Info: update link84.link_in84 set url_to =
replace(replace(url_to,'&amp;','&'),'%20','+'), url_prefix=repl
```

Существует несколько инструментов (в том числе *innotop*), умеющих показывать список процессов в динамике.

В столбцах `Command` и `State` отображается собственно «состояние» потока. Обратите внимание, что первый процесс выполняет запрос и отправляет данные, тогда как второй был прерван (Killed), возможно потому, что работал слишком долго, и кто-то сознательно остановил его командой `KILL`. Поток может оставаться в этом состоянии в течение определенного периода, потому что прерывание не происходит мгновенно. Например, для отката начатой в потоке транзакции требуется время.

Команда `SHOW FULL PROCESSLIST` (с дополнительным ключом `FULL`) показывает полный текст запроса, который в противном случае усекается до 100 символов.

Команда SHOW MUTEX STATUS

Эта команда возвращает подробную информацию о мьютексах InnoDB и полезна главным образом для разрешения проблем с масштабируемостью и параллелизмом. Как уже объяснялось, каждый мьютекс защищает одну критическую секцию программы.

Состав распечатки зависит от версии MySQL и опций, заданных при компиляции. Иногда выводятся имена мьютексов и несколько столбцов для каждого, а иногда только имя файла, номер строки в нем и номер мьютекса. Для агрегирования выходной информации, которая может оказаться очень объемной, имеет смысл написать сценарий. Ниже представлена одна строка распечатки:

```
*************************** 1. row ***************************
        Mutex: &(buf_pool->mutex)
       Module: buf0buf.c
        Count: 95
   Spin_waits: 0
  Spin_rounds: 0
     OS_waits: 0
    OS_yields: 0
OS_waits_time: 0
```

Проанализировав распечатку, можно понять, какие части InnoDB являются узкими местами. Например, сложности могут возникнуть из-за большого количества процессоров. В последних версиях MySQL были решены многие вопросы масштабируемости InnoDB в системах с несколькими процессорами, но кое-какие проблемы с мьютексами еще остались. К наиболее распространенным относятся блокировки `AUTO_INCREMENT`, которые ставятся на всю таблицу и защищены мьютексом, а также буфер вставки. Всюду, где есть мьютекс, существует опасность конкуренции.

Перечислим столбцы выводимой таблицы:

`Mutex`

Имя мьютекса.

`Module`

Имя исходного файла, в котором определен мьютекс.

`Count`

Сколько раз запрашивался мьютекс.

`Spin_waits`

Сколько раз InnoDB выбирала активное ожидание освобождения мьютекса. Напомним, что InnoDB сначала пытается немного подождать, выполняя цикл активного ожидания, и только если это не принесло успеха, обращается к ожиданию на уровне операционной системы.

Spin_rounds

Сколько раз InnoDB проверяла, свободен ли мьютекс в циклах активного ожидания.

OS_waits

Сколько раз InnoDB обращалась к ожиданию на уровне операционной системы.

OS_yields

Сколько раз поток, ожидающий мьютекса, уступал управление операционной системе, чтобы дать возможность поработать другому потоку.

OS_waits_time

Если системная переменная timed_mutexes установлена в 1, то здесь выводится количество миллисекунд, потраченных на ожидание.

Сравнивая значения этих счетчиков, можно найти, где происходят «горячие точки». Существует три основных способа избежать узких мест: держаться подальше от известных слабостей InnoDB, ограничить степень конкуренции и постараться отыскать компромисс между активным ожиданием, нагружающим процессор, и ожиданием на уровне ОС, потребляющим ресурсы. Дополнительные рекомендации см. в разделе «Настройка конкурентного доступа для InnoDB» главы 6 на стр. 370.

Состояние репликации

В MySQL есть несколько команд для мониторинга репликации. Выполнение команды SHOW MASTER STATUS на главном сервере показывает его состояние и конфигурацию репликации:

```
mysql> SHOW MASTER STATUS\G
*************************** 1. row ***************************
            File: mysql-bin.000079
        Position: 13847
    Binlog_Do_DB:
Binlog_Ignore_DB:
```

Выводится текущая позиция в двоичном журнале главного сервера. Получить список всех двоичных журналов вам позволит команда SHOW BINARY LOGS:

```
mysql> SHOW BINARY LOGS
+------------------+-----------+
| Log_name         | File_size |
+------------------+-----------+
| mysql-bin.000044 |     13677 |
...
| mysql-bin.000079 |     13847 |
+------------------+-----------+
36 rows in set (0.18 sec)
```

Чтобы просмотреть события в двоичных журналах, выполните команду `SHOW BINLOG EVENTS`.

Получить состояние и конфигурацию подчиненного сервера позволяет команда `SHOW SLAVE STATUS`. Мы не стали включать выдаваемую ей распечатку, потому что она довольно длинная, но сделаем на ее счет несколько замечаний. Во-первых, показывается состояние потока ввода/вывода и всех потоков SQL на подчиненном сервере, включая любые ошибки. Кроме того, видно, насколько далеко подчиненный сервер отстал от главного. И, наконец, для резервного копирования и клонирования подчиненных серверов печатается три набора координат в двоичном журнале:

`Master_Log_File/Read_Master_Log_Pos`

Позиция, с которой поток ввода/вывода читает двоичные журналы главного сервера.

`Relay_Log_File/Relay_Log_Pos`

Позиция команды в журнале ретрансляции подчиненного сервера, которую исполняет поток SQL.

`Relay_Master_Log_File/Exec_Master_Log_Pos`

Позиция команды в двоичном журнале главного сервера, которую исполняет поток SQL. Логически это та же позиция, что обозначается парой `Relay_Log_File/Relay_Log_Pos`, но только в журналах главного, а не подчиненного сервера. Иными словами, в указанной позиции в обоих журналах находится одно и то же событие.

База данных INFORMATION_SCHEMA

База данных `INFORMATION_SCHEMA` представляет собой набор системных представлений и соответствует стандарту SQL. В MySQL реализовано значительное число описанных в стандарте представлений, а также добавлены свои. В версии MySQL 5.1 многим представлениям соответствуют команды `SHOW`, например `SHOW FULL PROCESSLIST` и `SHOW STATUS`. Но есть и такие, для которых соответствующих команд `SHOW` не существует.

Прелесть представлений из базы данных `INFORMATION_SCHEMA` заключается в том, что их можно опрашивать с помощью стандартных команд SQL. В результате достигается куда большая гибкость, чем при использовании команд `SHOW`, порождающих результирующие наборы, которые невозможно агрегировать, соединять и вообще как-то манипулировать ими с использованием обычных средств SQL. Наличие данных в системных представлениях дает возможность составлять очень интересные и полезные запросы.

Например, какие таблицы в демонстрационной базе данных Sakila ссылаются на таблицу `actor`? При наличии последовательного соглашения об именовании определить это совсем нетрудно:

```
mysql> SELECT TABLE_NAME FROM INFORMATION_SCHEMA.COLUMNS
    -> WHERE TABLE_SCHEMA='sakila' AND COLUMN_NAME='actor_id'
    -> AND TABLE_NAME <> 'actor';
+------------+
| TABLE_NAME |
+------------+
| actor_info |
| film_actor |
+------------+
```

Для нескольких примеров из этой книги нам нужно было найти индексы по нескольким столбцам. Вот соответствующий запрос:

```
mysql> SELECT TABLE_NAME, GROUP_CONCAT(COLUMN_NAME)
    -> FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
    -> WHERE TABLE_SCHEMA='sakila'
    -> GROUP BY TABLE_NAME, CONSTRAINT_NAME
    -> HAVING COUNT(*) > 1;
+---------------+--------------------------------------+
| TABLE_NAME    | GROUP_CONCAT(COLUMN_NAME)            |
+---------------+--------------------------------------+
| film_actor    | actor_id,film_id                     |
| film_category | film_id,category_id                  |
| rental        | customer_id,rental_date,inventory_id |
+---------------+--------------------------------------+
```

Можно писать и более сложные запросы, точно так же, как для обычных таблиц. Сайт MySQL Forge (*http://forge.mysql.com*) – отличное место для поиска запросов к этим представлениям; там же вы можете поделиться и своими идеями. Имеются примеры запросов для поиска дублирующих или избыточных индексов, поиска индексов с очень низкой избирательностью и многое, многое другое.

Основной недостаток заключается в том, что некоторые представления работают очень медленно по сравнению с соответствующими командами `SHOW`. Как правило, они отбирают все данные, сохраняют их во временной таблице, а затем делают эту таблицу доступной для запроса. При реализации многих задач мониторинга, поиска неполадок и настройки быстрее просто набрать команду `SHOW`, нежели вводить полный SQL-запрос для выборки данных из представлений.

Кроме того, в текущей версии эти представления не обновляемы. Хотя они позволяют извлечь параметры сервера, обновить их так, чтобы изменилось его поведение, не удастся. На практике эти ограничения означают, что для конфигурирования сервера по-прежнему необходимо пользоваться командами `SHOW` и `SET`, пусть даже представления из базы `INFORMATION_SCHEMA` очень удобны для других целей.

14

Инструменты для оптимизации производительности

В дистрибутив сервера MySQL не входят инструменты для решения многих типичных задач, например для мониторинга сервера или сравнения данных, хранящихся на разных серверах. К счастью, сообщество, сложившееся вокруг MySQL, разработало множество полезных утилит, избавляя вас от необходимости придумывать собственные. Кроме того, многие компании предлагают коммерческие альтернативы или дополнения к инструментам, входящим в состав MySQL.

В этой главе мы рассмотрим некоторые из наиболее популярных и важных приложений, повышающих продуктивность работы с MySQL. Мы разобьем их на четыре категории: интерфейс, мониторинг, анализ и утилиты.

Средства организации интерфейса

Инструменты из этой категории помогают запускать запросы, создавать таблицы и новых пользователей, а также решать другие рутинные задачи. В этом разделе мы приведем краткий перечень наиболее популярных приложений такого рода. Как правило, все или большинство их функций можно реализовать с помощью запросов или SQL-команд; они лишь помогают работать быстрее и с большим комфортом, допуская меньше ошибок.

MySQL Visual Tools

Компания MySQL AB распространяет набор визуальных инструментов, включающий следующие программы: MySQL Query Browser, MySQL Administrator, MySQL Migration Toolkit и MySQL Workbench. Все они бесплатны и могут быть скачаны и установлены единым пакетом. Существуют версии для всех популярных операционных систем, исполь-

зуемых на ПК. Раньше данные инструменты изобиловали раздражающими недостатками, но недавно MySQL AB приложила серьезные усилия для поиска и устранения ошибок.

Программа MySQL Query Browser предназначена для запуска запросов, создания таблиц и хранимых процедур, экспорта данных и просмотра структуры базы данных. В нее интегрирована документация по SQL-командам и функциям MySQL. Особенно полезна она тем, кто разрабатывает и выполняет запросы к базам данных MySQL.

Программа MySQL Administrator ориентирована, прежде всего, на управление сервером и потому наиболее полезна для администраторов, а не разработчиков и аналитиков. Она помогает автоматизировать такие задачи, как создание резервных копий, регистрация пользователей и назначение им привилегий, а также просмотр журналов сервера и информации о его состоянии. В нее включены некоторые простые средства мониторинга, например, графическое отображение переменных состояния, но они не такие гибкие, как в интерактивных инструментах мониторинга, которые описываются ниже. Кроме того, она не сохраняет статистику для последующего анализа, хотя многие другие инструменты считают это само собой разумеющимся.

В пакет включен также комплект приложений MySQL Migration Toolkit, который позволяет переносить базы данных с других СУБД на MySQL, и программу моделирования MySQL Workbench.

Достоинствами инструментов, разработанных самой компанией MySQL AB, являются бесплатность, вполне приличное качество и возможность работы в большинстве операционных систем для ПК. Их функциональность хотя и проста, но достаточна для решения многих задач. Особенно хорошо реализованы управление пользователями и средства резервного копирования в MySQL Administrator и интегрированная документация в MySQL Query Browser.

Основной недостаток состоит в том, что эти инструменты все же не слишком изощренны, им не хватает всех тех «фенечек», к которым привыкли опытные пользователи и без которых они не желают обходиться. Полное описание всех утилит, включающее снимки с экрана, имеется на сайте MySQL по адресу *http://www.mysql.com/products/tools/*.

Программа MySQL Workbench недавно была переписана с нуля и теперь существует бесплатная и коммерческая ее версия. Нельзя сказать, что функциональность бесплатной версии сильно урезана, но для коммерческой доступны некоторые подключаемые модули, которые помогают автоматизировать задачи, требующие большого количества ручного труда. На момент работы над этой книгой инструмент MySQL Workbench tool все еще находился на уровне бета-версии.

SQLyog

SQLyog – самый популярный визуальный инструмент для MySQL. Он отлично спроектирован для повышения продуктивности как администратора, так и разработчика. Список функций слишком велик, чтобы приводить его полностью, но некоторые мы все же упомянем:

- Автозавершение кода, ускоряющее составление запросов
- Возможность подключаться к удаленным серверам по SSH-туннелю
- Визуальные инструменты и мастера, помогающие решать типичные задачи, например построение запросов
- Возможность планировать такие операции, как резервное копирование, импорт и синхронизация данных
- Назначение комбинаций клавиш
- Сравнение схем, позволяющее получить доступ к свойствам объектов, например таблиц и представлений
- Управление пользователями

SQLyog обладает всеми функциями, которых ожидают от программы такого рода, в частности имеется редактор схемы. Реализован он только на платформе Microsoft Windows, причем полная версия платная, а издание с ограниченной функциональностью распространяется бесплатно. Дополнительную информацию о программе SQLyog см. на сайте *http://www.webyog.com*.

phpMyAdmin

phpMyAdmin – это популярный инструмент администрирования, который работает на веб-сервере и предоставляет веб-интерфейс к серверам MySQL. Он обладает рядом полезных функций для выполнения запросов и администрирования баз данных. Основные достоинства – независимость от платформы, широкий набор функций и доступ через броузер. Последнее удобно, если вы находитесь далеко от привычной среды, и броузер – все, что у вас есть под рукой. Например, можно установить phpMyAdmin на серверах, к которым у вас есть только FTP-доступ, поэтому запустить клиента *mysql* или другую программу из оболочки не представляется возможным.

phpMyAdmin, безусловно, удобный инструмент, который во многих ситуациях оказывается как раз тем, что нужно. Но будьте очень осторожны при установке его на систему, доступную из веб, поскольку в случае, если ваш сервер не защищен надлежащим образом, то вы предоставите противнику такой вход, о котором можно только мечтать.

Противники phpMyAdmin считают, что у этого продукта слишком много функций и что он чересчур громоздкий и сложный. Страница разработчиков phpMyAdmin размещена на сайте SourceForge.net, где устой-

чиво занимает одно из верхних мест в рейтинге проектов. Дополнительную информацию можно найти по адресу *http://sourceforge.net/projects/phpmyadmin/*.

Инструменты мониторинга

Мониторинг MySQL – тема, заслуживающая отдельной книги; это большая и сложная задача, причем разные приложения зачастую предъявляют различные требования. Однако мы можем рассказать о некоторых наиболее полезных инструментах и ресурсах, посвященных этой тематике.

Слово «мониторинг» слишком многозначно. Многие употребляют его, подразумевая, что собеседник знает, о чем идет речь. Но наш опыт показывает, что в большинстве центров, где используется MySQL, имеется необходимость в различных видах мониторинга.

Мы уделим внимание инструментам как для интерактивного, так и для неинтерактивного мониторинга. Неинтерактивный мониторинг обычно подразумевает наличие автоматизированной системы, которая собирает результаты измерений и может уведомить администратора о том, что некоторый параметр вышел за пределы безопасного диапазона. Инструменты интерактивного мониторинга позволяют наблюдать за работой сервера в режиме реального времени. Каждая категория будет далее описана в отдельном разделе.

Вас могут заинтересовать и другие различия между инструментами, например между пассивным (примером может служить *innotop*) и активным мониторингом. В последнем случае инструмент может посылать оповещения или инициировать некоторые действия (например, Nagios). А быть может, вы ищете утилиту, которая создает хранилище информации, а не просто отображает текущую статистику. По ходу изложения мы будем отмечать и такие детали.

Неинтерактивные системы мониторинга

Многие системы мониторинга разрабатывались не специально для сервера MySQL server, а являются программами общего назначения, которые предназначены для периодической проверки состояния разнообразных ресурсов – от компьютеров до маршрутизаторов и программного обеспечения (в частности, MySQL). Обычно в основе их архитектуры лежит модульный принцип, а в состав дистрибутива входят уже готовые компоненты для MySQL. Некоторые продукты такого рода могут протоколировать состояние наблюдаемых систем и графически представлять его в веб-интерфейсе. Многие способны также отправлять оповещения или инициировать то или иное действие, когда наблюдаемая система выходит из строя или ее параметры оказываются за пределами безопасного диапазона.

Обычно подобный продукт устанавливается на отдельном сервере и применяется для наблюдения за другими серверами. Если он используется для мониторинга важных систем, то быстро становится неотъемлемой частью инфраструктуры, поэтому приходится предпринимать дополнительные меры, например, резервировать его и организовывать аварийное переключение при отказе.

Автоматизированная система мониторинга, которая сохраняет историю и показывает тренды, может стать спасением, когда производительность MySQL падает из-за возросшей нагрузки или возникают какие-то другие проблемы. Чтобы исправить ошибку, зачастую нужно знать, что изменилось, а для этого необходима история сервера и, следовательно, записи об этой истории. Система, оповещающая администратора о том, что что-то пошло не так, может предупредить еще до наступления катастрофы, а также подсказать направление поиска причин, если она все-таки произошла.

Доморощенные системы

Во многих организациях пытаются создавать собственные механизмы мониторинга и оповещения. Обычно они хорошо работают, пока количество наблюдаемых систем невелико, и в процессе участвует всего несколько человек. Но по мере роста и усложнения структуры организации и расширения штата администраторов доморощенные инструменты перестают работать. Они могут наводнять почтовые ящики тысячами писем при каждом нарушении работоспособности сети или молча отказывать, не оповещая никого о критической ситуации. Повторные и избыточные оповещения – проблема, от которой страдают многие самодельные системы, и это может стать препятствием к продуктивной работе.

Если вы подумываете о том, чтобы написать инструмент мониторинга самостоятельно, – пусть даже такой простой, как задание *cron*, которое делает запрос и отправляет по почте сообщение, если обнаружилась проблема, – не спешите и подумайте еще разок. Быть может, лучше потратить силы и время на изучение какой-то из систем, упомянутых в последующих разделах. Хотя придется освоить достаточно большой объем информации, и может показаться, что первоначальные затраты не оправдываются, в долгосрочной перспективе все затраченные усилия окупятся, и ваша организация будет чувствовать себя лучше. Установка подобной системы, даже если на первых порах она проведена плохо, в конечном итоге окажется предпочтительнее реализации собственной. И уж в любом случае вы приобретете опыт и знания, необходимые для использования стандартной системы мониторинга.

Nagios

Nagios (*http://www.nagios.org*) – система мониторинга и оповещения с открытым исходным кодом, которая периодически проверяет заданные службы и сравнивает результаты с определенными явно или по

умолчанию лимитами. Если результаты выходят за пределы заданного диапазона, то Nagios может запустить программу и/или оповестить кого-то о неполадках. Система регистрации ответственных лиц и оповещений, реализованная в Nagios, позволяет эскалировать тревогу, направляя оповещение другому лицу, изменять уведомления или отправлять их в разные места в зависимости от времени суток и других условий, а также учитывать запланированное время вывода системы из эксплуатации. Nagios также понимает зависимости между службами, поэтому не будет беспокоить вас сообщением об остановке MySQL, если замечает, что сервер недоступен из-за того, что вышел из строя маршрутизатор на пути к нему, или если обнаруживает, что остановлен сам сервер, на котором работает MySQL.

Nagios может запускать любой исполняемый файл как подключаемый модуль при условии, что он принимает определенные параметры и генерирует ожидаемый результат. Поэтому дополнительные компоненты для Nagios пишутся на разных языках, включая язык оболочки, Perl, Python, Ruby и другие языки сценариев. Существует даже сайт *http://www.nagiosexchange.org*, посвященный обмену подключаемыми модулями, которые разбиты на категории. И если не удается найти модуль, который делает точно то, что вам надо, то совсем несложно написать свой собственный. От него требуется лишь умение принимать стандартные аргументы, завершаться с определенным кодом состояния и, возможно, выводить текст, который Nagios сможет перехватить.

Nagios в состоянии вести мониторинг практически всего, что поддается измерению, и работает во многих операционных системах, применяя различные методы (в том числе активные проверки, дистанционно запускаемые поключаемые модули и пассивные проверки, когда просто принимаются данные о состоянии, «выталкиваемые» из других систем). У нее имеется и веб-интерфейс, позволяющий проверять статус, строить графики, визуализировать сеть и ее состояние, планировать штатные отключения и делать еще много чего.

Основной недостаток Nagios – ошеломляющая сложность. Даже хорошо изучив эту систему, поддерживать ее достаточно трудно. К тому же вся конфигурационная информация хранится в файлах со специальным синтаксисом, в котором легко допустить ошибку, а модифицировать конфигурацию по мере развития и роста наблюдаемых систем – весьма трудоемкое занятие. Наконец, средства построения графиков, анализа трендов и визуализации ограничены. Nagios может хранить некоторые показатели производительности и другие сведения в базе данных MySQL и генерировать графики по этим данным, но все же в этом отношении она уступает в гибкости иным системам.

Системе Nagios посвящено несколько книг; нам нравится книга Вольфганга Барта (Wolfgang Barth) «Nagios System and Network Monitoring» (издательство No Starch Press).

Альтернативы Nagios

Хотя Nagios – самая популярная программа мониторинга и оповещения общего назначения[1], есть и еще несколько аналогичных систем с открытым исходным кодом.

Zenoss

Программа Zenoss написана на языке Python и имеет веб-интерфейс, который для повышения быстродействия и продуктивности построен на основе технологии Ajax. Она умеет автоматически обнаруживать ресурсы в сети и объединяет в одном унифицированном инструменте средства мониторинга, оповещения, анализа трендов, построения графиков и сохранения исторических данных. Zenoss использует протокол SNMP для сбора данных с удаленных компьютеров, но может работать и по протоколу SSH. Имеется также поддержка подключаемых модулей Nagios. Дополнительную информацию можно найти на сайте *http://www.zenoss.com*.

Hyperic HQ

Hyperic HQ – система мониторинга, написанная на языке Java; она, в большей степени, ориентирована на так называемый корпоративный мониторинг, чем другие системы этого класса. Как и Zenoss, она умеет автоматически обнаруживать ресурсы и поддерживает подключаемые модули Nagios, но архитектурно устроена иначе и несколько более тяжеловесна. Отвечает ли она вашим целям, зависит в основном от личных предпочтений и того, что вы собираетесь мониторить. Дополнительную информацию можно найти на сайте *http://www.hyperic.com*.

OpenNMS

Система OpenNMS написана на Java и вокруг нее сложилось активное сообщество разработчиков. Она обладает обычными средствами, такими как мониторинг и оповещение, к которым добавляет построение графиков и анализ трендов. При проектировании ставились следующие цели: высокая производительность и масштабируемость, автоматизация и гибкость. Как и Hyperic, она предназначена для мониторинга крупных, критически важных систем масштаба предприятия. Дополнительную информацию можно найти на сайте *http://www.opennms.org*.

Groundwork Open Source

Система Groundwork Open Source построена на базе Nagios и объединяет Nagios с несколькими другими инструментами, предоставляя портальный интерфейс. Быть может, правильнее всего описать

[1] Возможно, потому, что один раз намучившись с установкой и конфигурированием Nagios, вы уже никогда не захотите даже помыслить еще о какой-нибудь системе мониторинга.

ее как систему, которую можно было бы построить самостоятельно, будь вы экспертом в программах Nagios, Cacti и целом ряде других и располагай временем для их интеграции друг с другом. Дополнительную информацию можно найти на сайте *http://www.groundworkopensource.com*.

Zabbix

Zabbix – это система мониторинга с открым исходным кодом, которая во многих отношениях напоминает Nagios, но имеет и ряд существенных отличий. Например, вся конфигурационная информация и прочие данные хранятся в базе, а не в файлах. Кроме того, количество типов сохраняемых данных больше, чем в Nagios, поэтому она лучше производит анализ трендов и генерирует более качественные отчеты. Средства построения графиков и визуализации также значительно лучше, чем в Nagios, и многие считают, что ее проще конфигурировать и что она более гибкая. С другой стороны, сообщество пользователей Zabbix уже, чем у Nagios, а средства оповещения не настолько развиты. Дополнительную информацию можно найти на сайте *http://www.zabbix.com*.

Служба MySQL Monitoring and Advisory Service

Monitoring and Advisory Service – это система, разработанная самой компанией MySQL AB и предназначенная специально для мониторинга экземпляров MySQL. Она умеет также вести наблюдение за некоторыми существенными аспектами компьютера, на котором работает сервер MySQL. Исходный код системы закрыт, и поставляется она только в составе издания MySQL Enterprise.

Основное преимущество этой службы по сравнению с Nagios заключается в том, что она включает предопределенный набор правил, или, как их еще называют, «помощников», которые исследуют многочисленные аспекты производительности, состояния и конфигурации сервера. Кроме того, она предлагает решения для устранения обнаруженных проблем, а не просто оставляет администратору возможность гадать, что не так. В систему встроена хорошо продуманная инструментальная панель, на которой отображается информация о состоянии сразу всех наблюдаемых серверов.

Хотя в принципе возможно использовать для мониторинга той же статистики Nagios или любую другую систему, было бы весьма трудоемко написать все необходимые подключаемые модули и сконфигурировать Nagios для наблюдения за десятками метрик, которые в MySQL Monitoring and Advisory Service включены изначально.

Недостаток этого продукта заключается в том, что он не умеет вести наблюдение за остальной частью сети; он предназначен только для мониторинга MySQL. Кроме того, на каждую наблюдаемую систему необходимо установить специальный агент. Это претит некоторым админи-

страторам MySQL, которые хотят свести количество программного обеспечения, используемого на серверах, к абсолютному минимуму.

Дополнительная информация имеется на сайте *http://www.mysql.com/products/enterprise/advisors.html*.

MONyog

MONyog (*http://www.webyog.com*) – это облегченная система мониторинга без агентов, в которой применен совершенно другой подход, нежели в ранее описанных инструментах. Она предназначена для работы на пользовательском компьютере и запускает прослушиватель протокола HTTP на неиспользуемом порту. Вы можете указать этот порт броузеру и получать информацию о своих серверах MySQL в виде комбинации JavaScript и Flash. В основе реализации лежит интерпретатор JavaScript, а все конфигурирование производится с помощью объектной модели JavaScript.

MONyog является одновременно интерактивной и неинтерактивной системой, так что имеет смысл изучить ее возможности для обоих видов мониторинга.

Системы на базе RRDTool

Хотя, строго говоря, RRDTool (*http://www.rrdtool.org*) не является системой мониторинга, она достаточно важна, чтобы упомянуть ее здесь. Во многих организациях применяется тот или иной сценарий или программа – часто доморощенные – для извлечения информации и сохранения ее в циклической базе данных (round-robin database – RRD). RRD-файлы дают элегантное решение многих задач, в которых необходимо сохранять информацию и представлять ее в графическом виде. Они автоматически агрегируют поступающие данные, интерполируют отсутствующие значения, если входные сведения не поступают в ожидаемое время, и располагают мощным инструментарием для генерации красивых графиков. Имеется несколько систем на базе RRDTool.

Программа Multi Router Traffic Grapher, или MRTG (*http://oss.oetiker.ch/mrtg/*), представляет собой квинтэссенцию систем на базе RRDTool. В действительности она предназначена для протоколирования сетевого трафика, но может быть обобщена на протоколирование и визуализацию других данных.

Система Munin (*http://munin.projects.linpro.no*) собирает данные, передает их RRDTool, а затем генерирует по ним графики с различными уровнями детализации. На основе конфигурационных параметров она создает статические HTML-файлы, которые можно посмотреть в броузере и легко выявить тренды. Определить график нетрудно; требуется создать подключаемый сценарий, который выводит информацию в специальном синтаксисе. Munin интерпретирует эти данные как инструкции по построению графика. К числу недостатков Munin следует отне-

сти необходимость установки агента на каждую наблюдаемую систему, упрощенную конфигурацию – одну на все случаи жизни – и недостаточно гибкие средства построения графиков.

Cacti (*http://www.cacti.net*) – еще одна популярная система построения графиков и анализа трендов. Она извлекает данные из RRD-файлов, а затем обращается через PHP-интерфейс к RRDTool для построения графиков. Тот же интерфейс используется для конфигурирования и управления (конфигурационные параметры хранятся в базе данных MySQL). Система управляется шаблонами, определяемыми пользователем. Данные могут собираться по протоколу SNMP или с помощью написанных пользователем сценариев.

Система Cricket (*http://cricket.sourceforge.net*) похожа на Cacti и написана на языке Perl, но ее конфигурационные данные хранятся в файлах. Система Ganglia (*http://ganglia.sourceforge.net*) тоже аналогична Cacti, но предназначена для мониторинга кластеров и grid-систем, поэтому позволяет просматривать агрегированные сведения по нескольким серверам с возможностью детализировать отчет до уровня отдельного сервера. (Cacti и Cricket не умеют агрегировать данные.)

Все эти продукты можно использовать для сбора, сохранения, графического представления данных и генерации отчетов в системах на базе MySQL с разной степенью гибкости и для слегка различных целей. Всем им недостает по-настоящему легко настраиваемых средств для оповещения ответственного лица в случае возникновения неполадок, а у некоторых даже отсутствует само понятие «неполадки». Некоторые считают это достоинством, полагая, что лучше разделять задачи сбора данных, графического представления и оповещения; более того, Munin специально спроектирована так, чтобы в качестве оповещателя использовать Nagios. Однако другие рассматривают это как недостаток. Еще один минус – время и силы, которые нужно потратить на установку и конфигурирование системы, которая отвечает вашим требованиям, но не в полной мере.

И, наконец, следует учитывать и будущие потребности. RRD-файлы не позволяют опрашивать данные с помощью языка SQL или других стандартных средств и не могут вечно хранить информацию с высокой степенью детализации. Многие администраторы MySQL не готовы смириться с такими ограничениями и предпочитают хранить исторические сведения в реляционной базе данных. Многим АБД нужны также более гибкие и лучше поддающиеся настройке способы сбора данных, поэтому они склоняются к написанию собственных систем или модификации какой-нибудь существующей.

Считать ли систему на основе RRDTool подходящей в конкретных обстоятельствах, зависит от личного вкуса, от наличия квалифицированных специалистов и от требований, предъявляемых организацией.

Интерактивные инструменты

Под интерактивными понимаются такие инструменты, которые запускаются по запросу и непрерывно обновляют картину происходящего на сервере. Мы будем говорить в основном о программе *innotop* (*http://innotop.sourceforge.net*), но существуют и другие, например *mtop* (*http://mtop.sourceforge.net*), *mytop* (*http://jeremy.zawodny.com/mysql/mytop/*), а также некоторые клоны *mytop* с веб-интерфейсом.

innotop

Программу *innotop* написал Бэрон Шварц, один из авторов этой книги. Несмотря на свое название, она не ограничивается только мониторингом внутренней работы InnoDB. Хотя этот инструмент был создан под несомненным влиянием программы *mytop*, он предлагает гораздо более широкую функциональность. Он имеет много режимов для мониторинга различных внутренних механизмов MySQL, в том числе всей информации, которую выдает команда `SHOW INNODB STATUS` (*innotop* разбирает результат, выделяя из него отдельные компоненты). Предоставляется также возможность вести наблюдение сразу за несколькими экземплярами MySQL. Ко всему прочему, инструмент очень гибко конфигурируется и допускает расширение.

Перечислим некоторые его возможности (а также то, что он может показывать):

- Список всех текущих транзакций InnoDB
- Список выполняемых в текущий момент запросов
- Список текущих блокировок и ожиданий блокировок
- Сводная информация о состоянии сервера и переменных, отражающая относительные величины значений
- Режимы для вывода информации о внутреннем состоянии InnoDB, в частности о буферах, взаимоблокировках, ошибках внешнего ключа, вводе/выводе, операциях со строками, семафорах и прочем
- Мониторинг репликации, причем состояние главного и подчиненных серверов можно увидеть одновременно
- Режим для просмотра любых серверных переменных
- Функция группировки серверов позволяет удобно организовать их логическое расположение на экране
- Неинтерактивный режим для применения в командных сценариях

Установить *innotop* нетрудно. Можно сделать это из репозитория пакетов для вашей операционной системы или загрузить исходный код с сайта *http://innotop.sourceforge.net*, распаковать его и запустить стандартную процедуру `make install`:

```
perl Makefile.PL
make install
```

Установив программу, вызовите *innotop* из командной строки, и она проведет вас по всему процессу подключения к серверу MySQL. Поскольку *innotop* умеет читать конфигурационный файл *~/.my.cnf*, то, возможно, вам придется указать лишь имя компьютера, на котором работает сервер, и несколько раз нажать клавишу Enter. После подключения вы окажетесь в режиме T (InnoDB Transaction) и увидите список текущих транзакций InnoDB, показанный на рис. 14.1.

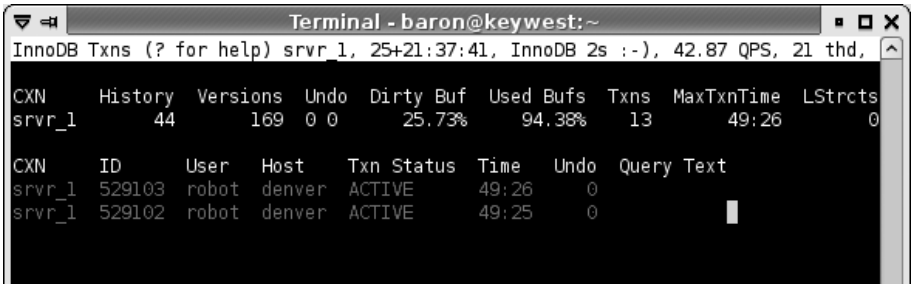


Рис. 14.1. *innotop в режиме T (InnoDB Transaction)*

По умолчанию *innotop* применяет фильтры, чтобы не загромождать экран (как и везде в *innotop*, вы можете писать собственные или модифицировать встроенные фильтры). На рис. 14.1 большинство транзакций отфильтровано и оставлено только пять активных. Чтобы отключить фильтр и вывести на экране столько транзакций, сколько поместится, нажмите клавишу `i`.

В этом режиме *innotop* отображает заголовок и главный список потоков. В заголовке показана сводная информация об InnoDB, например, длина списка истории, количество неудаленных транзакций, процентная доля грязных буферов в пуле и т. д.

Первым делом нажмите клавишу ?, чтобы посмотреть справку. Содержимое этого окна зависит от того, в каком режиме работает *innotop*, но в любом случае будет приведен список активных клавиш, чтобы можно было увидеть все доступные действия. На рис. 14.2 изображено окно справки в режиме T.

Мы не станем подробно рассматривать все остальные режимы, но, как легко понять, глядя на окно справки, возможности *innotop* весьма обширны. Единственное, на чем мы хотим еще остановиться, – это выполнение простенькой настройки с целью продемонстрировать, как можно следить именно за тем, что вам интересно. Одна из сильных сторон *innotop* – умение интерпретировать определенные пользователем выражения, например, выражение `Uptime/Questions` обозначает метрику «количество запросов в секунду». Можно вывести результат, рассчитанный за время с момента запуска сервера, и/или инкрементно с момента последнего опроса.

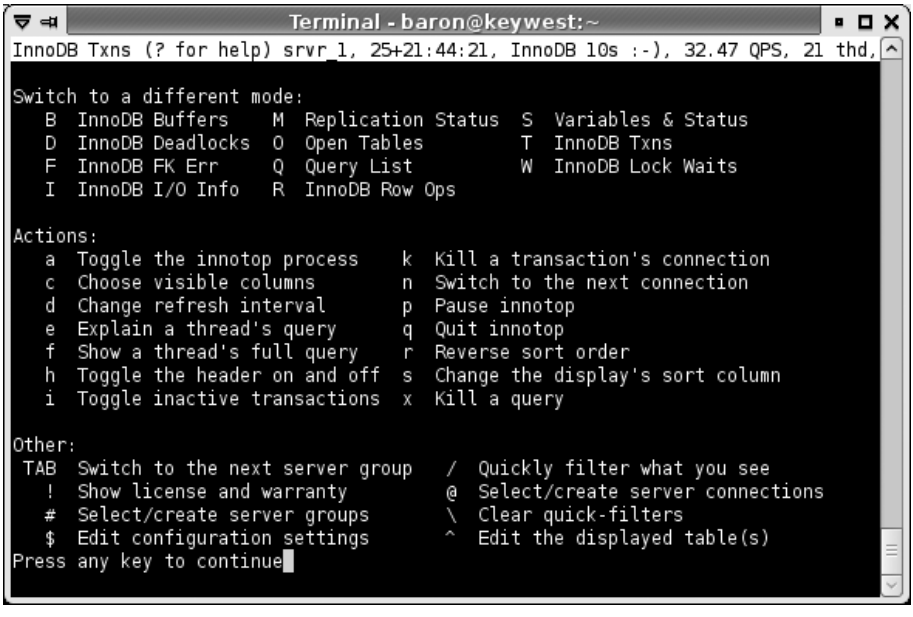


Рис. 14.2. *Окно справки innotop*

Таким способом можно добавлять собственные столбцы в отображаемые таблицы. Например, в режиме Q (Query List – список запросов) имеется заголовок, в котором демонстрируется сводная информация о сервере. Посмотрим, как модифицировать его, чтобы отображались сведения о заполненности кэша ключей. Запустите *innotop* и нажмите клавишу Q для входа в режим Q. Результат будет выглядеть, как показано на рис. 14.3.

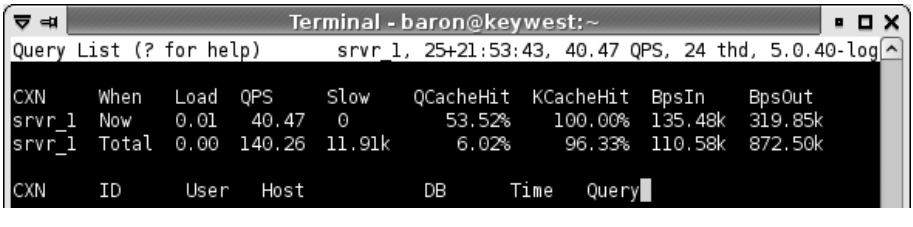


Рис. 14.3. *innotop в режиме Q (Query List)*

Этот снимок с экрана обрезан, поскольку нас в данном случае сам список запросов не интересует; нам важен только заголовок.

В заголовке показана статистика «Now» (измеряет инкрементную активность с момента последнего извлечения данных с сервера утилитой *innotop*) и «Total» (измеряет активность с момента запуска сервера MySQL 25 дней назад). Каждое поле в заголовке является результа-

том вычисления некоторого выражения, содержащего значения, которые возвращают команды SHOW STATUS и SHOW VARIABLES. Отображаемые по умолчанию поля, показанные на рис. 14.3, встроены, но легко добавить и свои собственные. Нужно лишь включить соответствующий столбец в «таблицу» заголовка. Нажмите клавишу ^, чтобы запустить редактор таблиц, а затем нажмите q в ответ на вопрос о том, какой заголовок вы собираетесь изменять (рис. 14.4). Механизм завершения по клавише Tab встроен, так что достаточно нажать клавишу q, а затем Tab – для завершения слова.

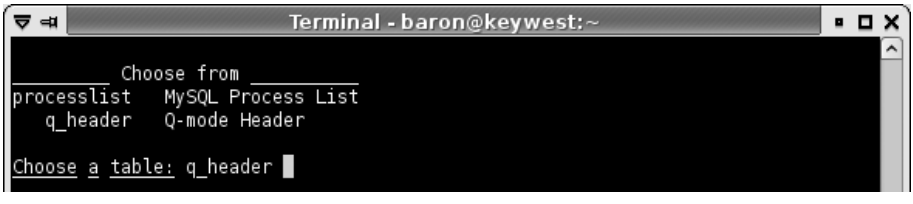


Рис. 14.4. *Редактирование заголовка (начало)*

После этого вы увидите определение заголовка в режиме Q (рис. 14.5). Определение представлено в виде таблицы, в которой выделен первый столбец. Разрешается перемещать выделение, переупорядочивать и редактировать столбцы, а также делать ряд других вещей (чтобы увидеть полный перечень, нажмите клавишу ?). Но сейчас мы хотим просто создать новый столбец. Нажмите клавишу n и наберите на клавиатуре имя столбца (рис. 14.6).

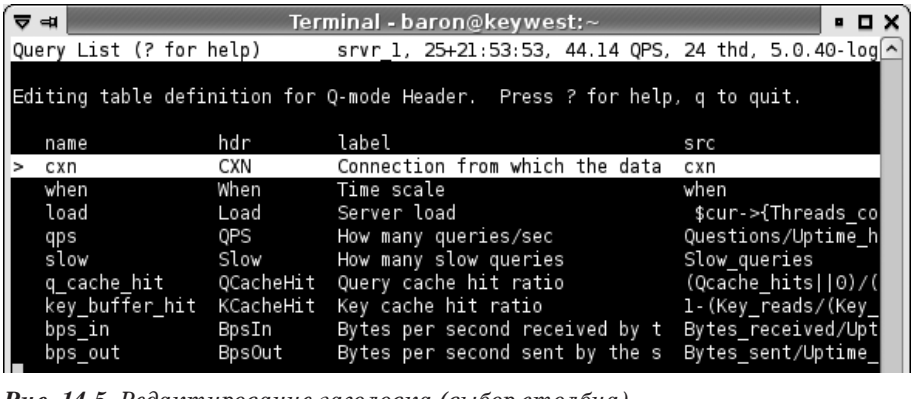


Рис. 14.5. *Редактирование заголовка (выбор столбца)*

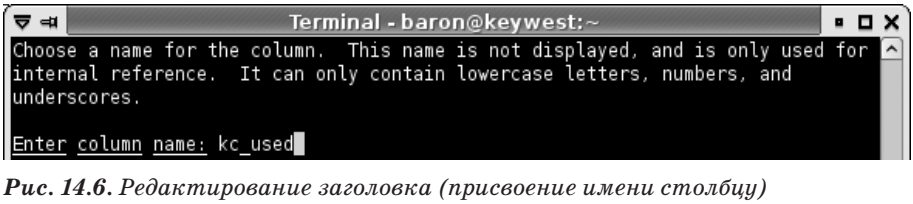


Рис. 14.6. *Редактирование заголовка (присвоение имени столбцу)*

Затем введите заголовок столбца (рис. 14.7) и, наконец, выберите источник данных для него. Это выражение, которое *innotop* компилирует во внутреннее представление в виде функции. В выражении можно использовать имена переменных, возвращаемых командами `SHOW VARIABLES` и `SHOW STATUS`. Мы добавили скобки и Perl-оператор «or», чтобы предотвратить деление на нуль, но в остальном это выражение не содержит ничего сложного. Мы также воспользовались встроенной в *innotop* функцией преобразования `percent()`, чтобы отформатировать результат в виде процента; дополнительную информацию см. в документации по *innotop*. Выражение показано на рис. 14.8.

Рис. 14.7. *Добавление заголовка (текста над столбцом)*

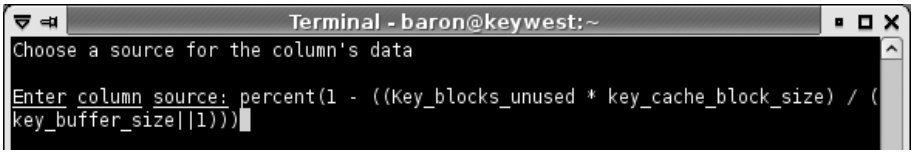


Рис. 14.8. *Добавление заголовка (вычисляемое выражение)*

Нажав Enter, вы увидите то же определение таблицы, что и прежде, но внизу появится описание добавленного столбца. Несколько раз нажмите клавишу +, чтобы поднять этот столбец выше в списке, а затем q, чтобы выйти из редактора. Готово: новый столбец расположился между `KCacheHit` и `BpsIn` (рис. 14.9). Как видите, настроить программу *innotop* так, чтобы она отслеживала именно то, что вам нужно, совсем нетрудно. Можно даже написать подключаемый модуль, если по-другому ничего не получается. Полная документация по *innotop* имеется на сайте *http://innotop.sourceforge.net*.

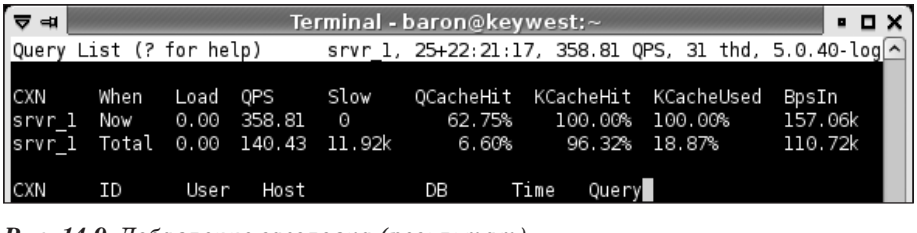


Рис. 14.9. *Добавление заголовка (результат)*

Инструменты анализа

Инструменты анализа помогают автоматизировать утомительную работу по поиску тех областей, оптимизация или настройка которых могла бы дать выигрыш с точки зрения производительности сервера. С применения подобных инструментов очень хорошо начинать исследование вопросов, относящихся к производительности. Если какой-нибудь из них сообщает об очевидной проблеме, то можно уделить ей основное внимание и, быть может, быстрее найти решение.

Комплект инструментов HackMySQL

Дэниэл Нихтер (Daniel Nichter) поддерживает сайт HackMySQL, на котором размещены некоторые полезные инструменты для работы с MySQL. Так, *mysqlreport* – это написанный на языке Perl сценарий, который разбирает результат команды `SHOW STATUS` и преобразует его в удобный для чтения отчет, который выводит в файл. Разобраться в этом достаточно полном отчете куда проще и быстрее, чем изучать то, что вывела команда `SHOW STATUS`.

Приведем обзор основных разделов отчета по состоянию на версию 3.23.

- В разделе «Key» показано, как используются ключи (индексы). Если присутствуют какие-то проблемы, то, вероятно, следует настроить параметры кэша ключей.
- В разделе «Questions» демонстрируется, какие типы запросов выполняет сервер; это дает представление о том, где сосредоточена основная нагрузка.
- В разделе «SELECT and Sort» показано, какие планы выполнения запросов и стратегии сортировки сервер применяет наиболее часто. Здесь же отражены проблемы, обусловленные неудачными индексами или плохо оптимизированными запросами.
- В разделе «Query Cache» отображаются сведения о том, насколько эффективно функционирует кэш запросов. Если эффективность низкая, то надо либо настроить параметры, либо, если рабочая нагрузка такова, что кэширование запросов не дает выигрыша, вовсе отключить кэш.
- В нескольких разделах приводятся сведения о таблицах, блокировках, соединениях и сетевом трафике. Проблемы здесь обычно свидетельствуют о том, что сервер плохо настроен.
- В трех разделах приведены метрики и настройки InnoDB. Обнаруженные здесь проблемы могут указывать на неудачно выбранные параметры сервера, на аппаратные ошибки либо на недостаточную оптимизацию схемы или отдельных запросов.

Дополнительная информация, в том числе и подробное пособие по интерпретации отчетов, имеется на сайте *http://hackmysql.com/mysqlreport*. Советуем потратить время на изучение того, как читать эти отчеты, особенно если вам часто приходится отлаживать незнакомые сервера. Попрактиковавшись, вы сможете выявлять проблемы, бросив лишь один мимолетный взгляд на отчет.

Еще одним полезным инструментом является программа *mysqlsla* (MySQL Statement Log Analyzer). С ее помощью можно анализировать журнал всех запросов, выполненных сервером, а также журнал медленных запросов (то есть тех, для которых время выполнения превысило заранее заданную величину) и любой другой. Она понимает целый ряд форматов журнала и может анализировать несколько журналов одновременно. Дополнительную информацию об анализе журналов MySQL см. в разделе «Тонкая настройка протоколирования» главы 2 на стр. 99.

На сайте есть и другие программы, помогающие изучить использование индексов и анализировать сетевой трафик, порождаемый MySQL.

Комплект инструментов анализа Maatkit

Комплект инструментов Maatkit – еще одно творение Бэрона Шварца (Baron Schwartz). По сути это комплект инструментов командной строки, которые предоставляют важную функциональность, отсутствующую в самих продуктах, поставляемых компанией MySQL AB. Скачать комплект, в который входят разнообразные приложения для анализа и прочие утилиты, можно с сайта *http://maatkit.sourceforge.net*.

Одним из инструментов анализа является сценарий *mk-query-profiler*, который умеет выполнять запросы, одновременно наблюдая за переменными состояния сервера. Он распечатывает подробный и понятный отчет о различиях переменных до и после запроса. Этот отчет дает более глубокое представление о том, как запрос влияет на производительность, чем можно было бы составить на основании одного лишь его выполнения.

Можно подавать запрос по конвейеру на стандартный вход *mk-query-profiler*, задавать один или несколько файлов запросов или просто указать режим наблюдения за сервером без выполнения запросов (это бывает полезно при запуске внешнего приложения). Можно также выполнять не запросы, а команды оболочки.

Отчет *mk-query-profiler* разбит на несколько разделов. По умолчанию профилировщик печатает сводную информацию по всем запросам в пакете, но может выдать отчет по каждому из них в отдельности или только по некоторым. Эти отчеты легко сравнить с помощью вспомогательного инструмента *mk-profile-compact*.

Перечислим основные разделы отчета:

- В разделе «Overall stats» приведены суммарные характеристики: время выполнения, количество команд и сетевой трафик.

- В разделе «Table and index accesses» показано, сколько различных планов выполнения породил обработанный пакет. Если количество операций полного сканирования таблиц велико, значит, не построены подходящие индексы.
- В разделе «Row operations» демонстрируется, сколько низкоуровневых операций над строками и/или операций InnoDB породил данный пакет. Если план выполнения неудачен, то низкоуровневых операций будет гораздо больше.
- В разделе «I/O operations» показано, сколько памяти и дискового трафика потребил данный пакет. В сопутствующем разделе приведены сведения об операциях с данными, специфичных для InnoDB.

Итак, этот отчет дает детальную информацию об объеме и характере работы, производимой сервером, а это куда ценнее, чем простое измерение скорости выполнения запроса. Например, он может помочь выбрать один из двух запросов, для которых время выполнения на небольшом наборе данных при низкой нагрузке примерно одинаково, но может сильно разниться, если данных много или нагрузка велика. Кроме того, отчет позволяет удостовериться в том, что проведенная оптимизация дала эффект. В каком-то смысле это миниатюрный инструмент эталонного тестирования.

В комплект входят и еще несколько инструментов анализа:

mk-visual-explain

Реконструирует план выполнения запроса по результатам команды `EXPLAIN` и отображает его в виде дерева, что с объективной точки зрения более удобно. Это особенно полезно для сложных планов; мы видели, что результат работы `EXPLAIN` может состоять из сотен строк, и при такой длине разобраться в нем практически невозможно. Программа *mk-visual-explain* хороша также для того, чтобы научиться читать результаты `EXPLAIN`.

mk-duplicate-key-checker

Находит дублирующие или избыточные индексы и внешние ключи, наличие которых может крайне негативно сказываться на производительности. Дополнительную информацию по этому поводу см. в разделе «Избыточные и дублирующие индексы» главы 3 на стр. 171.

mk-deadlock-logger

Наблюдает за взаимоблокировками InnoDB и заносит информацию о них в файл или в таблицу.

mk-heartbeat

Точно измеряет отставание репликации, позволяя обойтись без запуска команды `SHOW SLAVE STATUS` (которая не всегда выдает корректные результаты). По умолчанию вычисляется скользящее среднее за последние 1, 5 и 15 минут. Это более полная и лучше конфигурируемая

реализация сценария с использованием «сердцебиения» (heartbeat), который упоминался в первом издании книги.

Утилиты MySQL

Появился ряд инструментов, восполняющих отсутствующую в стандартном дистрибутиве MySQL функциональность. В этом разделе мы обсудим некоторые из них.

MySQL Proxy

Проект MySQL Proxy, разрабатываемый и поддерживаемый компанией MySQL AB, распространяется по лицензии GPL и, вероятно, в будущем будет включен в дистрибутив MySQL. На момент написания этой книги ему еще не исполнилось и года, и разработка велась очень активно[1]. Пока что найти проект можно в разделе Community на сайте *http://www.mysql.com*, а документация по нему включена в руководство по MySQL.

Основная идея MySQL Proxy состоит в том, что это приложение с запоминанием состояния (stateful application), которое понимает протокол MySQL и, располагаясь между клиентом и сервером, прозрачно транслирует сообщения протокола. Клиентское приложение может подключиться к нему так же, как и к обычному серверу. Затем прокси создает соединение с реальным сервером и действует в качестве посредника.

Уже одна лишь эта функциональность могла бы оказаться полезной многим приложениям (например, для балансирования нагрузки и переключения на резервный сервер при отказе), но прокси-сервер на этом не останавливается. Поскольку он знает все о протоколе взаимодействия, то может анализировать запросы и ответы. Кроме того, в него встроен интерпретатор языка Lua, на котором можно написать сценарий, делающий с запросами и ответами все, что только можно вообразить. Упомянем лишь некоторые возможности.

- Переписывание или фильтрация запросов. Например, можно посылать команды самому прокси-серверу, написав сценарий, который будет их распознавать и не передавать серверу, а делать что-то самостоятельно.
- Генерация нового результирующего набора, который выглядит так, будто пришел от сервера MySQL. При этом результаты, сгенерированные реальным сервером MySQL, отбрасываются.
- Динамическая настройка сервера MySQL на основе результатов наблюдений. Например, прокси-сервер может включать или выключать журнал медленных запросов или сохранять статистику запро-

[1] MySQL Proxy развивается настолько быстро, что к моменту выхода книги из печати эта информация вполне может устареть.

сов и выдавать гистограммы времени ответа в ответ на специальный запрос.

- Вставка запросов в момент фиксации каждой транзакции (например, для создания глобального идентификатора транзакции).

Для всего этого уже имеется работающий код, который можно скачать из опубликованных в сети статей или из репозиториев исходного кода. Возможности практически безграничны, и творчески мыслящие пользователи, безусловно, найдут прокси-серверу такие применения, о которых мы даже помыслить не могли. Если вам трудно представить себе открывающиеся перспективы, рекомендуем почитать статьи Джузеппе Максиа (Giuseppe Maxia) или Яна Кнешке (Jan Kneschke).

Dormando's Proxy для MySQL

Еще один проект прокси-сервера, распространяемый по лицензии GPL, начатый примерно в то же время, что MySQL Proxy (и первым предложивший идею использования языка сценариев Lua), называется Dormando's Proxy for MySQL. Отчасти он представляет собой ответ на проект MySQL Proxy, который в то время еще не был выпущен, поэтому его условия лицензирования были неясны. Как и MySQL Proxy, он быстро развивается, поэтому для ознакомления с текущим состоянием рекомендуется скачать последнюю версию. Его веб-сайт находится по адресу *http://www.consoleninja.net/code/dpm/*.

Утилиты, входящие в комплект Maatkit

Мы уже упоминали комплект инструментов Maatkit при обсуждении утилит анализа, но в него входит и много служебных сценариев. Наиболее важны *mk-table-checksum* и *mk-table-sync*, о которых мы писали в разделе «Как узнать, согласованы ли подчиненные серверы с главным» главы 8 (стр. 471). Помимо них, в комплект Maatkit входят следующие инструменты:

mk-archiver

Запускает задания для удаления и архивирования старых данных, освобождая от них таблицы. Инструмент предназначен в первую очередь для того, чтобы перемещать информацию, не создавая помех OLTP-запросам. Его также можно использовать для построения хранилищ и удаления устаревших данных. Он способен записывать значения как в файл, так и в другую таблицу на любом экземпляре сервера MySQL. Имеется механизм подключаемых модулей, упрощающий настройку работы под свои нужды; например, таким образом можно строить сводные таблицы в хранилище данных по ходу вставки значений в таблицу-журнал.

mk-find

Аналог команды *find* в UNIX, но работает для баз данных и таблиц MySQL.

mk-parallel-dump

Создает логические резервные копии в многопоточном режиме, разбивая каждую таблицу на порции желаемого размера. В системе с несколькими процессорами или дисками это позволяет ускорить резервное копирование. На самом деле, эту многопоточную обертку можно применить к любому инструменту, поэтому она равным образом полезна для распараллеливания операций `CHECK TABLE` или `OPTIMIZE TABLE` (к примеру). В системе с несколькими процессорами или дисками выигрыш от распараллеливания можно получить для многих видов задач.

mk-parallel-restore

Дополнение к *mk-parallel-dump*: параллельно загружает файлы в базу MySQL. Этот инструмент может загружать файлы с разделителями непосредственно с помощью команды `LOAD DATA INFILE` или делегировать работу клиентской программе *mysql*. Он представляет собой интеллектуальную обертку для многих операций загрузки, в частности, загрузки сжатых файлов по именованному каналу.

mk-show-grants

Приводит команды `GRANT` к каноническому виду, строит их отрицания, отделяет друг от друга и сортирует. Одно из интересных применений – сохранение привилегий на работу с базой данных в системе управления версиями, не опасаясь случайных изменений.

mk-slave-delay

Заставляет подчиненный сервер отставать от главного, что удобно для восстановления после аварий. Если на главном сервере случайно выполнена какая-нибудь разрушительная SQL-команда, то можно остановить подчиненный до того, как он ее повторит, воспроизвести двоичный журнал до этой команды, а затем преобразовать подчиненный сервер в главный. Обычно это быстрее, чем загружать последнюю резервную копию и накатывать на нее двоичные журналы за день.

mk-slave-prefetch

Реализует технику, описанную в разделе «Инициализируйте кэш для потока репликации» в главе 8 (стр. 498). При определенных характеристиках рабочей нагрузки это позволяет увеличить скорость репликации на подчиненном сервере.

mk-slave-restart

Перезапускает подчиненный сервер после ошибки.

mk-table-checksum

Эффективно вычисляет контрольные суммы содержимого таблиц параллельно на одном или нескольких серверах либо реплицирует запросы для вычисления контрольных сумм с целью проверки целостности подчиненных серверов.

mk-table-sync

Эффективно вычисляет разность между таблицами и генерирует минимальный набор SQL-команд для устранения различий. Может работать также совместно с репликацией.

Бэрон часто добавляет новые инструменты, поэтому этот перечень, возможно, уже устарел. В любой момент вы можете скачать сами сценарии и документацию к ним с сайта *http://maatkit.sourceforge.net*.

Источники дополнительной информации

Если вы ловите себя на том, что раз за разом вручную выполняете с сервером MySQL одни и те же действия, чреватые ошибками, то вполне может статься, что кто-то уже написал инструмент или сценарий, способный облегчить ваше бремя. Вопрос в том, как этот инструмент найти. Мы узнали о многих полюбившихся нам инструментах, регулярно читая агрегатор блогов Planet MySQL (*http://www.planetmysql.org*) и сайт сообщества MySQL Forge (*http://forge.mysql.com*). Это замечательные ресурсы для тех, кто хочет изучить MySQL в общем. Существуют также списки рассылки, IRC-каналы и форумы, где часто можно получить ответы от дружелюбно настроенных знатоков (но сначала поищите в архивах!).

Еще одно место, где мы многое узнали об инструментах и приемах работы с MySQL, – конференции. Даже если у вас не получается посетить конференцию лично, часто можно скачать презентации или посмотреть онлайновую видеозапись.

Информацию о более сложных инструментах типа Nagios можно также найти в посвященных им книгах. Там все описывается гораздо более подробно, чем мы могли позволить себе в этой главе.

A

Передача больших файлов

Копирование, упаковка и распаковка очень больших файлов (часто по сети) – задачи, очень часто встречающиеся при администрировании MySQL, инициализации серверов, клонировании подчиненных серверов, выполнении резервного копирования и восстановления. Не всегда самый очевидный способ выполнения таких операций является наиболее быстрым и лучшим, а разница между хорошим и плохим методом может оказаться весьма значительной. В настоящем приложении мы познакомим вас с некоторыми применениями стандартных утилит UNIX для передачи большого файла, содержащего образ резервной копии, с одного сервера на другой.

Обычно начинают с неупакованного файла, например, табличного пространства InnoDB или файлов журнала. Ну и, разумеется, желательно, чтобы по завершении копирования файл в месте назначения тоже был распакован. Другая распространенная ситуация – начать с упакованного файла, например образа резервной копии, и в результате получить распакованный файл.

Если пропускная способность сети ограничена, то обычно имеет смысл посылать файл по сети в сжатом виде. Кроме того, во избежание компрометации данных может понадобиться защитить файл во время передачи; это требование типично для образов резервных копий.

Копирование файлов

Таким образом, задача заключается в том, чтобы эффективно выполнить следующие операции:

1. Упаковать данные (необязательно).
2. Отправить их на другой компьютер.
3. Распаковать архив в месте назначения.
4. Проверить, что файлы не повреждены в процессе копирования.

Мы протестировали разные методы достижения этих целей. Далее мы опишем их и приведем наши выводы относительно самого быстрого способа.

Для многих обсуждавшихся в этой книге задач, например резервного копирования, следует решить, на какой машине выполнять сжатие. Если пропускная способность сети высока, то можно переносить образы резервных копий без сжатия и сэкономить ресурсы процессора на сервере MySQL для обработки запросов.

Наивный пример

Начнем с наивного примера. Требуется безопасно передать неупакованный файл с одной машины на другую, упаковать его по пути, а затем распаковать в месте назначения. На исходном сервере, назовем его `server1`, выполним следующие команды:

```
server1$ gzip -c /backup/mydb/mytable.MYD > mytable.MYD.gz
server1$ scp mytable.MYD.gz root@server2:/var/lib/myql/mydb/
```

А затем на конечном сервере `server2` дадим следующую команду:

```
server2$ gunzip /var/lib/mysql/mydb/mytable.MYD.gz
```

Наверное, это самый простой подход, но вовсе не самый эффективный, потому что все три шага – упаковка, копирование и распаковка – выполняются последовательно. На каждом шаге необходимо производить медленные операции дискового чтения/записи. Вот что на самом деле происходит при работе каждой из вышеупомянутых команд: *gzip* производит чтение и запись на сервере `server1`, *scp* читает с `server1` и пишет на `server2`, а *gunzip* читает и пишет на `server2`.

Метод с одним шагом

Более эффективно упаковывать и копировать файл на одном конце и распаковывать на другом за один шаг. В этот раз мы воспользуемся безопасным протоколом SSH, на котором основана программа SCP. Вот какую команду мы выполним на сервере `server1`:

```
server1$ gzip -c /backup/mydb/mytable.MYD | ssh root@server2
"gunzip -c - > /var/lib/mysql/mydb/mytable.MYD"
```

Обычно это работает гораздо быстрее, потому что существенно снижается объем дискового ввода/вывода: все сводится к чтению на `server1` записи на `server2`. Поэтому диск может выполнять операции последовательно.

Можно также воспользоваться механизмом сжатия, встроенным в SSH, но мы показали, как упаковывать и распаковывать с помощью конвейера, поскольку это более гибкий метод. Например, если распаковывать файл на другом конце не нужно, то сжатие на уровне SSH не подойдет.

Этот метод можно улучшить, если немного поиграть с флагами. Так, при задании флага `-1` программа *gzip* пакует быстрее. Обычно коэффи-

циент сжатия при этом уменьшается несущественно, зато скорость возрастает очень заметно, а это важно. Можно применять и другие алгоритмы компрессии. Например, если нужна очень высокая степень сжатия и неважно, сколько это займет времени, то вместо *gzip* можно использовать *bzip2*. Если же требуется максимально быстрая компрессия, то лучше выбрать какой-нибудь архиватор на основе алгоритма LZO. Размер сжатого файла может оказаться процентов на 20 больше, зато паковаться он будет примерно в пять раз быстрее.

Устранение накладных расходов на шифрование

SSH – не самый быстрый способ транспортировки данных по сети, поскольку ему присущи накладные расходы на шифрование и дешифрование. Если шифровать данные не требуется, то можно просто побитово скопировать их по сети с помощью программы *netcat*. Для неинтерактивных операций, которые нас и интересуют, она вызывается просто как *nc*.

Рассмотрим пример. Начнем прослушивать порт 12345 (подойдет любой неиспользуемый порт) на сервере `server2` и распаковывать все, что поступает через этот порт в файл с требуемым именем:

```
server2$ nc -l -p 12345 | gunzip -c - > /var/lib/mysql/mydb/mytable.MYD
```

На сервере `server1` запустим еще один экземпляр *netcat*, который будет отправлять данные в порт, прослушиваемый в месте назначения. Флаг *-q* говорит *netcat*, что после обнаружения конца входного файла соединение нужно закрыть. Это, в свою очередь, приведет к тому, что *netcat* на принимающем конце закроет полученный файл и завершится.

```
server1$ gzip -c - /var/lib/mysql/mydb/mytable.MYD | nc -q 1 server2 12345
```

Еще проще воспользоваться программой *tar*, при этом по сети будут передаваться также имена файлов. Это устраняет потенциальный источник ошибок, так как файлы автоматически записываются в нужное место. Опция *z* заставляет *tar* использовать *gzip* для сжатия и распаковки. Вот какую команду следует выполнить на сервере `server2`:

```
server2$ nc -l -p 12345 | tar xvzf -
```

А вот такую – на `server1`:

```
server1$ tar cvzf - /var/lib/mysql/mydb/mytable.MYD | nc -q 1 server2 12345
```

Эти команды можно поместить в один сценарий, который будет эффективно сжимать и копировать файлы по сети, а затем распаковывать их на другом конце.

Другие способы

Еще один вариант – воспользоваться программой *rsync*. Она удобна тем, что позволяет легко поддерживать зеркальное соответствие между ис-

ходным и конечным сервером и возобновлять прерванную передачу. Но если используемый в ней алгоритм вычисления разности между двоичными файлами невозможно эффективно применить, то она работает не очень хорошо. Пожалуй, имеет смысл применять ее в тех случаях, когда заведомо известно, что большую часть файла посылать не потребуется, например для завершения прерванной операции копирования с помощью *nc*.

Экспериментировать с передачей файлов следует в спокойных условиях, когда нет никакой спешки, так как для отыскания самого быстрого варианта придется воспользоваться методом проб и ошибок. Какой способ будет работать быстрее всего, зависит от конкретной системы. Основными факторами тут является количество дисков, сетевых карт и процессоров, а также их относительное быстродействие. Мы рекомендуем для начала выполнить команду `vmstat -n 5`, чтобы узнать, что является узким местом: диски или процессор.

Если часть процессоров простаивает, то, возможно, операцию удастся ускорить, запустив несколько процедур копирования параллельно. И наоборот, если процессор является узким местом, зато у дисков и сети имеется свободная пропускная способность, то можно исключить сжатие. И не забывайте вести мониторинг производительности серверов, чтобы понять, располагают ли они резервной пропускной способностью. Попытка запустить слишком много параллельных операций может даже привести к замедлению.

Эталонные тесты копирования файлов

Для сравнения в табл. А.1 показано, за какое время нам удалось скопировать тестовый файл по стандартной локальной сети Ethernet со скоростью 100 Мбит/с. Размер исходного файла составлял 738 Мбайт, а упакованного программой *gzip* с флагами по умолчанию – 100 Мбайт. На исходной и конечной машинах было достаточно памяти, ресурсов процессора и свободного места на диске; узким местом была сеть.

Таблица А.1. Эталонные тесты копирования файлов по сети

Метод	Время (сек.)
rsync без сжатия	71
scp без сжатия	68
nc без сжатия	67
rsync со сжатием (*-z*)	63
gzip, *scp* и *gunzip*	60 (44 + 10 + 6)
ssh со сжатием	44
nc со сжатием	42

Обратите внимание на то, как сжатие ускоряет передачу файла по сети, – три метода, в которых сжатие не применялось, оказались самыми медленными. Но, конечно, ваши результаты могут отличаться. Если процессоры и диски не отличаются быстродействием, зато имеется гигабитная сеть Ethernet, то узким местом будут чтение и компрессия данных, поэтому, может быть, лучше обойтись без сжатия.

Кстати, часто бывает гораздо быстрее использовать быструю архивацию, например *gzip --fast*, чем стандартные уровни компрессии, которые потребляют много процессорного времени, давая лишь небольшой выигрыш в коэффициенте сжатия. Мы в своем тесте довольствовались уровнем сжатия по умолчанию.

Последний шаг – проверка того, что данные не были искажены в процессе передачи данных. Тут в вашем распоряжении много разнообразных методов, например программа *md5sum*, которая, правда, обходится довольно дорого, так как должна еще раз прочитать весь файл. Это еще одна причина, по которой полезно сжатие: алгоритм компрессии, как правило, уже включает по меньшей мере циклический избыточный код (CRC), который «отлавливает» все ошибки, так что контроль отсутствия искажений вы получаете бесплатно.

В

Команда EXPLAIN

В этом приложении описывается, как с помощью команды EXPLAIN получать сведения о плане выполнения запроса и как интерпретировать ее результаты. Команда EXPLAIN — основной способ узнать, какие решения принимает оптимизатор запросов. У нее есть много ограничений и не всегда она говорит правду, но поскольку ничего лучшего все равно не существует, то имеет смысл научиться ею пользоваться, тогда вы сможете делать обоснованные предположения о том, как выполняются запросы.

Вызов команды EXPLAIN

Чтобы воспользоваться командой EXPLAIN, достаточно добавить слово EXPLAIN перед словом SELECT в запросе. MySQL пометит этот запрос специальным флагом. Во время его обработки этот флаг заставит сервер сообщать информацию о каждом шаге плана выполнения, а не исполнять его. При этом возвращается одна или несколько строк, описывающих этапы плана выполнения и порядок их запуска.

Ниже показан простейший из возможных результатов EXPLAIN:

```
mysql> EXPLAIN SELECT 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: NULL
         type: NULL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
        Extra: No tables used
```

Для каждой встречающейся в запросе таблицы выводится одна строка. Если соединяются две таблицы, то будет выведено две строки. Если одна и та же таблица попадается дважды, например при соединении таблицы с собой же, тоже будет выведено две строки. В этом контексте семантика слова «таблица» довольно широка: оно может обозначать подзапрос, результат объединения (UNION) и т. д. Позже мы увидим, почему это так.

Существует две важные разновидности команды EXPLAIN:

- Команда EXPLAIN EXTENDED, по видимости, ведет себя так же, как обычная EXPLAIN, но говорит серверу, что требуется произвести «обратную компиляцию» плана выполнения в команду SELECT. Чтобы просмотреть сгенерированный запрос, следует сразу после завершения EXPLAIN EXTENDED выполнить команду SHOW WARNINGS. Отображаемая при этом директива получена непосредственно из плана выполнения, а не из исходной SQL-команды, которая к этому моменту уже преобразована в некую структуру данных. В большинстве случаев исходная и восстановленная команды будут различаться. Сравнив их, вы поймете, как оптимизатор трансформировал ваш запрос. Команда EXPLAIN EXTENDED появилась в версии MySQL 5.0, а в версии MySQL 5.1 был добавлен дополнительный столбец filtered (подробнее об этом ниже).
- Команда EXPLAIN PARTITIONS показывает, к каким секциям обращается запрос (если это имеет смысл). Она доступна начиная с версии MySQL 5.1. Подробнее см. в разделе «Секционированные таблицы» главы 5 на стр. 323.

Часто считают, что при наличии слова EXPLAIN MySQL не выполняет запрос. На самом деле, если запрос содержит подзапрос во фразе FROM, то MySQL обрабатывает этот подзапрос, помещает результат во временную таблицу, после чего продолжает оптимизацию внешнего запроса. Сервер обязан выполнить все подзапросы перед тем, как сможет завершить оптимизацию внешнего запроса, а в этом и состоит суть EXPLAIN. Таким образом, если запрос содержит сложные подзапросы или представления, в которых применяется алгоритм TEMPTABLE, то выполнение EXPLAIN может в действительности означать большой объем работы для сервера.

Имейте в виду, что команда EXPLAIN – не более чем аппроксимация. Иногда хорошая, а иногда очень далекая от истины. Приведем перечень ее ограничений.

- EXPLAIN ничего не говорит о том, как влияют на запрос триггеры, хранимые и пользовательские (UDF) функции.
- Она не работает с хранимыми процедурами, хотя можно разложить процедуру на отдельные запросы и вызвать EXPLAIN для каждого из них.
- Она ничего не говорит об оптимизациях, которые MySQL производит уже на этапе выполнения запроса.

- Часть отображаемой статистической информации – всего лишь оценка, иногда очень неточная.
- Она не показывает все, что можно было бы сообщить о плане выполнения запроса (когда представляется возможность, разработчики MySQL включают дополнительную информацию).
- Она не делает различий между некоторыми операциями, называя их одинаково. Например, словом «filesort» обозначается и сортировка в памяти, и сортировка с помощью временных файлов, а при использовании временных таблиц – все равно, на диске или в памяти, – она сообщает «Using temporary».
- Результат может сбить с толку. Например, она может сообщить о полном сканировании индекса для запроса, который отбирает небольшое число строк благодаря наличию фразы `LIMIT`. (В версии MySQL 5.1 команда `EXPLAIN` выдает более точную информацию о количестве подлежащих просмотру строк, но в предыдущих версиях фраза `LIMIT` не принималась во внимание.)

Переписывание запросов, отличных от SELECT

Команда `EXPLAIN` работает только для запросов типа `SELECT`, но не для хранимых процедур, команд `INSERT`, `UPDATE`, `DELETE` и прочих. Однако некоторые отличные от `SELECT` запросы можно переписать в виде, пригодном для «объяснения». Для этого следует преобразовать команду в эквивалентный запрос `SELECT`, который выбирает те же самые столбцы. Любой упоминаемый в исходной директиве столбец должен встречаться либо в списке `SELECT`, либо в условии соединения, либо во фразе `WHERE`.

Предположим, например, что нужно переписать следующую команду `UPDATE`, сделав ее «объясняемой»:

```
UPDATE sakila.actor
   INNER JOIN sakila.film_actor USING (actor_id)
SET actor.last_update=film_actor.last_update;
```

Следующая директива `EXPLAIN` *не* эквивалентна этой команде `UPDATE`, так как от сервера не требуется выбирать столбец `last_update` ни из одной таблицы:

```
mysql> EXPLAIN SELECT film_actor.actor_id
    -> FROM sakila.actor
    ->    INNER JOIN sakila.film_actor USING (actor_id)\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: index
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
```

```
           ref: NULL
          rows: 200
         Extra: Using index
*************************** 2. row ***************************
            id: 1
  select_type: SIMPLE
         table: film_actor
          type: ref
possible_keys: PRIMARY
           key: PRIMARY
       key_len: 2
           ref: sakila.actor.actor_id
          rows: 13
         Extra: Using index
```

Это различие очень важно. Результат приведенной выше команды EXPLAIN показывает, что MySQL будет использовать покрывающие индексы, а это было бы невозможно в случае выборки и обновления столбца `last_update`. Следующий вариант гораздо ближе к оригиналу:

```
mysql> EXPLAIN SELECT film_actor.last_update, actor.last_update
    -> FROM sakila.actor
    ->    INNER JOIN sakila.film_actor USING (actor_id)\G
*************************** 1. row ***************************
            id: 1
  select_type: SIMPLE
         table: actor
          type: ALL
possible_keys: PRIMARY
           key: NULL
       key_len: NULL
           ref: NULL
          rows: 200
         Extra:
*************************** 2. row ***************************
            id: 1
  select_type: SIMPLE
         table: film_actor
          type: ref
possible_keys: PRIMARY
           key: PRIMARY
       key_len: 2
           ref: sakila.actor.actor_id
          rows: 13
         Extra:
```

Переписывание запросов – не точная наука, но часто этого бывает достаточно, чтобы узнать, как будет выполняться запрос.

Важно понимать, что не существует такого понятия, как «эквивалентный» запрос на чтение, который показал бы план выполнения запроса на запись. От запроса типа `SELECT` требуется найти и вернуть всего одну

копию данных. С другой стороны, запрос, модифицирующий данные, должен найти и изменить все копии данных, во всех индексах. Часто это оказывается гораздо дороже, чем эквивалентный запрос SELECT.

Столбцы результата команды EXPLAIN

Результат команды EXPLAIN всегда состоит из одних и тех же столбцов (за исключением команды EXPLAIN EXTENDED, которая в версии MySQL 5.1 добавляет столбец filtered, и команды EXPLAIN PARTITIONS, которая добавляет столбец partitions). Изменяется лишь количество и содержимое строк. Однако чтобы не усложнять примеры, мы не всегда будем показывать все столбцы.

В следующих разделах мы объясним назначение столбцов в результате, выдаваемом командой EXPLAIN. Имейте в виду, что строки следуют в том порядке, в котором MySQL фактически выполняет части запроса, а он не всегда совпадает с порядком упоминания в исходной SQL-команде.

Столбец id

В этом столбце всегда находится число, идентифицирующее запрос SELECT, которому принадлежит строка. Если в исходной команде нет ни подзапросов, ни фразы UNION, то существует лишь один запрос SELECT, так что во всех строках этот столбец будет содержать 1. В противном случае, внутренние запросы SELECT обычно нумеруются последовательно в том порядке, в каком встречаются в исходной команде.

MySQL разбивает все запросы SELECT на простые и составные, причем составные запросы можно отнести к одной из трех категорий: простые подзапросы, так называемые производные таблицы (подзапросы во фразе FROM) и объединения UNION[1]. Вот пример простого подзапроса:

```
mysql> EXPLAIN SELECT (SELECT 1 FROM sakila.actor LIMIT 1) FROM sakila.film;
+----+-------------+-------+...
| id | select_type | table |...
+----+-------------+-------+...
|  1 | PRIMARY     | film  |...
|  2 | SUBQUERY    | actor |...
+----+-------------+-------+...
```

Подзапросы во фразе FROM и UNION'ы усложняют содержимое столбца id. Вот пример несложного подзапроса во фразе FROM:

```
mysql> EXPLAIN SELECT film_id FROM (SELECT film_id FROM sakila.film) AS der;
+----+-------------+------------+...
| id | select_type | table      |...
```

[1] Утверждение «всякий подзапрос во фразе FROM является производной таблицей» истинно, однако утверждение «всякая производная таблица является подзапросом во фразе FROM» ложно. В языке SQL понятие «производная таблица» употребляется более широко.

```
+----+-------------+-----------+...
|  1 | PRIMARY     | <derived2> |...
|  2 | DERIVED     | film       |...
+----+-------------+-----------+...
```

Как вы знаете, этот запрос выполняется при помощи временной таблицы. MySQL ссылается на данную временную таблицу из внешнего запроса по псевдониму `der`, который в более сложных запросах указывается в столбце `ref`.

Наконец, приведем запрос с `UNION`:

```
mysql> EXPLAIN SELECT 1 UNION ALL SELECT 1;
+------+--------------+------------+...
| id   | select_type  | table      |...
+------+--------------+------------+...
|  1   | PRIMARY      | NULL       |...
|  2   | UNION        | NULL       |...
| NULL | UNION RESULT | <union1,2> |...
+------+--------------+------------+...
```

Обратите внимание на дополнительную строку, соответствующую результату выполнения `UNION`. Итоги обработки `UNION` всегда помещаются во временную таблицу, из которой MySQL затем считывает их обратно. Эта временная таблица отсутствует в исходной SQL-команде, поэтому в столбце `id` для нее стоит `NULL`. В отличие от предыдущего примера (иллюстрирующего подзапрос во фразе `FROM`), временная таблица, порождаемая в ходе выполнения запроса, показана в последней, а не в первой строке.

До сих пор все было достаточно просто, но комбинация всех трех категорий команд может существенно усложнить результат, в чем мы скоро убедимся.

Столбец select_type

Этот столбец показывает, соответствует ли строка простому или составному запросу `SELECT` (и если составному, то к какой из трех категорий он относится). Атрибут `SIMPLE` означает, что запрос не содержит ни подзапросов, ни `UNION`. Если же в запросе имеются такие компоненты, то самый внешний запрос помечается признаком `PRIMARY`, а остальные – следующим образом:

`SUBQUERY`

Запрос `SELECT`, который содержится в подзапросе, находящемся во фразе `SELECT` (иными словами, не во фразе `FROM`), помечается признаком `SUBQUERY`.

`DERIVED`

Значение `DERIVED` означает, что данный запрос `SELECT` является подзапросом во фразе `FROM`. MySQL выполняет его рекурсивно и помещает

результат во временную таблицу. Внутри сервер ссылается на нее по имени «derived table», так как она «произведена» из подзапроса.

`UNION`

Второй и последующий запросы `SELECT`, входящие в объединение `UNION`, помечаются признаком `UNION`. При этом первый `SELECT` помечается так, будто он является частью внешнего запроса. Именно поэтому первый `SELECT` в `UNION` из предыдущего примера помечен как `PRIMARY`. Если бы это объединение `UNION` было частью подзапроса во фразе `FROM`, то его первый `SELECT` был бы помечен как `DERIVED`.

`UNION RESULT`

Запрос `SELECT`, применяемый для выборки результатов из временной таблицы, созданной в ходе выполнения `UNION`, помечается признаком `UNION RESULT`.

Помимо этих значений, признаки `SUBQUERY` и `UNION` могут быть дополнительно квалифицированы как `DEPENDENT` и `UNCACHEABLE`. `DEPENDENT` означает, что результат `SELECT` зависит от данных, встречающихся во внешнем запросе; `UNCACHEABLE` означает, что нечто в запросе `SELECT` мешает поместить результаты в кэш `Item_cache`. Кэш `Item_cache` не документирован; это не то же самое, что кэш запросов, хотя помещению в тот и другой могут воспрепятствовать одни и те же конструкции, например функция `RAND()`.

Столбец table

Этот столбец показывает, к какой таблице относится данная строка. В большинстве случаев все просто: это таблица или ее псевдоним, встречающиеся в SQL-команде.

Читать этот столбец следует сверху вниз, чтобы видеть порядок соединения таблиц, выбранный оптимизатором. Например, в следующем примере MySQL выбрала порядок соединения, отличный от указанного в самом запросе:

```
mysql> EXPLAIN SELECT film.film_id
    -> FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id)
    ->    INNER JOIN sakila.actor USING(actor_id);
+----+-------------+------------+...
| id | select_type | table      |...
+----+-------------+------------+...
|  1 | SIMPLE      | actor      |...
|  1 | SIMPLE      | film_actor |...
|  1 | SIMPLE      | film       |...
+----+-------------+------------+...
```

Вспомните диаграммы в виде «левоглубоких» деревьев из раздела «План выполнения» главы 4 (стр. 225). Планы выполнения в MySQL всегда представляются «левоглубокими» деревьями. Если положить такой

план набок, то листовые узлы покажут порядок выполнения и будут точно соответствовать строкам результата команды EXPLAIN. План выполнения предыдущего запроса показан на рис. B.1.

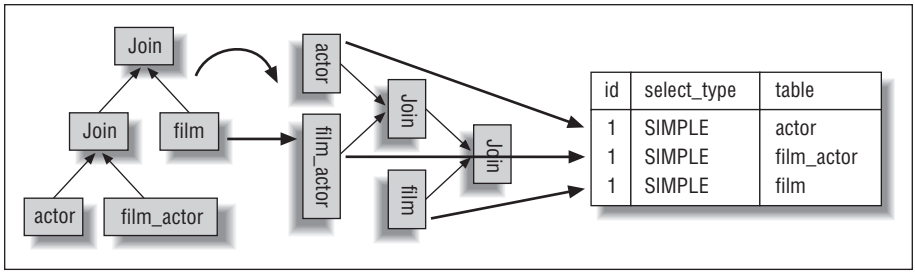


Рис. B.1. *Как план выполнения запроса соответствует строкам результата команды EXPLAIN*

Производные таблицы и объединения

Столбец table становится существенно сложнее, если запрос содержит подзапрос во фразе FROM или объединение UNION. В этих случаях не существует «таблицы», на которую можно было бы сослаться, так как временная таблица существует лишь до тех пор, пока запрос выполняется.

Для подзапроса во фразе FROM столбец table принимает вид <derived*N*>, где *N* – идентификатор подзапроса. Это всегда «опережающая ссылка», то есть *N* относится к строке, которая встречается в выданном EXPLAIN результате ниже.

Для запросов, содержащих UNION, строка UNION RESULT в столбце table содержит список идентификаторов запросов, для которых производится объединение. Это всегда «обратная ссылка», поскольку строка с признаком UNION RESULT встречается после всех строк, относящихся к частям UNION. Если в списке более 20 идентификаторов, то столбец table может быть обрезан, и всех значений вы не увидите. К счастью, нетрудно догадаться, какие строки были включены, поскольку идентификатор первой из них всегда виден, а все, что находится между ней и строкой UNION RESULT, так или иначе включается.

Пример с различными типами составных запросов SELECT

Ниже в качестве примера приведен бессмысленный запрос, который тем не менее в компактной форме демонстрирует некоторые типы составных запросов SELECT.

```
1 EXPLAIN
2 SELECT actor_id,
3    (SELECT 1 FROM sakila.film_actor WHERE film_actor.actor_id =
4       der_1.actor_id LIMIT 1)
5 FROM (
6    SELECT actor_id
```

```
 7    FROM sakila.actor LIMIT 5
 8 ) AS der_1
 9 UNION ALL
10 SELECT film_id,
11    (SELECT @var1 FROM sakila.rental LIMIT 1)
12 FROM (
13    SELECT film_id,
14       (SELECT 1 FROM sakila.store LIMIT 1)
15    FROM sakila.film LIMIT 5
16 ) AS der_2;
```

Фразы LIMIT включены только для удобства, на случай, если вы захотите выполнить этот запрос без EXPLAIN и посмотреть результаты его работы. Вот как выглядит результат EXPLAIN:

```
+------+---------------------+------------+...
| id   | select_type         | table      |...
+------+---------------------+------------+...
|  1   | PRIMARY             | <derived3> |...
|  3   | DERIVED             | actor      |...
|  2   | DEPENDENT SUBQUERY  | film_actor |...
|  4   | UNION               | <derived6> |...
|  6   | DERIVED             | film       |...
|  7   | SUBQUERY            | store      |...
|  5   | UNCACHEABLE SUBQUERY| rental     |...
| NULL | UNION RESULT        | <union1,4> |...
+------+---------------------+------------+...
```

Мы специально построили запрос так, чтобы каждая его часть обращалась к разным таблицам, и вам было бы понятно, что куда попадает. И тем не менее, разобраться трудно! Начнем сверху.

- Первая строка является опережающей ссылкой на производную таблицу der_1, которая в запросе помечена как <derived3>. Она соответствует строке 2 исходной SQL-команды. Чтобы понять, какие строки результата относятся к командам SELECT, являющимся частью <derived3>, смотрим вперед...
- ...на вторую строку с идентификатором 3. Значение 3 объясняется тем, что строка относится к третьему по порядку запросу SELECT и помечена признаком DERIVED, поскольку соответствующий подзапрос встречается во фразе FROM (строки 6 и 7 исходной SQL-команды).
- Третья строка имеет идентификатор 2. Она соответствует строке 3 исходной SQL-команды. Отметим, что она идет после строки с большим идентификатором, то есть соответствующий ей подзапрос выполняется позже, как и должно быть. Она помечена признаком DEPENDENT SUBQUERY, иными словами, результат зависит от итогов обработки внешнего запроса (такие подзапросы часто называют коррелированными). В данном случае внешний запрос начинается в строке 2 и выбирает данные из «таблицы» der_1.

- Четвертая строка помечена признаком UNION, следовательно, соответствует второму или последующему подзапросу SELECT в UNION. В столбце table находится значение <derived6>, это означает, что данные выбираются из результатов подзапроса во фразе FROM и добавляются во временную таблицу для результатов UNION. Как и раньше, чтобы найти в результате EXPLAIN строки, в которых показан план выполнения этого подзапроса, нужно заглянуть вперед.
- Пятая строка соответствует подзапросу der_2, определенному в строках 13, 14 и 15. EXPLAIN ссылается на него по имени <derived6>.
- Шестая строка – это обычный подзапрос в списке SELECT «таблицы» <derived6>. Его идентификатор равен 7, что существенно...
- ...так как он больше 5 – идентификатора седьмой строки. Почему это так важно? Потому, что очерчивает границы подзапроса <derived6>. Каждая выведенная EXPLAIN строка с признаком DERIVED является началом «вложенной области видимости». Как только появляется строка с меньшим идентификатором (в данном случае 5 меньше 6), вложенная область видимости закрывается. Поэтому мы знаем, что седьмая строка – часть списка SELECT, в котором выбираются данные из <derived6>, то есть часть списка SELECT из четвертой строки (строка 11 в исходной SQL-команде). Этот пример довольно легко понять, не зная о правилах вложенных областей видимости, но так бывает не всегда. Стоит также отметить, что эта строка помечена признаком UNCACHEABLE SUBQUERY из-за пользовательской переменной.
- Наконец, последняя строка помечена признаком UNION RESULT. Она представляет этап считывания строк из временной таблицы, соответствующей UNION. Можете, если хотите, начать с этой строки и двигаться в обратном направлении; она возвращает объединение результатов, полученных в строках с идентификаторами 1 и 4, которые, в свою очередь, ссылаются на <derived3> и <derived6>.

Как видите, комбинация сложных подзапросов SELECT может порождать трудные для восприятия результаты EXPLAIN. Знание правил облегчает их чтение, но нет ничего лучше практики.

При изучении результатов EXPLAIN часто приходится заглядывать вперед и возвращаться назад. Возьмем, к примеру, первую строку результата. Глядя на нее, невозможно сказать, что она является частью UNION. Это становится ясно только после ознакомления с последней строкой.

Столбец type

В руководстве по MySQL сказано, что в этом столбце отражается «тип соединения», но нам кажется, что следует говорить скорее о *методе доступа*, иными словами, о том, как MySQL решила искать строки в таблице. Перечислим наиболее важные методы доступа в порядке от наихудшего к наилучшему.

`ALL`

Этот подход обычно называют сканированием таблицы. В общем случае речь идет о том, что MySQL должна просмотреть таблицу от начала до конца, чтобы найти нужную строку. Существуют исключения, например запросы с фразой `LIMIT` или запросы, для которых в столбце `Extra` отображается значение «Using distinct/not exists».

`index`

То же, что сканирование таблицы, только MySQL просматривает ее в порядке, задаваемом индексом, а не в порядке следования строк. Основное преимущество заключается в том, что не требуется сортировка; недостаток же – высокая стоимость чтения всей таблицы в порядке, задаваемом индексом. Обычно это означает, что строки выбираются произвольным образом, что крайне накладно.

Если в столбце `Extra` при этом находится значение «Using index», значит, MySQL использует покрывающий индекс (см. главу 3) и просматривает только данные в индексе, не обращаясь к строкам. Это гораздо дешевле, чем сканирование таблицы в задаваемом индексе.

`range`

Просмотр диапазона – это ограниченная форма сканирования индекса. Просмотр начинается в определенной точке индекса и возвращает строки в некотором диапазоне значений. Это лучше, чем полное сканирование, так как не приходится перебирать индекс целиком. Очевидные примеры просмотра диапазона – запросы с условиями `BETWEEN` или `>` во фразе `WHERE`.

Когда MySQL использует индекс для поиска в списке значений, например, при вычислении предиката `IN()` или условий, объединенных связкой `OR`, также применяется просмотр диапазона. Однако это совершенно разные методы доступа с иными характеристиками производительности. Дополнительную информацию см. во врезке «Что такое условие поиска по диапазону?» в главе 3 (стр. 180).

К этому методу применимы те же самые соображения о стоимости, что и к методу `index`.

`ref`

Это доступ по индексу (иногда он называется поиском по индексу (index lookup)), в результате которого возвращаются строки, соответствующие единственному заданному значению. Но таких строк может быть несколько, поэтому поиск сочетается с просмотром. Данный тип доступа возможен лишь в случае неуникального индекса или неуникального префикса ключа в уникальном индексе. Он называется `ref`, потому что значения ключей в индексе сравниваются с некоторой справочной (reference) величиной. Справочная величина может быть как константой, так и значением из предыдущей таблицы, если в запросе участвует несколько таблиц.

Одним из вариантов ref является тип доступа ref_or_null. В этом случае MySQL должна выполнить второй просмотр для поиска записей с ключом NULL.

eq_ref

Это поиск по индексу в случае, когда MySQL точно знает, что будет возвращено не более одного значения. Такой метод доступа применяется, когда MySQL решает использовать первичный ключ или уникальный индекс для сравнения на равенство с некоторой справочной величиной. Этот тип доступа MySQL умеет отлично оптимизировать, так как заранее известно, что не придется просматривать диапазоны отвечающих условию строк или искать дополнительные строки после того, как одна уже найдена.

const, system

Эти типы доступа MySQL применяет, когда в процессе оптимизации какую-то часть запроса можно преобразовать в константу. Например, если вы выбираете первичный ключ строки и помещаете его в условие WHERE, то MySQL может преобразовать этот запрос в константу. Затем соответствующая таблица по существу удаляется из операции соединения.

NULL

Этот метод означает, что MySQL сумела разрешить запрос на фазе оптимизации, так что в ходе выполнения вообще не потребуется обращаться к таблице или индексу. Например, для выборки минимального значения проиндексированного столбца достаточно просмотреть только индекс, не обращаясь к таблице.

Столбец possible_keys

Этот столбец показывает, какие индексы можно было бы задействовать для выполнения запроса, исходя из того, к каким столбцам производится обращение и какие операторы сравнения используются. Список создается на ранних этапах фазы оптимизации, поэтому на последующих стадиях может выясниться, что некоторые индексы бесполезны.

Столбец key

Этот столбец показывает, какой индекс MySQL решила использовать для оптимизации доступа к таблице. Если этот индекс отсутствует в столбце possible_keys, значит, MySQL выбрала его по какой-то другой причине – например, покрывающий индекс может быть избран даже в том случае, если фразы WHERE нет вообще.

Иными словами, столбец possible_keys показывает, какие индексы могли бы способствовать эффективному поиску строк, тогда как столбец key говорит о том, на каком индексе оптимизатор остановился, чтобы

минимизировать стоимость запроса (о метриках стоимости запроса см. раздел «Процесс оптимизации запроса» главы 4 на стр. 214). Приведем пример:

```
mysql> EXPLAIN SELECT actor_id, film_id FROM sakila.film_actor\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: index
possible_keys: NULL
          key: idx_fk_film_id
      key_len: 2
          ref: NULL
         rows: 5143
        Extra: Using index
```

Столбец key_len

Этот столбец показывает, сколько байт индекса использует MySQL. Если задействованы лишь некоторые индексированные столбцы, то, зная это значение, можно определить, какие именно. Напомним, что MySQL может использовать левый префикс индекса. Например, первичный ключ таблицы `sakila.film_actors` состоит из двух столбцов типа `SMALLINT`, а длина типа `SMALLINT` – два байта, поэтому каждая запись в индексе содержит четыре байта. Рассмотрим такой запрос:

```
mysql> EXPLAIN SELECT actor_id, film_id FROM sakila.film_actor WHERE actor_id=4;
...+------+---------------+---------+---------+...
...| type | possible_keys | key     | key_len |...
...+------+---------------+---------+---------+...
...| ref  | PRIMARY       | PRIMARY | 2       |...
...+------+---------------+---------+---------+...
```

Исходя из значения столбца `key_len`, можно сделать вывод, что для поиска по индексу задействован только первый столбец, `actor_id`. Определяя, какие столбцы используются, не забывайте о кодировке символьных столбцов:

```
mysql> CREATE TABLE t (
    ->    a char(3) NOT NULL,
    ->    b int(11) NOT NULL,
    ->    c char(1) NOT NULL,
    ->    PRIMARY KEY  (a,b,c)
    -> ) ENGINE=MyISAM DEFAULT CHARSET=utf8 ;
mysql> INSERT INTO t(a, b, c)
    ->    SELECT DISTINCT LEFT(TABLE_SCHEMA, 3), ORD(TABLE_NAME),
    ->       LEFT(COLUMN_NAME, 1)
    ->    FROM INFORMATION_SCHEMA.COLUMNS:
mysql> EXPLAIN SELECT a FROM t WHERE a='sak' AND b = 112;
```

```
...+------+---------------+---------+---------+...
...| type | possible_keys | key     | key_len |...
...+------+---------------+---------+---------+...
...| ref  | PRIMARY       | PRIMARY | 13      |...
...+------+---------------+---------+---------+...
```

Длина ключа 13 в этом запросе равна сумме длин столбцов `a` и `b`. Длина столбца `a` составляет 3 символа, для каждого из которых в кодировке `utf8` может понадобиться до трех байтов, а столбец `b` представляет собой четырехбайтовое целое число.

MySQL не всегда показывает, какая часть индекса реально используется. Например, для запроса с предикатом `LIKE`, в котором производится сопоставление с шаблоном по совпадению префикса, MySQL сообщит, что задействована вся ширина столбца.

В столбце `key_len` отражается максимально возможная длина индексированных полей, а не фактическая длина данных, хранимых в таблице. Так, в предыдущем примере MySQL всегда показывает 13 байтов, даже если столбец `a` не содержит более одного символа. Иными словами, значение `key_len` вычисляется, исходя из определения структуры таблицы, а не значений в ней.

Столбец ref

Этот столбец показывает, какие столбцы и константы из предыдущих таблиц используются для поиска в индексе, имя которого указано в столбце `key`. В следующем примере демонстрируется сочетание условий соединения и псевдонимов. Обратите внимание на столбец `ref`, в котором показано, что таблица `film` фигурирует в тексте запроса под псевдонимом `f`:

```
mysql> EXPLAIN
    -> SELECT STRAIGHT_JOIN f.film_id
    -> FROM sakila.film AS f
    ->    INNER JOIN sakila.film_actor AS fa
    ->       ON f.film_id=fa.film_id AND fa.actor_id = 1
    ->    INNER JOIN sakila.actor AS a USING(actor_id);
...+-------+...+--------------------+---------+-----------------------+...
...| table |...| key                | key_len | ref                   |...
...+-------+...+--------------------+---------+-----------------------+...
...| a     |...| PRIMARY            | 2       | const                 |...
...| f     |...| idx_fk_language_id | 1       | NULL                  |...
...| fa    |...| PRIMARY            | 4       | const,sakila.f.film_id |...
...+-------+...+--------------------+---------+-----------------------+...
```

Столбец rows

В этом столбце демонстрируется, сколько строк по оценке MySQL придется прочитать, чтобы найти запрошенные. Это значение вычисляется в расчете на каждую итерацию плана выполнения с вложенными

циклами. Иными словами, это не просто количество строк, которые, по мнению MySQL, предстоит прочитать, а среднее количество строк, которые нужно будет прочитать, чтобы удовлетворить критерию, действующему в данной точке выполнения запроса. Критерием может быть как константа, заданная в SQL-команде, так и текущее значение столбца из предыдущей таблицы в порядке соединения.

В зависимости от имеющейся статистики таблицы и избирательности индекса оценка может оказаться очень неточной. Кроме того, в версии MySQL 5.0 и более ранних не учитывается фраза `LIMIT`. Например, при обработке следующего запроса не будут просматриваться 1022 строки:

```
mysql> EXPLAIN SELECT * FROM sakila.film LIMIT 1\G
...
          rows: 1022
```

Грубо оценить количество просматриваемых строк для всего запроса можно, перемножив все значения в столбце `rows`. Например, при выполнении следующего запроса, возможно, понадобится просмотреть примерно 2600 строк:

```
mysql> EXPLAIN
    -> SELECT f.film_id
    -> FROM sakila.film AS f
    ->    INNER JOIN sakila.film_actor AS fa USING(film_id)
    ->    INNER JOIN sakila.actor AS a USING(actor_id);
...+------+...
...| rows |...
...+------+...
...|  200 |...
...|   13 |...
...|    1 |...
...+------+...
```

Напомним, что речь идет не о числе строк в результирующем наборе, а о количестве строк, которое MySQL должна будет просмотреть. Кроме того, имейте в виду, что существует множество оптимизаций, например, буферы соединения и кэши, которые не учитываются при оценивании количества строк. Очень может быть, что число реально читаемых сервером строк окажется меньше предсказанного. К тому же MySQL ничего не знает о кэшах, реализованных на уровне операционной системы и оборудования.

Столбец filtered

Этот столбец появился в версии MySQL 5.1 и отображается при выполнении команды `EXPLAIN EXTENDED`. Он показывает пессимистическую оценку процентной доли строк, удовлетворяющих некоторому критерию, заданному, например, во фразе `WHERE` или в условии соединения. Если умножить значение в столбце `rows` на этот процент, то получится оценка числа строк, которые MySQL должна будет соединить с преды-

дущими таблицами. В настоящее время оптимизатор использует эту оценку только для методов доступа ALL, index, range и index_merge.

Для иллюстрации мы создали следующую таблицу:

```
CREATE TABLE t1 (
   id INT NOT NULL AUTO_INCREMENT,
   filler char(200),
   PRIMARY KEY(id)
);
```

Затем мы вставили в нее 1000 строк со случайным текстом в столбце filler. Его назначение состоит в том, чтобы MySQL не использовала покрывающий индекс при выполнении следующего запроса:

```
mysql> EXPLAIN EXTENDED SELECT * FROM t1 WHERE id < 500\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1000
     filtered: 49.40
        Extra: Using where
```

MySQL могла бы воспользоваться методом доступа range для поиска всех строк, в которых поле id меньше 500, но не стала так поступать, потому что при этом удалось бы исключить только половину строк. Сервер полагает, что полное сканирование таблицы обойдется дешевле. При этом лишние строки отфильтровываются по условию во фразе WHERE. Имея оценку стоимости, вычисленную для просмотра диапазона, сервер знает, сколько строк это условие отсеет. Поэтому-то в столбце filtered и отображается значение 49,40%.

Столбец Extra

Этот столбец содержит дополнительную информацию, для которой не нашлось места в других столбцах. В руководстве по MySQL документированы многие значения, отображаемые в указанном столбце; часть из них упоминалась на страницах этой книги.

Наиболее важны следующие выражения, с которыми вам придется сталкиваться чаще всего.

"Using index"

Означает, что MySQL воспользуется покрывающим индексом, чтобы избежать доступа к самой таблице (см. раздел «Покрывающие индексы» главы 3 на стр. 163). Не путайте покрывающие индексы с методом доступа по индексу.

"Using where"

Означает, что сервер произведет дополнительную фильтрацию строк, отобранных подсистемой хранения. Многие условия `WHERE`, в которых участвуют индексированные столбцы, могут быть проверены подсистемой хранения, когда (и если) она читает индекс, поэтому не для всех запросов с фразой `WHERE` признак «Using where» присутствует. Иногда его наличие означает лишь, что запрос можно было бы выполнить более эффективно при другом индексировании.

"Using temporary"

Означает, что MySQL будет применять временную таблицу для сортировки результатов запроса.

"Using filesort"

Означает, что MySQL прибегнет к обычной сортировке для упорядочения результатов, а не станет читать строки из таблицы в порядке, задаваемом индексом. В MySQL реализовано два алгоритма файловой сортировки, о которых можно прочитать в разделе «Оптимизация файловой сортировки» главы 6 (стр. 375). В обоих случаях сортировка может быть произведена в памяти или на диске. `EXPLAIN` ничего не говорит о том, какой тип файловой сортировки будет использован и где именно это будет происходить.

"range checked for each record (index map: `N`)"

Означает, что подходящего индекса не нашлось, поэтому сервер будет заново искать индекс при обработке каждой строки в операции соединения. *N* представляет собой битовую карту индексов, показанных в столбце `possible_keys`, так что эта информация избыточна.

Визуальное представление плана выполнения

Разработчики MySQL говорили, что неплохо было бы выводить результат работы `EXPLAIN` в виде дерева, поскольку оно может более точно отразить структуру плана выполнения. Того же хотели бы и пользователи. Имеющееся представление не очень удобно для изучения плана, но древовидное отображение сложно совместить с табличным форматом. Неудобство особенно наглядно проявляется в большом количестве возможных значений в столбце `Extra`, а также в представлении `UNION`. Конструкция `UNION` совершенно не похожа на другие виды соединений, выполняемых MySQL, поэтому плохо сочетается с форматом `EXPLAIN`.

Хорошо разбираясь в различных правилах и особенностях `EXPLAIN`, можно восстановить изначально древовидный план выполнения. Но это утомительное занятие, которое лучше оставить автоматическому инструменту. Как раз такой инструмент, сценарий `mk-visual-explain`, имеется в комплекте Maatkit (см. главу 14).

C

Использование Sphinx совместно с MySQL

Sphinx (*http://www.sphinxsearch.com*) – это бесплатная полнотекстовая поисковая система с открытым исходным кодом, которая изначально спроектирована в расчете на интеграцию с базами данных. Она обладает рядом средств, характерных для СУБД, работает очень быстро, поддерживает распределенный поиск и хорошо масштабируется. Кроме того, в проект заложено эффективное использование памяти и минимизация дискового ввода/вывода, что особенно важно, так как часто именно эти факторы ограничивают производительность крупных операций.

Sphinx прекрасно работает в сочетании с MySQL. Ее можно использовать для ускорения разнообразных запросов, в том числе и полнотекстовых, а также для выполнения быстрой сортировки и группировки – и это далеко не все возможные применения. Кроме того, существует реализованная в виде подключаемого модуля подсистема хранения, с помощью которой программист или администратор может обращаться к Sphinx напрямую из MySQL. Система Sphinx особенно полезна для некоторых запросов, которые MySQL не очень хорошо оптимизирует для больших наборов данных вследствие универсальности своей архитектуры. Короче говоря, Sphinx дополняет функциональность сервера MySQL и повышает его производительность.

Источником данных для индекса Sphinx обычно является запрос SELECT к MySQL, но, вообще говоря, можно построить индекс на основе неограниченного количества источников разных типов, а каждый экземпляр Sphinx способен искать в произвольном числе индексов. Например, можно включить в индекс некоторые документы из базы данных MySQL, работающей на одном удаленном сервере, документы из базы данных PostgreSQL, работающей на другом сервере, и данные, поставляемые локальным сценарием по XML-конвейеру.

В этом приложении мы рассмотрим ряд случаев, в которых возможности Sphinx позволяют повысить производительность, опишем порядок установки и конфигурирования, подробно расскажем о функциях системы и обсудим реальные примеры практического применения.

Обзор: типичный поиск с помощью Sphinx

Начнем с простого, но достаточно показательного примера использования Sphinx, чтобы наметить отправную точку для дальнейшего обсуждения. Мы воспользуемся языком PHP из-за его популярности, но API существует и для ряда других языков.

Предположим, что нужно реализовать полнотекстовый поиск для использования в системе сравнения товаров. К ней предъявляются следующие требования:

- Поддерживать полнотекстовый индекс для поиска по таблице товаров, хранящейся в базе данных MySQL
- Обеспечить полнотекстовый поиск по названию и описанию каждого товара
- При необходимости производить поиск только в пределах заданной категории
- Иметь возможность сортировать результаты не только по релевантности, но и по цене товара или дате поступления в продажу

Для начала опишем источник данных и индекс в конфигурационном файле Sphinx:

```
source products
{
    type        = mysql
    sql_host    = localhost
    sql_user    = shopping
    sql_pass    = mysecretpassword
    sql_db      = shopping
    sql_query   = SELECT id, title, description, \
                  cat_id, price,
                  UNIX_TIMESTAMP(added_date) \
                  AS added_ts FROM products
    sql_attr_uint        = cat_id
    sql_attr_float       = price
    sql_attr_timestamp   = added_ts
}

index products
{
    source    = products
    path      = /usr/local/sphinx/var/data/products
    docinfo   = extern
}
```

В этом примере предполагается, что в базе данных `shopping` имеется таблица `products`, из которой мы выбираем в запросе `SELECT` столбцы для заполнения индекса Sphinx. Сам индекс Sphinx также называется `products`. Описав новый источник данных и индекс, мы запускаем программу *indexer*, которая создает начальный полнотекстовый индекс, а затем запускаем или перезапускаем программу-демон *searchd*, которая подхватит изменения:

```
$ cd /usr/local/sphinx/bin
$ ./indexer products
$ ./searchd --stop
$ ./searchd
```

Теперь индекс готов отвечать на запросы. Чтобы протестировать его, запустим сценарий *test.php*, входящий в дистрибутив Sphinx:

```
$ php -q test.php -i products ipod

Query 'ipod ' retrieved 3 of 3 matches in 0.010 sec.
Query stats:
    'ipod' found 3 times in 3 documents

Matches:
1. doc_id=123, weight=100, cat_id=100, price=159.99, added_ts=2008-01-03 22:38:26
2. doc_id=124, weight=100, cat_id=100, price=199.99, added_ts=2008-01-03 22:38:26
3. doc_id=125, weight=100, cat_id=100, price=249.99, added_ts=2008-01-03 22:38:26
```

Последний шаг – добавить функцию поиска в веб-приложение. Мы должны настроить параметры сортировки и фильтрации в соответствии с пожеланиями пользователя и красиво отформатировать результаты. Кроме того, поскольку Sphinx возвращает клиенту только идентификаторы документов и прописанные в конфигурационном файле атрибуты (он не хранит исходные текстовые данные), то недостающую информацию мы должны получить от MySQL самостоятельно:

```
 1 <?php
 2 include ( "sphinxapi.php" );
 3 // ... прочие директивы include, код подключения к MySQL,
 4 // вывод заголовка страницы, формы поиска и т. п. – все это здесь
 5
 6 // устанавливаем те параметры запроса, которые задал пользователь
 7 $cl = new SphinxClient ( );
 8 $sortby = $_REQUEST["sortby"];
 9 if ( !in_array ( $sortby, array ( "price", "added_ts" ) ) )
10    $sortby = "price";
11 if ( $_REQUEST["sortorder"]=="asc" )
12     $cl->SetSortMode ( SPH_SORT_ATTR_ASC, $sortby );
13 else
14     $cl->SetSortMode ( SPH_SORT_ATTR_DESC, $sortby );
15 $offset = ($_REQUEST["page"]-1)*$rows_per_page;
16 $cl->SetLimits ( $offset, $rows_per_page );
17
```

```
// отправляем запрос, получаем результаты
$res = $cl->Query ( $_REQUEST["query"], "products" );

// обрабатываем ошибки поиска
if ( !$res )
{
    print "<b>Ошибка поиска:</b>" . $cl->GetLastError ( );
    die;
}

// выбираем дополнительные столбцы из базы данных MySQL
$ids = join ( ",", array_keys ( $res["matches"] ) );
$r = mysql_query ( "SELECT id, title FROM products WHERE id IN ($ids)" )
    or die ( "Ошибка MySQL: " . mysql_error( ) );
while ( $row = mysql_fetch_assoc($r) )
{
    $id = $row["id"];
    $res["matches"][$id]["sql"] = $row;
}

// выводим результаты в том порядке, в котором их вернула система Sphinx
$n = 1 + $offset;
foreach ( $res["matches"] as $id=>$match )
{
    printf ( "%d. <a href=details.php?id=%d>%s</a>, USD %.2f<br>\n",
        $n++, $id, $match["sql"]["title"], $match["attrs"]["price"] );
}

?>
```

Хотя показанный код несложен, но кое-что в нем все же следует пояснить.

- Функция `SetLimits()` говорит Sphinx, что необходимо извлечь лишь то количество результатов, которое клиент желает вывести на странице. В Sphinx такое ограничение обходится дешево (в отличие от поиска, встроенного в MySQL), а число результатов, которое было бы возвращено, не будь этого ограничения, можно получить, дополнительно не нагружая систему, из переменной `$result['total_found']`.
- Поскольку Sphinx только *индексирует* столбец `title`, но не *хранит* его, мы должны извлекать данные непосредственно из базы MySQL.
- Мы выбираем значения из базы MySQL одним запросом для всех документов, перечисляя их идентификаторы во фразе `WHERE id IN (...)`, а не выполняем отдельные запросы для каждого найденного документа (это было бы неэффективно).
- Полученные от MySQL строки мы вставляем в результат полнотекстового поиска, чтобы сохранить исходный порядок сортировки. Чуть ниже мы еще вернемся к этому пункту.

- Отображаемые строки сконструированы из значений, полученных как от Sphinx, так и от MySQL.

Код включения данных в отображаемые строки специфичен для PHP и заслуживает чуть более подробного объяснения. Мы не можем просто обойти результирующий набор, возвращенный сервером MySQL, поскольку порядок строк в нем, как правило, отличается от последовательности идентификаторов во фразе `WHERE id IN (...)`. Однако в хеше (ассоциативном массиве) PHP результаты хранятся в том порядке, в котором были помещены, поэтому при обходе `$result["matches"]` мы формируем строки в той последовательности, в какой их вернула система Sphinx. Таким образом, чтобы сохранить требуемый порядок результатов (вместо полуслучайного порядка, в котором возвращает строки MySQL), мы вставляем полученные от MySQL данные по одному в хеш, где PHP хранит результирующий набор Sphinx.

Между MySQL и Sphinx имеется существенная разница как в реализации, так и в производительности подсчета количества найденных результатов и применения фразы `LIMIT`. Во-первых, сразу отметим, что `LIMIT` обходится в Sphinx очень дешево. Рассмотрим фразу `LIMIT 500,10`. MySQL выбрала бы 510 первых попавшихся строк (это медленно) и 500 из них отбросила бы, тогда как Sphinx возвращает идентификаторы, с помощью которых вы позже выберете из базы MySQL именно те 10 строк, которые нужны. Во-вторых, Sphinx всегда возвращает точное количество найденных записей, вне зависимости от того, что задано во фразе `LIMIT`. MySQL не умеет делать это эффективно (см. раздел «Оптимизация SQL_CALC_FOUND_ROWS» главы 4 на стр. 249).

Зачем использовать Sphinx?

Sphinx может дополнить приложение на основе MySQL разными способами, демонстрируя высокую производительность там, где MySQL не блещет, и предлагая функциональность, которой MySQL не располагает. Перечислим некоторые типичные сценарии использования:

- Быстрый, эффективный, масштабируемый и релевантный полнотекстовый поиск
- Оптимизация условий `WHERE`, когда индекс отсутствует или обладает низкой селективностью
- Оптимизация запросов с фразами `ORDER BY ... LIMIT N` и `GROUP BY`
- Параллельная генерация результирующих наборов
- Масштабирование по вертикали и по горизонтали
- Агрегирование секционированных данных

В следующих разделах мы рассмотрим все эти сценарии. Но этот список не исчерпывающий, и пользователи Sphinx постоянно находят новые применения. Например, один из наиболее важных способов прак-

тической работы с системой Sphinx – быстрый поиск и фильтрация записей – результат творчества пользователей, оно не закладывалось при проектировании.

Эффективный и масштабируемый полнотекстовый поиск

Средства полнотекстового поиска, встроенные в MySQL[1], хороши для небольших наборов данных, но с ростом базы начинают работать очень медленно. Когда количество записей исчисляется миллионами, а объем индексированного текста – гигабайтами, время выполнения запроса может варьироваться от секунды до 10 минут и более, что для высокопроизводительного веб-приложения, конечно, неприемлемо. Полнотекстовый поиск средствами MySQL допускает масштабирование за счет распределения данных по нескольким серверам, но организовывать параллельный поиск и объединение найденных результатов придется в приложении.

Sphinx работает значительно быстрее, чем встроенные в MySQL полнотекстовые индексы. Например, на поиск в тексте размером свыше 1 Гбайта затрачивается от 10 до 100 миллисекунд, и при объеме до 10–100 Гбайт время линейно зависит от количества процессоров. Кроме того, система Sphinx обладает следующими достоинствами:

- Может индексировать данные, хранящиеся в InnoDB и других подсистемах хранения, а не только в MyISAM.
- Может создавать индексы по данным, взятым из многих таблиц, а не только из столбцов одной таблицы.
- Может динамически объединять результаты поиска по нескольким индексам.
- Помимо текстовых столбцов индексы могут содержать неограниченное количество числовых *атрибутов*, их можно считать в некотором роде «дополнительными столбцами»[2]. В качестве значений атрибутов допускаются целые числа, числа с плавающей точкой и временные метки UNIX.
- Умеет оптимизировать полнотекстовый поиск с дополнительными условиями на атрибуты.
- Алгоритм ранжирования, учитывающий полные фразы, позволяет возвращать наиболее релевантные результаты. Например, при поиске в таблице песен о любви по запросу «Я люблю тебя, родная» песня, содержащая в точности эту фразу, окажется в начале списка – раньше тех, что несколько раз содержат слова «люблю» и «родная».

---

[1] См. раздел «Полнотекстовый поиск» в главе 5 на стр. 307.

[2] Их иногда используют для фильтрации и сортировки результатов поиска. – *Прим. науч. ред.*

- Значительно упрощает масштабирование по горизонтали. Подробнее о масштабировании см. главу 9 и раздел «Масштабирование» на стр. 767 ниже в этом приложении.

Эффективное применение фразы WHERE

Иногда требуется осуществить выборку из очень больших таблиц (с миллионами записей), а в запросе присутствуют условия `WHERE` для столбцов, по которым построены индексы с очень низкой селективностью (то есть для заданного условия возвращается слишком много строк) или индексы вообще не построены. Типичные случаи – поиск пользователей в социальной сети или лотов на аукционном сайте. Как правило, в форме поиска можно задать 10 и более столбцов, а сортироваться результаты могут совсем по другим столбцам. Пример такого приложения и соответствующей стратегии индексирования см. в разделе «Практические примеры индексирования» главы 3 (стр. 176).

Если правильно спроектировать схему и провести оптимизацию запросов, то MySQL вполне прилично справляется с такими ситуациями, если в условии `WHERE` не слишком много столбцов. Но по мере роста количества столбцов число индексов, необходимых для поддержки всех возможных комбинаций, растет экспоненциально. Уже полное покрытие всех возможных сочетаний четырех столбцов опасно близко к пределам возможностей MySQL. К тому же поддерживать такие индексы очень медленно и дорого. Следовательно, практически бессмысленно строить все требуемые индексы для поддержки условий `WHERE` с большим числом столбцов.

Но еще важнее тот факт, что даже добавив индексы, толку от них вы получите немного, если они обладают низкой селективностью. Классический пример – столбец `gender` (пол), который мало чем помогает, потому что каждому значению соответствует примерно половина всех строк. Если селективность индекса недостаточна, то MySQL обычно предпочитает полное сканирование таблицы.

Sphinx выполняет такие запросы гораздо быстрее, чем MySQL. При построении индекса Sphinx можно указать только требуемые столбцы из таблицы с данными. Sphinx допускает два способа доступа к таблицам: поиск ключевого слова по индексу или полное сканирование. В обоих случаях Sphinx применяет фильтры – эквивалент фразы `WHERE`. Но в отличие от MySQL, которая сама определяет, использовать ли индекс или прибегнуть к полному сканированию, Sphinx позволяет пользователю выбирать метод доступа.

Чтобы воспользоваться полным сканированием с фильтрами, следует задать в качестве критерия поиска пустую строку. Если же нужен поиск по индексу, то при его построении включите в поисковые поля псевдоключевые слова, а затем ищите по ним. Например, если требуется найти товары в категории 123, то на этапе индексирования нужно добавить в документ ключевое слово «категория123» и затем искать по

нему. Добавлять слова можно в одно из уже имеющихся полей с помощью функции CONCAT(), а можно для пущей гибкости создать специальное поисковое поле для псевдоключевых слов. Обычно стоит использовать фильтры, когда критерий неселективен, то есть ему удовлетворяет больше 30% строк, и фиктивные ключевые слова – когда критерию удовлетворяет менее 10% строк. В серой зоне от 10 до 30% возможно всякое, так что для поиска оптимального решения следует воспользоваться эталонным тестированием.

Sphinx выполняет и поиск по индексу, и полное сканирование быстрее, чем MySQL. Иногда для полного сканирования Sphinx требуется даже меньше времени, чем MySQL на чтение из индекса.

Поиск первых нескольких результатов с учетом сортировки

В веб-приложениях часто нужно искать первые *N* результатов с учетом заданной сортировки. В разделе «Оптимизация LIMIT со смещением» главы 4 (стр. 248) мы говорили о том, что этот случай плохо поддается оптимизации в MySQL.

Хуже всего дело обстоит, когда поиск по указанному условию WHERE возвращает много строк (допустим, 1 миллион), а столбцы, перечисленные во фразе ORDER BY, не проиндексированы. MySQL воспользуется индексом для поиска всех подходящих строк, будет читать их одну за другой в буфер сортировки, собирая по всему диску, затем произведет сортировку (filesort) – и все это только для того, чтобы большую часть отбросить. На время обработки запроса MySQL вынуждена сохранять и обрабатывать все строки, игнорируя ограничение LIMIT и потребляя огромное количество памяти. А если результат не помещается в буфер сортировки, то придется прибегнуть к сортировке на диске, что лишь увеличивает число операций дискового ввода/вывода.

Это крайний случай, и может даже показаться, что в реальной жизни такое случается редко, но проблемы, которые он иллюстрирует, возникают сплошь и рядом. MySQL ограничивает множество индексов, пригодных для сортировки: допускается использование только левой части ключа, не поддерживается неплотный просмотр индекса (loose index scan) и разрешается только условие с указанием диапазона. Поэтому многие реальные запросы ничего не выигрывают от наличия индексов. А даже если выигрывают, то дисковый ввод/вывод для извлечения строк в полуслучайном порядке способен похоронить производительность.

Еще одной проблемой в MySQL является разбиение результирующего набора на страницы, для которого обычно требуется выполнять запросы типа SELECT ... LIMIT *N*, *M*. При этом с диска читается *N* + *M* строк и, следовательно, производится много операций произвольной выборки, вследствие чего впустую расходуется память. Sphinx может существен-

но ускорить выполнение таких запросов, устраняя две самые крупные проблемы.

Потребление памяти

Потребление памяти в Sphinx всегда строго ограничено, причем лимиты можно конфигурировать. Sphinx поддерживает размер и смещение результирующего набора аналогично конструкции `LIMIT N, M`, применяемой в MySQL, но допускает также параметр `max_matches`. Он управляет размером аналога «буфера сортировки» как на уровне сервера, так и на уровне отдельного запроса. Гарантируется, что потребление памяти в Sphinx не превышает заданных ограничений.

Ввод/вывод

Если атрибуты находятся в памяти, то Sphinx вообще не обращается к диску. Но даже если атрибуты хранятся на диске, то для их считывания Sphinx выполняет последовательный ввод/вывод, что гораздо быстрее извлечения в полуслучайном порядке, характерного для MySQL.

Результаты поиска можно сортировать по комбинации релевантности (веса), значений атрибутов и (при использовании `GROUP BY`) значений агрегатной функции. Синтаксис задания порядка сортировки аналогичен фразе `ORDER BY` в SQL:

```
<?php
$cl = new SphinxClient ( );
$cl->SetSortMode ( SPH_SORT_EXTENDED, 'price DESC, @weight ASC' );
// дополнительный код и вызов Query() ...
?>
```

В этом примере `price` – это заданный пользователем атрибут, хранящийся в индексе, а `@weight` – специальный атрибут, создаваемый во время выполнения и содержащий вычисленную релевантность результата. Можно сортировать также по арифметическому выражению, включающему значения атрибутов, стандартные математические операции и функции:

```
<?php
$cl = new SphinxClient ( );
$cl->SetSortMode ( SPH_SORT_EXPR, '@weight + log(pageviews)*1.5' );
// дополнительный код и вызов Query() ...
?>
```

Оптимизация запросов, содержащих GROUP BY

Поддержка типичных для языка SQL фраз была бы неполной без функциональности `GROUP BY`, поэтому в Sphinx она тоже реализована. Но в отличие от универсального механизма в MySQL, Sphinx специализируется на эффективном решении полезного на практике подмножества задач, решаемых конструкцией `GROUP BY`. Это подмножество покрывает

генерацию отчетов по большим (от 1 до 100 миллионов строк) наборам данных в случаях, когда выполняется одно из следующих условий:

- Результирующий набор составляет лишь «малую толику» от общего числа группируемых строк (здесь под «малой толикой» может пониматься от 100 000 до миллиона строк).
- Требуется очень высокая скорость работы, поэтому можно смириться с тем что значение `COUNT(*)` будет приближенным, если много групп извлекается из данных, распределенных между машинами, входящими в кластер.

Эти условия не настолько ограничительны, как может показаться. Первый случай покрывает практически все мыслимые отчеты по времени. Например, для построения детального почасового отчета за 10 лет нужно извлечь менее 90 000 записей. Второй случай можно на естественном языке описать так: «найти 20 наиболее значимых записей в секционированной таблице из 100 миллионов строк настолько быстро и точно, насколько это возможно».

Эта техника позволяет ускорить выполнение запросов общего вида, но ее можно использовать и в приложениях с полнотекстовым поиском. Часто бывает нужно не просто показать результаты полнотекстового поиска, но и агрегировать их тем или иным способом. Например, на многих страницах поисковой выдачи отображается количество результатов, найденных в каждой категории товаров, или выводится график изменения счетчика обнаруженных документов в зависимости от времени. Еще одно типичное требование – сгруппировать результаты и показать наиболее релевантные в каждой категории. Поддержка `GROUP BY` в Sphinx позволяет комбинировать группировку с полнотекстовым поиском и тем самым избежать накладных расходов на группировку в приложении или в MySQL.

Как и в случае сортировки, для группировки в Sphinx отведена память фиксированного размера. Она работает немного более эффективно, чем аналогичные запросы в MySQL (от 10 до 50%), в случае, когда набор данных целиком умещается в памяти. При таком условии преимущество системы Sphinx обусловлено ее способностью распределять нагрузку и существенно уменьшать время задержки. Для очень больших наборов данных, которые в память никак не уместить, можно в целях формирования отчетов построить специальный индекс на диске с помощью встраиваемых атрибутов (подробнее об этом ниже). Запросы к таким индексам выполняются примерно с той же скоростью, с какой можно считывать данные с диска, – на современном оборудовани 30–100 Мбит/с. В таком случае производительность оказывается многократно выше, чем у MySQL, хотя результаты получаются приближенными.

Самое важное различие между реализациями `GROUP BY` в MySQL и Sphinx состоит в том, что Sphinx при определенных условиях дает примерные результаты. Тому есть две причины.

- Для группировки используется память фиксированного размера. Если групп больше, чем может поместиться в памяти, а результаты возвращаются в некотором «несчастливом» порядке, то счетчики по группам могут оказаться меньше истинных значений.
- При распределенном поиске между узлами пересылаются только агрегированные результаты, а не сами найденные документы. Если в разных узлах имеются записи-дубликаты, то счетчики различающихся записей в каждой группе могут оказаться больше истинных значений, поскольку информация, которая позволила бы устранить дубликаты, между узлами не передается.

На практике часто можно смириться с приближенными счетчиками по группам, лишь бы они быстро вычислялись. Если же это недопустимо, то часто удается получить точные результаты, соответствующим образом настроив демон и клиентское приложение.

Можно также получить эквивалент конструкции `COUNT(DISTINCT <attribute>)`. Например, этим можно воспользоваться на аукционном сайте для подсчета различных продавцов в каждой категории.

Наконец, Sphinx позволяет задать критерий для выбора одного «наилучшего» документа в каждой группе. В частности, можно выбрать наиболее релевантный документ в каждом домене в ходе группировки по домену и сортировки результирующего набора по счетчику найденных в домене. В MySQL для этого понадобился бы очень сложный запрос.

Генерация параллельных результирующих наборов

Sphinx позволяет одновременно генерировать несколько результирующих наборов из одних и тех же данных, опять-таки не выходя за пределы ограничений по памяти. Это дает ощутимый выигрыш по сравнению с традиционно применяемым в SQL подходом, когда либо запускается два запроса (в надежде, что после первого какие-то данные останутся в кэше), либо для каждого результирующего набора создается временная таблица.

Предположим, к примеру, что нужно сгенерировать отчеты по дням, по неделям и по месяцам за некоторый период. Для этого в MySQL пришлось бы выполнить три запроса с разными фразами `GROUP BY`, то есть обратиться к источнику данных трижды. А Sphinx умеет генерировать все три отчета параллельно, обрабатывая источник всего один раз.

Для этого в Sphinx предусмотрен механизм «мультизапросов». Вместо того чтобы выдавать запросы по одному, вы собираете их в пакет и отправляете один раз:

```
<?php
$cl = new SphinxClient ( );
$cl->SetSortMode ( SPH_SORT_EXTENDED, “price desc” );
$cl->AddQuery ( “ipod” );
$cl->SetGroupBy ( “category_id”, SPH_GROUPBY_ATTR, “@count desc” );
```

```
$cl->AddQuery ( “ipod” );
$cl->RunQueries ( );
?>
```

Sphinx проанализирует поступивший запрос, выделит из него отдельные части и, если возможно, распараллелит их выполнение.

Например, Sphinx может заметить, что отличаются только режимы сортировки и группировки, а в остальном запросы одинаковы. Именно так обстоит дело в примере выше, где сортировка производится по полю `price`, а группировка – по `category_id`. Для обработки таких запросов Sphinx создаст несколько очередей сортировки. Затем каждая выбранная строка будет помещена во все очереди. По сравнению с последовательным выполнением запросов это устраняет излишние операции полнотекстового поиска или полного сканирования.

Отметим, что генерация параллельных результирующих наборов, будучи распространенной и важной оптимизацией, является лишь частным случаем более общего механизма мультизапросов. Это не единственная возможная оптимизация. Можно посоветовать всюду, где возможно, объединять несколько запросов в один пакет, так как это позволяет Sphinx применить различные внутренние оптимизации. Даже если Sphinx не сумеет распараллелить выполнение запросов, все равно количество обращений к серверу уменьшится. А если в будущих версиях появятся новые оптимизации, то они будут применены к вашим запросам автоматически, так что вам не потребуется ничего изменять.

Масштабирование

Sphinx отлично масштабируется как по горизонтали, так и по вертикали. Эту систему можно распределять по нескольким компьютерам без ограничений. Во всех вышеупомянутых способах применения можно получить выигрыш от распределения нагрузки между несколькими процессорами.

Поисковый демон Sphinx (*searchd*) поддерживает специальные *распределенные индексы*, которые понимают, какие локальные и удаленные индексы следует опрашивать и агрегировать. Следовательно, горизонтальное масштабирование становится тривиальной задачей. Нужно лишь распределить данные по узлам и сконфигурировать главный, который будет рассылать удаленные запросы, выполняющиеся параллельно с локальными. Вот и все.

Допустимо также вертикальное масштабирование, когда с целью уменьшения задержки на одной машине устанавливается большее количество процессоров или ядер. Для этого нужно лишь запустить на одном компьютере несколько экземпляров демона *searchd* и опрашивать их все с другого компьютера через распределенный индекс. Или сконфигурировать единственный экземпляр так, чтобы он общался сам с собой;

тогда параллельные «удаленные» запросы на самом деле будут выполняться на одной машине, но разными процессорами или ядрами.

Иными словами, в Sphinx можно сделать так, что один запрос будет исполняться несколькими процессорами (несколько параллельных запросов автоматически обрабатывают разные процессоры). Это существенное отличие от MySQL, где одному запросу всегда выделяется один процессор вне зависимости от того, сколько их имеется в наличии. Кроме того, Sphinx не нуждается в синхронизации между параллельно выполняющимися запросами. Это позволяет отказаться от мьютексов (механизм синхронизации), которые являются печально известной причиной многих узких мест в работе MySQL на машине с несколькими процессорами.

Еще один важный аспект вертикального масштабирования – это масштабирование дискового ввода/вывода. Чтобы повысить пропускную способность и уменьшить задержку, различные индексы (в том числе, части большого распределенного индекса) можно поместить на разные физические диски или тома RAID. Этому подходу присущи некоторые из достоинств механизма секционированных таблиц в версии MySQL 5.1, который также позволяет размещать данные в нескольких местах. Однако у распределенных индексов есть преимущества по сравнению с секционированными таблицами. Sphinx использует их как для разделения нагрузки, так и для параллельной обработки частей запроса. Что же касается механизма секционирования в MySQL, то он позволяет оптимизировать некоторые (не все) запросы за счет отсечения секций, но никак не распараллелить обработку. И хотя методики секционирования и в Sphinx, и в MySQL повышают пропускную способность, но для запросов, требующих большого объема ввода/вывода, от Sphinx всегда можно ожидать линейного снижения задержки, а от MySQL – только в тех случаях, когда оптимизатору удается отсечь некоторые секции целиком.

Последовательность операций при распределенном поиске бесхитростна:

1. Разослать запросы всем удаленным серверам.
2. Выполнить последовательный локальный поиск по индексу.
3. Получить частичные результаты поиска от удаленных серверов.
4. Объединить все частичные результаты в окончательный и вернуть его клиенту.

Если оборудование позволяет, то можно запустить параллельный поиск по набору индексов и на одной машине. При наличии нескольких физических дисков и процессорных ядер эти одновременно выполняющиеся поиски не будут мешать друг другу. Можно сделать вид, что некоторые индексы являются удаленными, и сконфигурировать *searchd* так, чтобы он обращался сам к себе для запуска параллельного запроса на той же машине:

```
index distributed_sample
{
```

```
        type = distributed
        local = chunk1 # resides on HDD1
        agent = localhost:3312:chunk2 # находится на HDD2,
                        # searchd обращается сам к себе
    }
```

С точки зрения клиента, распределенные индексы абсолютно ничем не отличаются от локальных. Это позволяет создавать «деревья» распределенных индексов, используя одни узлы как прокси для других. Например, узел первого уровня мог бы транслировать запросы нескольким узлам второго уровня, котрые либо обрабатывают их локально, либо передают дальше, причем глубина дерева не ограничена.

Агрегирование секционированных данных

При построении масштабируемой системы часто применяется секционирование (sharding) информации на несколько физических серверов MySQL. Этот вопрос мы подробно обсуждали в разделе «Секционирование данных» главы 9 (стр. 516).

Если секционирование детальное, то просто для выборки нескольких строк с селективным условием `WHERE` (вообще-то, это быстрая операция) необходимо обратиться к нескольким серверам, проконтролировать ошибки и объединить результаты в самом приложении. Sphinx отчасти решает эту проблему, так как вся необходимая функциональность уже реализована внутри поискового демона.

Рассмотрим пример. Пусть таблица размером 1 Тбайт, содержащая миллиард сообщений в блогах, секционирована по идентификатору пользователя и размещена на 10 физических серверах MySQL, так что все сообщения одного пользователя всегда оказываются на одном и том же сервере. Если в запросе фигурирует только один пользователь, то все хорошо: выбираем сервер по идентификатору пользователя и дальше работаем как обычно.

Но предположим, что требуется реализовать страницы архива, на которых отображаются сообщения друзей данного пользователя. Как мы будем выводить страницу 50, на которой показываются записи с 981 по 1000, отсортированные по дате сообщения? Скорее всего, данные, относящиеся к разным друзьям, находятся на разных серверах. Если друзей всего 10, то вероятность того, что понадобится обращаться более чем к 8 серверам, составляет 90%, а когда друзей 20, эта вероятность возрастает до 99%. Таким образом, для большинства запросов придется опрашивать все серверы. Более того, нам предстоит отобрать 1000 сообщений с каждого сервера и отсортировать их в приложении. Следуя рекомендациям, приведенным в главе 10 и в других местах, мы могли бы свести необходимые сведения только к идентификатору сообщения и к временной метке, но все равно придется отсортировать 10 000 записей внутри приложения. У большинства современных сценарных языков только на эту сортировку уйдет масса процессорного времени. И не

забудем, что опрашивать серверы надо будет последовательно (это медленно) или написать какой-то код, который запустит параллельные потоки (а его трудно реализовать и сопровождать).

В таких ситуациях имеет смысл не изобретать велосипед, а обратиться к Sphinx. Всего-то и нужно запустить несколько экземпляров Sphinx, скопировать наиболее востребованные атрибуты из каждой таблицы – в данном случае идентификатор сообщения, его дату и идентификатор пользователя, – и запросить у главного экземпляра Sphinx записи с 981 по 1000, отсортированные по дате публикации. На все про все требуется три строчки кода. Такой способ масштабирования куда разумнее.

Обзор архитектуры

Система Sphinx представляет собой автономный набор программ. Из них наиболее важны две:

indexer

Это приложение извлекает документы из указанных источников (например, из результата выполнения запроса к MySQL) и строит по ним полнотекстовый индекс. Обычно оно периодически запускается в пакетном режиме.

searchd

Программа-демон, которая выполняет запросы к индексам, построенным программой *indexer*. Приложения обращаются к ней во время выполнения.

В дистрибутив Sphinx входят также API обращения к *searchd* для нескольких языков программирования (на момент написания книги в этот список входили PHP, Python, Perl, Ruby и Java) и клиент SphinxSE, реализованный в виде подключаемой подсистемы хранения для версии MySQL 5.0 и более поздних. Данные API и SphinxSE позволяют клиентскому приложению подключиться к демону *searchd*, передать ему поисковый запрос и получить результаты поиска.

Полнотекстовый индекс Sphinx можно сравнить с таблицей базы данных, но состоит он не из строк, а из документов. В Sphinx есть также отдельная структура данных, называемая многозначным атрибутом, мы обсудим ее ниже. У каждого документа есть уникальный 32- или 64-разрядный целочисленный идентификатор, который должен извлекаться из индексируемой таблицы базы данных (например, из столбца, содержащего первичный ключ). Кроме того, в каждом документе имеется одно или несколько полнотекстовых полей (одно поле соответствует одному текстовому столбцу в базе данных) и числовых атрибутов. Как и в таблице базы данных, в одном индексе Sphinx состав полей и атрибутов во всех документах одинаков. В табл. С.1 проведена аналогия между таблицей базы данных и индексом Sphinx.

Таблица С.1. Соответствие между структурами базы данных и индекса Sphinx

Структура базы данных	Структура индекса Sphinx
CREATE TABLE documents (id' int(11) NOT NULL auto_increment, title' varchar(255), content' text, group_id' int(11), added' datetime, PRIMARY KEY (id));	index documents document ID title field, full-text indexed content field, full-text indexed group_id attribute, sql_attr_uint added' attribute, sql_attr_timestamp

Sphinx не хранит текстовые поля из базы данных, а лишь использует их содержимое для построения индекса.

Общие сведения об установке

Установка Sphinx не вызывает затруднений и обычно сводится к следующим шагам:

1. Сборка программ из исходных текстов:

```
$ configure && make && make install
```

2. Создание конфигурационного файла, содержащего описания источников данных и полнотекстовые индексы.
3. Первоначальное индексирование.
4. Запуск *searchd*.

После этого функциональность поиска сразу же становится доступна клиентским программам:

```
<?php
include ( 'sphinxapi.php' );
$cl = new SphinxClient ( );
$res = $cl->Query ( 'test query', 'myindex' );
// здесь можно использовать результат поиска $res
?>
```

Остается только регулярно запускать индексатор для обновления полнотекстового индекса. Индексы, с которыми в данный момент работает демон *searchd*, остаются полностью функциональными во время переиндексации; индексатор видит, что они используются, создает «теневой» индекс и по завершении уведомляет *searchd*, что нужно переключиться на него.

Полнотекстовые индексы хранятся в файловой системе (в каталоге, который указан в конфигурационном файле) в специальном «монолитном» формате, плохо приспособленном для инкрементных обновлений. Обычный способ обновления данных в индексе – полная перестройка.

Но это не такая серьезная проблема, как могло бы показаться, по следующим причинам:

- Индексирование производится быстро. На современном оборудовании система Sphinx способна индексировать простой текст (без HTML-разметки) со скоростью 4–8 Мбит/с.
- В предыдущем разделе показано, что данные можно секционировать на несколько индексов и при каждом запуске *indexer* переиндексировать только изменившиеся части.
- Нет необходимости «дефрагментировать» индексы – они строятся в расчете на оптимальное использование подсистемы ввода/вывода, что повышает скорость поиска.
- Числовые атрибуты можно обновлять без полной перестройки индекса.

В будущих версиях планируется реализовать дополнительные средства, которые будут поддерживать обновление индексов в реальном масштабе времени.

Типичное применение секционирования

Рассмотрим секционирование более подробно. Простейшая схема секционирования называется *главный + разностный*, при этом создается два индекса по одному и тому же набору документов. *Главный* индекс построен по всем документам, а *разностный* – только по тем, что изменились с момента построения главного индекса.

Эта схема прекрасно подходит для многих вариантов модификации данных. В качестве примеров можно привести форумы, блоги, архивы почты и новостей, вертикальные поисковые системы. Данные в этих репозиториях, по большей части, никогда не изменяются после ввода, а число регулярно добавляемых или модифицируемых документов мизерно по сравнению с общим количеством. Поэтому разностный индекс очень мал, и его можно перестраивать часто (например, раз в 1–15 минут). Это эквивалентно индексированию только вновь добавленных строк.

Для изменения ассоциированных с документами атрибутов нет необходимости перестраивать индексы – это можно сделать оперативно с помощью *searchd*. Чтобы пометить строку как удаленную, достаточно просто установить атрибут «deleted» в главном индексе. А для обработки обновлений нужно будет пометить этим атрибутом документы в главном индексе, а затем перестроить разностный. Тогда поиск всех документов, не имеющих признака «deleted», вернет правильный результат.

Отметим, что данные для индексирования могут поступать в виде результата произвольной команды `SELECT` и необязательно только из одной таблицы. На структуру команды `SELECT` не накладывается никаких ограничений. Следовательно, перед индексированием можно подвергнуть данные в базе предварительной обработке. Обычно такая

предобработка включает соединение с другими таблицами, создание дополнительных полей на лету, исключение некоторых полей из индекса и манипулирование значениями.

Специальные средства

Помимо «простого» индексирования и поиска по содержимому базы данных Sphinx предлагает ряд специальных средств. Перечислим наиболее важные:

- Алгоритмы поиска и ранжирования принимают во внимание позиции слов и близость поисковой фразы к содержимому документа
- К документам можно привязывать числовые атрибуты, в том числе многозначные (multi-valued attributes, MЗА)
- Разрешается сортировать, фильтровать и группировать по значениям атрибутов
- Можно создавать фрагменты документов, в которых поисковые слова подсвечены
- Поиск можно производить сразу на нескольких машинах
- Можно оптимизировать запросы, порождающие сразу несколько результирующих наборов из одних и тех же данных
- Подсистема хранения SphinxSE позволяет обращаться к результатам поиска непосредственно из MySQL
- Можно настраивать нагрузку на сервер со стороны Sphinx

Часть этих средств мы уже рассматривали выше, а в настоящем разделе обсудим некоторые из оставшихся.

Ранжирование по близости фразы

Как и другие поисковые системы с открытым исходным кодом, Sphinx запоминает позиции начала каждого слова в документе. Но в отличие от большинства других, она также использует эти позиции для ранжирования результатов поиска, что позволяет возвращать более релевантные результаты.

На окончательный ранг документа может повлиять множество факторов. В большинстве систем для ранжирования используется только частота появления поисковых слов, то есть учитывается, сколько раз каждое слово встречалось в документе. Классическая весовая функция BM25, которую применяют практически все полнотекстовые поисковые системы, придает больший вес тем словам, которые чаще встречаются в конкретном тексте или редко встречаются во всем наборе документов. Обычно результат, возвращенный функцией BM25, считается окончательным значением ранга.

Но Sphinx вычисляет также близость поисковой фразы, то есть величину самого длинного отрезка фразы, встречающегося в документе, выраженную в количестве слов. Например, близость фразы «John Doe Jr» к документу, содержащему текст «John Black, John White Jr, and Jane Dunne», равна 1, потому что никакие два слова, указанные в запросе, не встречаются в нем подряд и в том же порядке, что и в ключевой фразе. Но тот же запрос в применении к документу, содержащему предложение «Mr. John Doe Jr and friends» дает близость 3, поскольку здесь присутствуют все три слова из запроса, причем в том же порядке. Близость той же фразы к документу «John Gray, Jane Doe Jr» равна 2 благодаря наличию отрезка «Doe Jr».

По умолчанию Sphinx ранжирует результаты сначала по близости фразы, а потом по классической весовой функции BM25. Следовательно, буквальные цитаты гарантированно окажутся первыми в списке, цитаты, отличающиеся только одним словом, – под ними и т. д.

Когда и как близость фразы влияет на результаты? Рассмотрим поиск выражения «To be or not to be» (быть или не быть) в 1 000 000 страниц текста. Sphinx поместит страницы, содержащие буквальную цитату, в начало списка результатов, тогда как системы на основе функции BM25 в первую очередь вернут страницы, где чаще всего встречаются слова «to», «be», «or» и «not»; документы же с точной цитатой, в которых слово «to» встречается всего несколько раз, будут зарыты в глубине результатов поиска.

Большинство современных крупных поисковых машин также ранжируют результаты поиска с учетом позиций ключевых слов. Поиск фразы в Google, скорее всего, поместит документы, содержащие буквальную и близкую цитату, в начало списка, а уже потом будут располагаться тексты с «мешком слов» (то есть такие, в которых слова встречаются в произвольном порядке).

Однако для анализа позиций ключевых слов требуется дополнительное время, и иногда этот этап желательно пропустить из соображений производительности. Бывает также, что ранжирование с учетом близости фразы дает неожиданные результаты. Например, поиск тегов в «облаке» лучше вести без учета позиций ключевых слов: не имеет значения, насколько близко друг к другу находятся теги, встретившиеся в документе.

Чтобы обеспечить необходимую гибкость, Sphinx предлагает несколько режимов ранжирования. Помимо принимаемого по умолчанию – близость плюс BM25, можно выбрать и другие, в частности: только весовая функция BM25, вообще без вычисления веса (неплохая оптимизация, если вы не собираетесь сортировать по рангу) и т. д.

Поддержка атрибутов

Любой документ может содержать неограниченное количество числовых атрибутов. Атрибуты определяются пользователем и могут содер-

жать дополнительную информацию, необходимую для решения конкретной задачи, например: идентификатор автора сообщения в блоге, цена товара, идентификатор категории и т. д.

Атрибуты позволяют производить полнотекстовый поиск с дополнительной фильтрацией, сортировкой и группировкой результатов. Теоретически их можно было бы хранить в базе MySQL и извлекать оттуда каждый раз после завершения поиска. Но на практике, если полнотекстовый поиск находит хотя бы сотни или тысячи строк (не так уж много), то последующая выборка их из базы будет недопустимо медленной.

Sphinx поддерживает два способа хранения атрибутов: встраивание в документ и в отдельном внешнем файле. В случае встраивания все значения атрибутов приходится хранить в индексе многократно, по одному разу для каждого вхождения идентификатора документа. Это приводит к разбуханию индекса и увеличению объема ввода/вывода, но сокращает потребление памяти. Внешние же атрибуты требуется предварительно загружать в память на этапе запуска *searchd*.

Как правило, атрибуты помещаются в ОЗУ, так что обычно их хранят отдельно. При этом фильтрация, сортировка и группировка производятся очень быстро, так как для доступа к данным достаточно поиска в памяти. К тому же лишь внешние атрибуты можно обновлять во время выполнения. Встраивание атрибутов следует применять лишь тогда, когда для их хранения не хватает памяти.

Sphinx поддерживает также многозначные атрибуты (МЗА). Такой атрибут состоит из произвольно длинного списка целочисленных значений, ассоциированных с каждым документом. Примерами оправданного применения МЗА могут служить списки идентификаторов тегов, категории товаров и списки управления доступом.

Фильтрация

Имея доступ к атрибутам, Sphinx может использовать их для фильтрации и отвергать потенциальных кандидатов на ранних стадиях поиска. Технически фильтрация производится после того, как проверено, что документ содержит все заданные слова, но перед началом трудоемких вычислений (например, ранжирования). Благодаря такой оптимизации применение Sphinx для комбинирования полнотекстового поиска с фильтрацией и сортировкой может оказаться от 10 до 100 раз быстрее, чем использование Sphinx только для поиска с последующей фильтрацией результатов средствами MySQL.

Sphinx поддерживает два типа фильтров, аналогичных простым условиям `WHERE` в SQL:

- Значение атрибута попадает в заданный диапазон (аналог фразы `BETWEEN` или числового сравнения).
- Атрибут совпадает с одним из заданных значений (аналог списка `IN()`).

Если в фильтрах будет задаваться фиксированное количество величин (фильтры по набору, а не по диапазону), и если эти значения селективные, то имеет смысл заменить целые числа «фиктивными ключевыми словами» и индексировать их как текст, а не как атрибуты. Это относится как к обычным, так и к многозначным числовым атрибутам. Примеры этой техники будут приведены ниже.

Sphinx может также использовать фильтры для оптимизации полного сканирования. Система запоминает минимальное и максимальное значение атрибута для короткого непрерывного блока строк (по умолчанию 128 строк) и может быстро отбрасывать целые блоки, если они не удовлетворяют условиям фильтрации. Строки хранятся в порядке возрастания идентификаторов документов, поэтому такая оптимизация лучше всего работает, если значения столбцов коррелируют с идентификатором. Например, если в таблице имеется временная метка, которая возрастает вместе с идентификатором, то полное сканирование с фильтрацией по этой метке будет выполняться очень быстро.

Подключаемая подсистема хранения SphinxSE

Результаты полнотекстового поиска, полученные от Sphinx, почти всегда требуется затем обработать средствами MySQL – по крайней мере, для того чтобы извлечь значения текстовых столбцов, которые в Sphinx не хранятся. Поэтому нередко возникает необходимость соединить результаты поиска с другими таблицами MySQL, то есть выполнить операцию `JOIN`.

Хотя это можно сделать, отправив MySQL запрос с идентификаторами найденных документов, такая стратегия не позволяет получить элегантный и быстрый код. Если речь идет о больших объемах данных, то лучше воспользоваться подсистемой хранения SphinxSE, которую можно включить при компиляции MySQL версии 5.0 или старше, либо загрузить как подключаемый модуль на сервер версии 5.1 (и старше).

SphinxSE позволяет программисту опрашивать демон *searchd* и получать результаты поиска прямо из MySQL. Для этого достаточно создать специальную таблицу с фразой `ENGINE=SPHINX` (и необязательным параметром `CONNECTION`, который позволяет найти сервер Sphinx, если он не находится там, где подразумевается по умолчанию), а затем использовать ее в запросах:

```
mysql> CREATE TABLE search_table (
    ->    id        INTEGER NOT NULL,
    ->    weight    INTEGER NOT NULL,
    ->    query     VARCHAR(3072) NOT NULL,
    ->    group_id  INTEGER,
    ->    INDEX(query)
    -> ) ENGINE=SPHINX
    -> CONNECTION="sphinx://localhost:3312/test";
Query OK, 0 rows affected (0.12 sec)
```

```
mysql> SELECT * FROM search_table WHERE query='test;mode=all' \G
*************************** 1. row ***************************
      id: 123
  weight: 1
   query: test;mode=all
group_id: 45
1 row in set (0.00 sec)
```

Каждая команда SELECT передает Sphinx запрос, текст которого находится в условии на столбец query во фразе WHERE (см. пример ниже). Сервер *searchd* возвращает результаты. Подсистема хранения SphinxSE преобразует их в формат MySQL и возвращает команде SELECT.

В запросах можно употреблять оператор JOIN для соединения с другими таблицами любого типа.

Подсистема хранения SphinxSE поддерживает большинство возможностей поиска, доступных через API. Чтобы задать, к примеру, фильтрацию и лимиты, следует включить дополнительные параметры в строку запроса:

```
mysql> SELECT * FROM search_table WHERE query='test;mode=all;
    -> filter=group_id,5,7,11;maxmatches=3000';
```

Возвращаемая API статистика по запросам и словам доступна с помощью команды SHOW STATUS:

```
mysql> SHOW ENGINE SPHINX STATUS \G
*************************** 1. row ***************************
  Type: SPHINX
  Name: stats
Status: total: 3, total found: 3, time: 8, words: 1
*************************** 2. row ***************************
  Type: SPHINX
  Name: words
Status: test:3:5
2 rows in set (0.00 sec)
```

Даже при использовании SphinxSE рекомендуется оставить сортировку, фильтрацию и группировку демону *searchd*, то есть включить все необходимые для этого параметры в строку запроса, а не во фразы WHERE, ORDER BY и GROUP BY. Особенно это важно для условий WHERE. Причина в том, что SphinxSE – всего лишь клиент *searchd*, а не полноценная библиотека поиска. Следовательно, для достижения максимальной производительности лучше передавать все, что можно, ядру Sphinx.

Средства управления производительностью

Индексирование и поиск могут создавать дополнительную нагрузку как на поисковый сервер, так и на сервер баз данных. Однако для ограничения нагрузки, создаваемой системой Sphinx, имеется целый ряд параметров.

Нежелательная нагрузка на СУБД может быть вызвана запросами от индексатора, которые либо приводят к полной остановке работы MySQL из-за блокировок либо поступают слишком быстро и отнимают ресурсы у других конкурирующих запросов.

Первая ситуация характерна для MyISAM, здесь длительные блокировки чтения блокируют таблицы и приостанавливают другие операции чтения и записи; выполнять команду `SELECT * FROM big_table` на промышленном сервере просто нельзя, так как есть риск помешать всей остальной работе. Чтобы обойти эту сложность, Sphinx поддерживает запросы с указанием диапазона. Вместо того чтобы конфигурировать один гигантский запрос, можно задать запрос, который быстро вычислит диапазон значений в индексируемых строках, и другой запрос, который будет выбирать данные небольшими порциями:

```
sql_query_range  = SELECT MIN(id),MAX(id) FROM documents
sql_range_step   = 1000
sql_query        = SELECT id, title, body FROM documents \
                   WHERE id>=$start AND id<=$end
```

Эта возможность особенно полезна для индексирования таблиц типа MyISAM, но стоит подумать и о применении ее к таблицам InnoDB. Хотя при выполнении `SELECT *` InnoDB не блокирует таблицы и не мешает работе других запросов, все равно из-за особенностей архитектуры MVCC потребляется много ресурсов. Многоверсионность для тысячи транзакций, каждая из которых охватывает тысячу строк, может обойтись дешевле, чем для одной длительной транзакции, затрагивающей миллион строк.

Второй источник повышенной нагрузки проявляется, когда программа *indexer* обрабатывает данные быстрее, чем MySQL их поставляет. В этом случае также следует использовать запросы с указанием диапазона. Параметр `sql_ranged_throttle` заставляет индексатор приостанавливать работу на заданный промежуток времени (в миллисекундах) перед отправкой очередного запроса. Время индексирования при этом увеличивается, зато нагрузка на MySQL снижается.

Интересно, что существует один особый случай, когда имеет смысл настроить Sphinx для достижения прямо противоположного эффекта: сократить время индексирования за счет повышения нагрузки на MySQL. Если скорость сетевого соединения между машинами, на которых работает индексатор и MySQL, составляет 100 Мбит/с и строки хорошо сжимаются (это типично для текстовых данных), то применение протокола сжатия MySQL может уменьшить общее время индексирования. Правда, за это обеим сторонам приходится расплачиваться дополнительным расходованием процессорного времени на упаковку и распаковку строк, передаваемых по сети. Тем не менее, благодаря сокращению сетевого трафика время индексирования может уменьшиться на 20–30%.

Поисковые кластеры также могут испытывать кратковременные перегрузки, поэтому Sphinx предоставляет несколько способов контроля нагрузки на *searchd*.

Во-первых, параметр `max_children` ограничивает количество одновременно выполняемых запросов. Когда заданный порог достигнут, клиенту предлагается повторить попытку позже.

Кроме того, существуют лимиты на уровне запроса. Так, можно указать, что обработка запроса должна прекращаться, как только найдено максимальное количество документов или истекло отведенное время; для чего предназначены соответственно функции `SetLimits()` и `SetMaxQueryTime()`. Поскольку это делается на уровне запроса, то наиболее важные запросы можно выполнять до конца.

И наконец, при периодическом запуске индексатора могут возникать всплески активности ввода/вывода, что приводит к замедлению работы *searchd*. Во избежание этого предусмотрены параметры для ограничения обращений индексатора к диску. Параметр `max_iops` создает задержку между операциями ввода/вывода, ограничивая максимальное количество операций ввода-вывода в секунду своим значением. Но даже одной операции может оказаться слишком много (что если она читает 100 Мбайт?). На этой случай есть параметр `max_iosize`, гарантирующий, что длина каждой операции чтения или записи не будет превышать заданного значения. Более длинные процессы автоматически разбиваются на части, к которым затем применяется параметр `max_iops`.

Примеры практической реализации

Все описанные выше средства нашли успешное практическое применение в той или иной системе. В следующих разделах мы расскажем о нескольких реальных примерах использования Sphinx, упомянув о конкретных сайтах и о некоторых деталях реализации.

Полнотекстовый поиск на сайте Mininova.org

Популярная система поиска торрентов Mininova.org дает пример оптимизации «одного лишь» полнотекстового поиска. Sphinx заменила встроенные полнотекстовые индексы на нескольких подчиненных серверах MySQL, которые не справлялись с объемами работы. После замены поисковые сервера оказались недогружены, средняя нагрузка составляет примерно 0,3–0,4.

Приведем сведения о размере базы данных и особенностях работы сервера:

- На этом сайте база данных невелика, порядка 300 000–500 000 записей, а размер индекса составляет примерно 300–500 Мбайт.

- Нагрузка на сайт довольно высока: 8–10 миллионов поисков в день (на момент написания книги).

Данные, по большей части, представляют собой заданные пользователями имена файлов, часто без соблюдения пунктуации. Поэтому вместо поиска по целым словам применяется поиск по префиксу. В результате индекс оказывается в несколько раз больше, чем мог бы, но все равно он достаточно мал, так что строится быстро и допускает эффективное кэширование.

Результаты 1000 наиболее частых запросов кэшируются на уровне приложения. Примерно 20–30% запросов обслуживаются из кэша. Поскольку для их распределения характерен «длинный хвост», то увеличение размера кэша не помогло бы.

С целью повышения доступности сайт обслуживается двумя серверами, на каждом из которых хранятся идентичные реплики полнотекстовых индексов. Индексы перестраиваются полностью каждые несколько минут. Поскольку индексирование занимает меньше минуты, смысла в реализации более сложной схемы нет.

Из этого примера можно извлечь следующие уроки.

- Кэширование результатов поиска на уровне приложения весьма способствует повышению производительности.
- Иногда нет необходимости в огромном кэше, даже если приложение сильно нагружено. Может хватать всего 1000–10000 записей.
- Если размер базы данных составляет примерно 1 Гбайт, то даже для нагруженного сайта достаточно периодического полного переиндексирования. Изобретать более хитроумные схемы ни к чему.

Полнотекстовый поиск на сайте BoardReader.com

Mininova представляет один полюс высоконагруженных проектов – данных мало, но к ним обращается очень много запросов. Сайт BoardReader (*http://www.boardreader.com*) первоначально представлял собой прямо противоположный полюс: это система поиска по форумам, которая выполняет гораздо меньше поисков, но на значительно более объемном наборе данных. Sphinx заменила коммерческую поисковую систему, которой требовалось до 10 секунд на поиск в наборе размером 1 Гбайт. Sphinx дала возможность масштабировать BoardReader в широких пределах – как по размеру, так и по пропускной способности.

Приведем информацию общего характера:

- Документов больше миллиарда, объем текста в базе данных порядка 1,5 Тбайт.
- Ежедневно просматривается примерно 500 000 страниц и выполняется от 700 000 до миллиона поисков.

Когда писалась эта книга, поисковый кластер состоял из шести серверов, каждый из которых был оснащен четырьмя логическими ЦП (два двухъядерных процессора Xeon), 16 Гбайт оперативной памяти и дисками совокупной емкостью 0,5 Тбайт. Сама база данных находится на отдельном кластере. Поисковый кластер использовался только для индексирования и поиска.

На каждом из шести серверов запущено по четыре экземпляра *searchd*, чтобы были задействованы все четыре ядра. Один из четырех экземпляров агрегирует результаты от остальных трех. Всего получается 24 экземпляра *searchd*. Данные между ними распределены равномерно. Каждая копия *searchd* обслуживает несколько индексов, в сумме составляющих примерно 1/24 часть общего объема данных (примерно 60 Гбайт).

Результаты поиска, возвращенные шестью узлами «первого уровня», в свою очередь агрегируются еще одним демоном *searchd*, который работает на веб-сервере переднего плана (frontend). Этот экземпляр не содержит локальных данных, но обслуживает несколько распределенных индексов, которые ссылаются на шесть поисковых серверов, входящих в кластер.

Зачем нужно четыре экземпляра *searchd* на одном узле? Почему бы не запустить на каждом сервере по одному экземпляру, сконфигурировав его для обслуживания четырех групп индексов, и не заставить его посылать самому себе «удаленные» запросы, чтобы задействовать несколько ЦП? Такую конфигурацию мы упоминали выше. Но у четырех экземпляров вместо одного есть свои плюсы. Во-первых, сокращается время запуска. Существует несколько гигабайтов атрибутивных данных, которые необходимо предварительно загрузить в память; одновременный запуск нескольких демонов позволяет распараллелить эту задачу. Во-вторых, улучшается доступность. Когда выходит из строя или обновляется один экземпляр *searchd*, недоступной становится только 1/24 часть индекса, а не 1/6.

На каждом из 24 экземпляров *searchd* в поисковом кластере мы применяем секционирование по времени, чтобы еще больше сократить нагрузку. Многие запросы необходимо выполнять только для недавних данных, поэтому мы разделили весь объем информации на три непересекающихся набора индексов: данные за последнюю неделю, за последние три месяца и за все время. Эти индексы распределены по нескольким физическим дискам, отнесенным к отдельным экземплярам. Таким образом, у каждого экземпляра имеется собственный процессор и собственный физический диск, так что друг другу они не мешают.

Индексы периодически обновляются локальными заданиями *cron*. Данные извлекаются из MySQL по сети, но файлы индексов создаются локально.

Оказалось, что несколько отдельных неформатированных (raw) дисков работают быстрее одного тома RAID. В этом случае можно точно кон-

тролировать, на какой неформатированный диск записывается конкретный файл. В случае RAID это не так – здесь контроллер решает, какой блок на какой диск отправить. Кроме того, неформатированные диски гарантируют, что ввод/вывод в разные группы индексов будет полностью распараллелен, тогда как при одновременном запуске нескольких поисков на RAID-массиве возможна сериализация операций. Мы выбрали уровень RAID 0, не обеспечивающий резервирования, поскольку отказы дисков нас не заботили, т.к. индексы на поисковых серверах легко перестроить. Можно было бы использовать несколько томов RAID 1 (с зеркалированием) и получить ту же пропускную способность, какую дают отдельные диски, но с повышенной надежностью.

На примере BoardReader интересно отметить еще одну вещь: как выполняется переход на новую версию Sphinx. Очевидно, что останавливать кластер целиком нельзя. Поэтому критичной оказывается обратная совместимость. К счастью, Sphinx ее обеспечивает – новые версии *searchd* могут читать старые индексы и взаимодействовать по сети со старыми клиентами. Отметим, что узлы первого уровня, которые служат для агрегирования результатов поиска, выглядят как обычные клиенты для узлов второго уровня, где собственно и производится поиск. Поэтому сначала обновляются узлы второго уровня, потом – первого уровня, а в последнюю очередь – приложение на веб-сервере.

Вот какие уроки мы извлекли из этого примера:

- Девиз разработчика Очень Большой Базы Данных: секционируй, секционируй, секционируй, распараллеливай.
- На крупных поисковых фермах организуйте демоны *searchd* в виде древовидных иерархий с несколькими уровнями.
- По возможности стройте оптимизированные индексы, каждый из которых содержит только часть данных.
- Устанавливайте соответствие между файлами и дисками явно, не полагаясь на RAID-контроллер.

Оптимизация выборки на сайте Sahibinden.com

У ведущего турецкого аукционного сайта Sahibinden.com было немало проблем с производительностью, в том числе из-за полнотекстового поиска. После развертывания Sphinx и профилирования выяснилось, что Sphinx способна обрабатывать многие часто выполняемые запросы с фильтрами быстрее, чем MySQL, несмотря на то, что по одному из участвующих столбцов был построен индекс. Кроме того, использование Sphinx для обычного (не полнотекстового) поиска позволило унифицировать код приложения так, что его стало проще писать и сопровождать.

MySQL не показывала высокой производительности, поскольку селективность индексов по отдельным столбцам была недостаточна для существенного сокращения пространства поиска. Более того, было прак-

тически невозможно создать и поддерживать все необходимые индексы, так как в условиях отбора участвовало слишком много столбцов. В таблице лотов было около 100 столбцов и теоретически каждый из них мог применяться для фильтрации или сортировки.

Интенсивные вставки и обновления «горячих» лотов выполнялись с черепашьей скоростью, так как приходилось обновлять слишком много индексов.

Поэтому система Sphinx стала естественным выбором для выполнения всех, а не только полнотекстовых запросов `SELECT` к таблице лотов.

Приведем размер базы данных и характеристики нагрузки:

- База данных содержит примерно 400 000 записей, а ее объем составляет 500 Мбайт.
- В день выполняется около 3 миллионов запросов.

Чтобы эмулировать обычные запросы `SELECT` с условиями `WHERE`, в полнотекстовый индекс Sphinx были добавлены специальные ключевые слова. Они имели вид `__CAT`*`N`*`___`, где вместо *N* подставлялся идентификатор соответствующей категории. Эта подстановка производилась на этапе индексирования путем включения функции `CONCAT()` в запрос к MySQL, поэтому изменять источник данных не пришлось.

Индексы нужно было перестраивать как можно чаще. Мы решили делать это раз в минуту. На полную перестройку индекса уходило 9–15 секунд на одном из многих процессоров, поэтому схема *главный + разностный*, которую мы обсуждали выше, оказалась излишней.

Как выяснилось, PHP API тратил достаточно много времени (7–9 мс/запрос) на разбор результирующего набора, в котором было много атрибутов. Обычно эти издержки можно не принимать во внимание, так как затраты на полнотекстовый поиск, особенно по большим наборам данных, безусловно доминируют. Но в данном случае нам было также необходимо выполнять неполнотекстовые запросы к небольшому набору данных. Чтобы решить эту проблему, мы разбили индексы на пары: «облегченный», который содержал 34 наиболее востребованных атрибута, и «полный», содержавший все 99 атрибутов.

Можно было бы поступить и по-другому: воспользоваться подсистемой SphinxSE или перенести в Sphinx только часть столбцов. Однако решение на базе двух индексов оказалось быстрее реализовать, а время играло решающую роль.

Из этого примера мы извлекли следующие уроки:

- Иногда полное сканирование средствами Sphinx работает быстрее, чем доступ по индексу MySQL.
- Для условий с высокой селективностью применяйте «фиктивные ключевые слова» вместо фильтрации по атрибутам, тогда механизм полнотекстового поиска сможет взять на себя большую часть работы.

- API сценарных языков может оказаться узким местом в некоторых экзотических, но встречающихся на практике случаях.

Оптимизация GROUP BY на сайте BoardReader.com

Для улучшения службы BoardReader потребовалось подсчитывать гиперссылки и строить по этим данным различные отчеты. Например, в одном из отчетов нужно было показать *N* верхних доменов второго уровня, которые сослались на BoardReader.com в течение прошлой недели. В другом отчете отображались *N* верхних доменов второго и третьего уровня, ссылающихся на данный сайт, например YouTube. Запросы, необходимые для построения этих отчетов, обладали следующими общими характеристиками:

- Они всегда группировали по домену.
- Они сортировали по количеству записей в группе или по количеству различных значений в группе.
- Они обрабатывали очень большой объем данных (миллионы записей), но результирующий набор, состоящий из лучших групп, мал.
- Были приемлемы приближенные результаты.

На этапе тестирования прототипа выяснилось, что MySQL тратит на выполнение таких запросов до 300 секунд. Теоретически путем секционирования данных, размещения их на разных серверах и ручного агрегирования результатов в приложении можно было довести время выполнения до 10 секунд. Но подобную архитектуру трудно построить; даже одна лишь реализация секционирования далеко не тривиальна.

Поскольку мы добились успеха в применении Sphinx для распределения поисковой нагрузки, было решено использовать Sphinx и в целях реализации приближенной группировки. Для этого потребовалось предварительно обработать данные перед индексированием, преобразовав все интересующие нас подстроки в отдельные «слова». Вот пример URL до и после предобработки:

```
source_url      = http://my.blogger.com/my/best-post.php
processed_url   = my$blogger$com, blogger$com,
                  my$blogger$com$my,
                  my$blogger$com$my$best,
                  my$blogger$com$my$best$post.php
```

Знаки доллара ($) просто заменяют все символы-разделители в URL. Необходимо это для того, чтобы поиск можно было производить по любой части URL, будь то доменное имя или путь. В результате такой предобработки все «интересные» подстроки выделяются в виде обособленных слов, чтобы поиск производился максимально быстро. Технически мы могли использовать для поиска целую фразу или применить индексирование по префиксу, но тогда индексы получились бы более громоздкими, а производительность снизилась.

Ссылки преобразуются во время индексирования с помощью специально написанной пользовательской функции (UDF). Для этой задачи мы также модифицировали Sphinx, чтобы она могла подсчитывать различные значения. В итоге мы смогли полностью перенести запросы на поисковый кластер, распределить их, и тем самым существенно снизить время задержки.

Приведем размер базы данных и характеристики нагрузки:

- В базе примерно 150–200 миллионов записей, после предобработки это выливается в 50–100 Гбайт данных.
- Нагрузка составляет приблизительно 60 000–100 000 запросов с группировкой в день.

Индексы для распределенных запросов `GROUP BY` были размещены на том же поисковом кластере из 6 машин с 24 процессорами, который был описан выше. Это лишь немного увеличило нагрузку по сравнению с той, что создается поисковыми запросами к базе данных объемом 1,5 Тбайт.

Sphinx позволила заменить точные, но медленные вычисления, выполняемые сервером MySQL на одном процессоре, приближенными, но быстрыми и распределенными. Присутствовали все факторы, приводящие к погрешностям: входные данные часто содержали слишком много строк, которые не помещаются в «буфер сортировки» (мы ограничили его размер 100K строками), использовалась агрегатная функция `COUNT(DISTINCT)` и собирались результирующие наборы, полученные от разных узлов. Несмотря на все это, итоги для первых 10–1000 групп (а именно столько нам было нужно для отчетов) оказались правильными на 99–100%.

Индексированные данные очень сильно отличались от тех, что использовались бы при обычном полнотекстовом поиске. Количество документов и ключевых слов огромно, хотя сами документы совсем маленькие. Их нумерация не последовательная, так как применяется специальное соглашение о присвоении номеров (сервер-источник, таблица-источник и первичный ключ), и результирующий номер не умещается в 32 бита. Огромное количество «ключевых слов» часто приводило к коллизиям при вычислении CRC32-свертки (в Sphinx используется алгоритм CRC32 для отображения самих ключевых слов на их внутренние идентификаторы). Из-за этого нам пришлось повсюду перейти к 64-разрядным идентификаторам.

В настоящий момент производительность удовлетворительная. Для самых сложных доменов запрос обычно завершается за 0,1–1,0 секунд.

Из этого примера мы извлекли следующие уроки:

- Для запросов с группировкой точность иногда можно принести в жертву скорости.

- Для очень больших наборов текстовых документов и даже для наборов умеренного размера, но специального вида могут понадобиться 64-разрядные идентификаторы.

Оптимизация запросов с JOIN к секционированным данным на сайте Grouply.com

Поддержка многозначных атрибутов в Sphinx – сравнительно новая функция, но пользователи уже нашли ей хитроумные применения. На сайте Grouply.com решение на основе Sphinx позволило осуществлять поиск в базе данных, насчитывающей многие миллионы тегированных сообщений. Для повышения масштабируемости эта база распределена между несколькими физическими серверами, поэтому иногда приходится опрашивать таблицы, находящиеся на разных машинах. Так как количество серверов, баз данных и таблиц велико, то выполнять произвольные запросы с соединениями невозможно.

Для хранения тегов сообщений на сайте Grouply.com применяются многозначные атрибуты Sphinx. Список тегов извлекается из кластера Sphinx с помощью PHP API. Это позволяет заменить несколько последовательных запросов `SELECT` к разным серверам. Чтобы также уменьшить количество SQL-запросов, некоторые данные, нужные только для отображения (например, короткий список пользователей, недавно прочитавших сообщение), также хранится в отдельном MЗА, и доступ к нему производится через Sphinx.

Два основных новшества, примененных на этом сайте, – это эксплуатация системы Sphinx для предварительного построения результатов соединения и использование ее распределенных возможностей для объединения данных, разбросанных по многим секциям (shards). С помощью одной лишь СУБД MySQL это было бы почти невозможно сделать. Для эффективного объединения нужно было стремиться к уменьшению количества серверов и таблиц, содержащих секции, но это вошло бы в противоречие с масштабируемостью и расширяемостью.

Из этого примера мы извлекли следующие уроки:

- Sphinx можно использовать для эффективного агрегирования глубоко секционированных данных.
- Многозначные атрибуты можно использовать для хранения и оптимизации предварительного построенных результатов операции JOIN.

Заключение

В этом приложении мы смогли лишь кратко рассмотреть систему полнотекстового поиска Sphinx. Мы сознательно опустили многие средства Sphinx, например индексирование HTML-документов, запросы с указанием диапазона для улучшенной поддержки таблиц типа MyISAM, поддержку морфологии и синонимии, индексирование по префиксам

и инфиксам и индексирование документов на восточных языках. Тем не менее, мы надеемся, что вы смогли получить представление о том, как с помощью Sphinx можно эффективно решать различные реальные задачи. Эта система не ограничивается полнотекстовым поиском, а может быть применена ко многим трудным задачам, которые традиционно решались с помощью SQL.

Sphinx не является ни панацеей, ни заменой MySQL. Однако во многих случаях (а их в современных веб-приложениях становится все больше), ее можно использовать как полезное дополнение к MySQL. Она не только позволяет разгрузить сервер базы данных, но и открывает новые возможности для созданий приложений.

Скачайте Sphinx с сайта *http://www.sphinxsearch.com* – и не забудьте поделиться своими идеями!

D

Отладка блокировок

В любой системе, где для управления совместным доступом к разделяемым ресурсам применяются блокировки, очень трудно отлаживать возникающие вследствие конкуренции за их получение ошибки. Возможно, вы пытаетесь добавить в таблицу новый столбец или просто запускаете запрос и внезапно обнаруживаете, что операция не выполняется, потому что кто-то другой заблокировал таблицу или необходимые вам строки. В этом приложении мы расскажем, что делать, столкнувшись с такими проблемами в MySQL. Зачастую нужно лишь понять, почему запрос заблокирован, но иногда хочется определить, кто его заблокировал, чтобы знать, какой процесс убивать. Мы покажем, как решить обе задачи.

Ожидание блокировки на уровне сервера

Блокировки могут происходить как на уровне сервера, так и на уровне подсистемы хранения[1]. Блокировки на уровне приложения тоже могут представлять проблему, но сейчас мы говорим только о СУБД MySQL.

Сам сервер MySQL применяет несколько типов блокировок. Узнать о том, что запрос ожидает блокировку на уровне сервера, можно из результатов команды `SHOW PROCESSLIST`. Помимо блокировок на уровне сервера любая подсистема, поддерживающая блокировку строк, например InnoDB, реализует собственный механизм блокировок, по крайней мере, так было на момент работы над этой книгой. В версии MySQL 5.0 и более ранних сервер ничего не знал о таких блокировках, и они были по большей части скрыты от пользователей и администраторов. Возможно, что в будущих реализациях на уровне сервера будет выдаваться более подробная информация о блокировках, например за счет таблиц, подключаемых к базе `INFORMATION_SCHEMA`.

[1] Если вы забыли о том, как разделяются обязанности между сервером и подсистемами хранения, см. рис. 1.1 в главе 1.

Перечислим виды блокировок, используемых самим сервером MySQL:

Табличные блокировки

Таблицы можно блокировать, устанавливая явные блокировки чтения и записи. Существует несколько разновидностей подобных блокировок, например локальные блокировки чтения. Прочитать о всех разновидностях можно в разделе руководства по MySQL, посвященном команде `LOCK TABLES`. Помимо явных блокировок запросы захватывают неявные на время своего выполнения.

Глобальные блокировки

Существует единственная глобальная блокировка, захватываемая командой `FLUSH TABLES WITH READ LOCK`.

Блокировки на имя

Блокировка на имя – это разновидность табличной блокировки, которую сервер захватывает, когда переименовывает или удаляет таблицу.

Пользовательские блокировки (user lock или string lock)

Можно захватить и освободить блокировку на произвольную строку на уровне сервера, воспользовавшись функцией `GET_LOCK()` и родственными ей.

В последующих разделах мы рассмотрим все эти блокировки более подробно.

Табличные блокировки

Табличные блокировки могут быть явными и неявными. Явная блокировка захватывается командой `LOCK TABLES`. Например, если в сеансе работы с программой *mysql* выполнить следующую команду, то будет захвачена явная блокировка на таблицу `sakila.film`:

```
mysql> LOCK TABLES sakila.film READ;
```

Если затем в другом сеансе выполнить показанную ниже команду, то запрос «повиснет» и не завершится:

```
mysql> LOCK TABLES sakila.film WRITE;
```

На первом соединении вы увидите ожидающий поток:

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 7
   User: baron
   Host: localhost
     db: NULL
Command: Query
   Time: 0
  State: NULL
   Info: SHOW PROCESSLIST
```

```
*************************** 2. row ***************************
     Id: 11
   User: baron
   Host: localhost
     db: NULL
Command: Query
   Time: 4
  State: Locked
   Info: LOCK TABLES sakila.film WRITE
2 rows in set (0.01 sec)
```

Обратите внимание, что поток 11 находится в состоянии `Locked`. В коде MySQL есть только одно место, где поток входит в это состояние: когда он пытается получить блокировку на таблицу, которая в данный момент заблокирована другим потоком. Таким образом, увидев подобную картину, можно сразу же сказать, что поток ждет блокировки на уровне сервера, а не какой-то подсистемы хранения.

Однако явные блокировки – не единственная причина остановки подобной операции. Как мы уже отмечали, сервер неявно блокирует таблицы на время выполнения запросов. Продемонстрировать это проще всего, запустив длительный запрос, для чего достаточно применить функцию `SLEEP()`:

```
mysql> SELECT SLEEP(30) FROM sakila.film LIMIT 1;
```

Если попытаться еще раз заблокировать таблицу `sakila.film`, пока работает этот запрос, то операция повиснет из-за неявной блокировки – точно так же, как это произошло бы в случае явной. В списке процессов вы будете наблюдать такую же ситуацию, как и выше:

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 7
   User: baron
   Host: localhost
     db: NULL
Command: Query
   Time: 12
  State: Sending data
   Info: SELECT SLEEP(30) FROM sakila.film LIMIT 1
*************************** 2. row ***************************
     Id: 11
   User: baron
   Host: localhost
     db: NULL
Command: Query
   Time: 9
  State: Locked
   Info: LOCK TABLES sakila.film WRITE
```

В этом примере неявно захваченная при выполнении команды SELECT блокировка чтения препятствует получению явной блокировки на запись командой LOCK TABLES. Неявные блокировки могут блокировать и друг друга.

Может возникнуть вопрос, чем неявные блокировки отличаются от явных. На внутреннем уровне их структура одинакова и управляются они одним и тем же кодом. А на внешнем вы можете управлять явными блокировками с помощью команд LOCK TABLES и UNLOCK TABLES.

Однако когда речь заходит о подсистемах хранения, отличных от MyISAM, между этими блокировками обнаруживается одно очень важное различие. Блокировка, созданная явно, делает в точности то, что вы попросили, тогда как неявные блокировки скрыты и ведут себя «магическим» образом. Сервер захватывает и освобождает неявные блокировки автоматически по мере необходимости и сообщает о них подсистеме хранения. Подсистема хранения «конвертирует» эти блокировки, исходя из своих потребностей. Например, в InnoDB существуют правила, диктующие, какую табличную блокировку следует создавать для данной табличной блокировки на уровне сервера. Поэтому трудно понять, какие же блокировки InnoDB в реальности создает «за кулисами».

В версиях MySQL 5.0 и 5.1 СУБД управляет табличными блокировками на уровне сервера так, что взаимоблокировок не возникает, создавая и освобождая их все сразу и в четко определенном порядке. В MySQL 6.0 можно захватывать дополнительные блокировки, не освободив уже имеющиеся, поэтому возникает опасность взаимоблокировок на уровне таблиц. Однако в данный момент этот код еще не полностью завершен, поэтому окончательное его поведение неизвестно.

Определение владельца блокировки

Если вы видите много процессов в состоянии Locked, то, возможно, дело в том, что MyISAM или похожая подсистема хранения применяется в условиях высокой конкурентной рабочей нагрузки. Это может помешать ручному выполнению той или иной операции, например построению нового индекса. Если в очередь на получение блокировки для таблицы типа MyISAM стоит запрос UPDATE, то не могут выполняться даже SELECT-запросы. Дополнительную информацию об очередях на получение блокировки и о приоритетах можно почерпнуть в руководстве по MySQL.

Иногда становится очевидно, что некоторое соединение удерживает блокировку на таблицу слишком долго, и его просто нужно принудительно разорвать (или воззвать к совести пользователя и попросить, чтобы он не мешал работать всем остальным!). Но как определить, какое соединение владеет блокировкой?

В настоящее время не существует SQL-команды, показывающей, какой поток удерживает табличные блокировки, которые не дают выполнить

ваш запрос. Команда SHOW PROCESSLIST демонстрирует только процессы, которые ждут блокировки, но не процессы, которые их захватили. К счастью, существует команда *debug* (запустить ее через SQL нельзя), которая выводит информацию о блокировках в журнал ошибок сервера. Чтобы запустить ее, воспользуйтесь утилитой *mysqladmin*:

```
$ mysqladmin debug
```

Она выводит достаточно много отладочной информации, но ближе к концу находится то, что нас интересует. Эта распечатка получена после того, как мы заблокировали таблицу в одном соединении и затем попробовали заблокировать ее же в другом:

```
Thread database.table_name Locked/Waiting   Lock_type

7    sakila.film    Locked - read     Read lock  without concurrent inserts
8    sakila.film    Waiting - write   Highest priority write lock
```

Как видите, поток 8 ожидает блокировку, удерживаемую потоком 7.

Команда *debug* утилиты *mysqladmin* выводит еще больше информации, если при компиляции MySQL был активирован отладочный режим, но сведения о блокировках и о некоторых других вещах печатаются в любом случае.

Глобальная блокировка чтения

Сервер MySQL реализует также глобальную блокировку чтения. Получить ее можно следующим образом:

```
mysql> FLUSH TABLES WITH READ LOCK;
```

Если теперь попытаться заблокировать любую таблицу в другом сеансе, то этот сеанс повиснет:

```
mysql> LOCK TABLES sakila.film WRITE;
```

Как узнать, ждет ли запрос глобальную блокировку чтения или табличную блокировку? Следует взглянуть на результат команды SHOW PROCESSLIST:

```
mysql> SHOW PROCESSLIST\G
...
*************************** 2. row ***************************
     Id: 22
   User: baron
   Host: localhost
     db: NULL
Command: Query
   Time: 9
  State: Waiting for release of readlock
   Info: LOCK TABLES sakila.film WRITE
```

Обратите внимание на состояние процесса: `Waiting for release of readlock` (Ожидает освобождения блокировки чтения). Это как раз и означает, что запрос ждет глобальную блокировку чтения, а не табличную блокировку.

MySQL не позволяет узнать, кто удерживает глобальную блокировку чтения.

Блокировки на имя

Блокировка на имя – это разновидность табличной блокировки, которую сервер захватывает, когда собирается переименовать или удалить таблицу. Такая блокировка конфликтует с обычной табличной блокировкой, все равно, явной или неявной. Например, если мы, как и раньше, выполним в одном сеансе команду `LOCK TABLES`, а в другом попытаемся эту таблицу переименовать, то запрос повиснет, но на этот раз не в состоянии `Locked`:

```
mysql> RENAME TABLE sakila.film2 TO sakila.film;
```

Посмотрев на заблокированный запрос в списке процессов, мы увидим, что он находится в состоянии `Waiting for table` (Ожидает таблицу):

```
mysql> SHOW PROCESSLIST\G
...
*************************** 2. row ***************************
     Id: 27
   User: baron
   Host: localhost
     db: NULL
Command: Query
   Time: 3
  State: Waiting for table
   Info: rename table sakila.film to sakila.film 2
```

Результат блокировки на имя можно увидеть также командой `SHOW OPEN TABLES`:

```
mysql> SHOW OPEN TABLES;
+----------+-----------+--------+-------------+
| Database | Table     | In_use | Name_locked |
+----------+-----------+--------+-------------+
| sakila   | film_text |      3 |           0 |
| sakila   | film      |      2 |           1 |
| sakila   | film2     |      1 |           1 |
+----------+-----------+--------+-------------+
3 rows in set (0.00 sec)
```

Отметим, что заблокированы оба имени (старое и новое). Имя `sakila.film_text` заблокировано, потому что к таблице `sakila.film` присоединен ссылающийся на нее триггер; таким образом, мы видим, что блокиров-

ки могут проявляться там, где их и не ожидаешь. При обращении к таблице sakila.film триггер вызывает неявное обращение к sakila.film_text, а, стало быть, и захват неявной блокировки на нее. Вообще-то при переименовании триггер не должен бы срабатывать, поэтому, строго говоря, блокировка не нужна, но так уж сложилось: уровень детализации блокировок в MySQL не всегда такой, каким хотелось бы его видеть.

MySQL не позволяет узнать, кто удерживает блокировку на имя, но обычно это несущественно, так как такие блокировки очень быстротечны. Конфликт скорее может быть вызван тем, что саму блокировку на имя не удается получить из-за того, что кто-то захватил обычную табличную блокировку, но это как раз можно узнать с помощью *mysqladmin debug*, как показано выше.

Пользовательские блокировки

И последняя разновидность блокировок, реализованная на уровне сервера, – пользовательские блокировки, по существу являющиеся ни чем иным, как именованными мьютексами. Вы задаете строку, на которую нужно получить блокировку, и время ожидания в секундах, по истечении которого произойдет тайм-аут:

```
mysql> SELECT GET_LOCK('my lock', 100);
+--------------------------+
| GET_LOCK('my lock', 100) |
+--------------------------+
|                        1 |
+--------------------------+
1 row in set (0.00 sec)
```

Эта попытка удалась с первого раза, так что теперь поток владеет данным именованным мьютексом. Если другой поток попытается получить блокировку на ту же самую строку, то он подвиснет до истечения тайм-аута. В списке процессов для него показывается специальное состояние:

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 22
   User: baron
   Host: localhost
     db: NULL
Command: Query
   Time: 9
  State: User lock
   Info: SELECT GET_LOCK('my lock', 100)
```

Состояние User lock характерно только для таких блокировок. MySQL не позволяет узнать, кто владеет пользовательской блокировкой.

Ожидание блокировки на уровне подсистемы хранения

Блокировки на уровне сервера отлаживать проще, чем блокировки внутри подсистемы хранения. Механизм блокирования зависит от самой подсистемы, и некоторые из них вообще не предоставляют средств для получения информации о своих блокировках. В этом приложении мы будем говорить в основном об InnoDB, так как на сегодняшний день это самая популярная подсистема с собственным механизмом блокирования.

Ожидание блокировки в InnoDB

InnoDB раскрывает информацию о блокировках с помощью команды `SHOW INNODB STATUS`. Если транзакция ожидает некоторую блокировку, то последняя будет присутствовать в секции `TRANSACTIONS`. Например, выполнив следующие команды в одном сеансе, вы захватите блокировку записи на первую строку таблицы:

```
mysql> SET AUTOCOMMIT=0;
mysql> BEGIN;
mysql> SELECT film_id FROM sakila.film LIMIT 1 FOR UPDATE;
```

Если теперь выполнить те же команды в другом сеансе, то запрос повиснет в ожидании блокировки, захваченной в первом. `SHOW INNODB STATUS` продемонстрирует, что произошло (для простоты мы оставили только часть распечатки):

```
1 LOCK WAIT 2 lock struct(s), heap size 1216
2 MySQL thread id 8, query id 89 localhost baron Sending data
3 SELECT film_id FROM sakila.film LIMIT 1 FOR UPDATE
4 ------- TRX HAS BEEN WAITING 9 SEC FOR THIS LOCK TO BE GRANTED:
5 RECORD LOCKS space id 0 page no 194 n bits 1072 index `idx_fk_language_id`
  of table `sakila/film` trx id 0 61714 lock_mode X waiting
```

В последней строке показано, что запрос ожидает монопольную (`lock_mode X`) блокировку на страницу 194 индекса `idx_fk_language_id` данной таблицы. В конечном итоге тайм-аут ожидания истечет, и запрос вернет такую ошибку:

```
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

К сожалению, не зная, кто удерживает блокировку, трудно понять, какая транзакция стала источником проблемы. Иногда можно сделать обоснованное предположение, посмотрев, какие транзакции открыты очень давно. Можно также активировать монитор блокировок InnoDB, который показывает до 10 блокировок, удерживаемых каждой транзак-

цией. Чтобы это сделать, нужно создать таблицу типа InnoDB со специальным именем:[1]

```
mysql> CREATE TABLE innodb_lock_monitor(a int) ENGINE=INNODB;
```

После выполнения этого запроса InnoDB начинает периодически (с разной частотой, но обычно несколько раз в минуту) печатать в стандартный поток вывода ту же информацию, что команда SHOW INNODB STATUS, но в слегка расширенном формате. В большинстве систем стандартный поток вывода перенаправлен в журнал ошибок сервера, поэтому, изучив его, вы сможете узнать, какие транзакции удерживают блокировки. Чтобы остановить монитор, удалите эту таблицу.

Вот пример информации, печатаемой монитором:

```
 1 ---TRANSACTION 0 61717, ACTIVE 3 sec, process no 5102,
   OS thread id 1141152080
 2 3 lock struct(s), heap size 1216
 3 MySQL thread id 11, query id 108 localhost baron
 4 show innodb status
 5 TABLE LOCK table `sakila/film` trx id 0 61717 lock mode IX
 6 RECORD LOCKS space id 0 page no 194 n bits 1072 index
   `idx_fk_language_ id` of    table `sakila/film` trx id 0 61717 lock_mode X
 7 Record lock, heap no 2 PHYSICAL RECORD: n_fields 2; compact format;
   info bits 0
 8  ... опущено ...
 9
10 RECORD LOCKS space id 0 page no 231 n bits 168 index `PRIMARY` of table
   `sakila/film` trx id 0 61717 lock_mode X locks rec but not gap
11 Record lock, heap no 2 PHYSICAL RECORD: n_fields 15; compact format;
   info bits 0
12  ... опущено ...
```

Обратите внимание на строку 3, где показан идентификатор потока MySQL, – то же самое значение, что в столбце Id в списке процессов. В строке 5 видно, что транзакция владеет неявной монопольной блокировкой (IX) на таблицу. В строках 6–7 демонстрируется блокировка на индекс. Информацию в строке 8 мы опустили, потому что это довольно длинный дамп заблокированной записи. В строках 10–11 показана блокировка на первичный ключ (фраза FOR UPDATE приводит к блокировке не только индекса, но и самой строки).

Хотя это и не документировано, при активированном мониторе блокировок дополнительная информация появляется и в распечатке, формируемой командой SHOW INNODB STATUS, поэтому необязательно заглядывать в журнал ошибок сервера.

---

[1] InnoDB понимает несколько таких «волшебных» имен таблиц. Сейчас предпочитают динамически изменяемые серверные переменные, но InnoDB существует уже давно, поэтому сохранила некоторые старые «привычки».

Более удобный вывод информации о блокировках

Монитор блокировок – не самое оптимальное решение по нескольким причинам. Основная проблема – чрезмерная подробность выдаваемой информации, которая включает дамп заблокированных записей в шестнадцатеричном и ASCII виде. Все это быстро заполняет журнал ошибок и легко может переполнить буфер вывода `SHOW INNODB STATUS`, размер которого фиксирован. А тогда вы не увидите именно то, что ищете, поскольку эти сведения находятся в последних секциях (см. раздел «Секция LATEST DETECTED DEADLOCK» главы 13 на стр. 696). Кроме того, в InnoDB «зашито» ограничение на количество печатаемых блокировок для одной транзакции (10), поэтому сведений об интересующей вас блокировке вы можете как раз и не увидеть. Но даже если искомое присутствует, найти его отнюдь не всегда просто (попробуйте активировать монитор на загруженном сервере и убедитесь сами!).

Чтобы сделать формат вывода информации о блокировках более удобным, есть два пути. Во-первых, один из авторов этой книги написал заплату для InnoDB и сервера MySQL, которая удаляет подробные дампы записей из выводимой информации, включает информацию о блокировках в команду `SHOW INNODB STATUS` по умолчанию (поэтому монитор вообще не нужно активировать) и добавляет динамически изменяемые серверные переменные для управления уровнем подробности и количеством блокировок, печатаемых для одной транзакции. Скачать эту заплату для версии MySQL 5.0 можно по адресу *http://lists.mysql.com/internals/35174*.

Второй способ – воспользоваться программой *innotop* для разбора и форматирования вывода. В режиме `Lock` она показывает блокировки, сгруппированные по соединению и по таблице, так что можно без труда понять, какие транзакции удерживают блокировку на данную таблицу. Это не безупречный метод поиска «виновной» транзакции, так как для обнаружения заблокированной записи необходимо исследовать дампы записей. Но все же он гораздо лучше имеющихся альтернатив и достаточно хорош в большинстве случаев.

Разработчики InnoDB говорили нам, что работают над экспортом информации InnoDB в таблицы `INFORMATION_SCHEMA`, но пока этот код еще не опубликован. В будущем это, вероятно, станет наиболее предпочтительным методом получения информации о блокировках.

Ожидание блокировки в Falcon

Транзакционная подсистема хранения Falcon, которая на момент написания этой книги входила в состав альфа-версии MySQL 6.0, экспортирует информацию о транзакциях в таблицу `INFORMATION_SCHEMA`. Найти причину ожидания блокировки позволяет следующая SQL-команда:

```
mysql> SELECT a.THREAD_ID AS blocker, a.STATEMENT AS blocking_query,
    ->    b.THREAD_ID AS blocked, b.STATEMENT AS blocked_query
```

```
    -> FROM INFORMATION_SCHEMA.FALCON_TRANSACTIONS AS a
    ->    INNER JOIN INFORMATION_SCHEMA.FALCON_TRANSACTIONS AS b ON
    ->    a.ID = b.WAITING_FOR
    -> WHERE b.WAITING_FOR > 0;
+---------+---------------+---------+-----------------------------+
| blocker | blocking_query | blocked | blocked_query               |
+---------+---------------+---------+-----------------------------+
|       4 |               |       5 | SELECT * FROM tbl FOR UPDATE |
+---------+---------------+---------+-----------------------------+
```

Такого рода диагностическая информация должна существенно облегчить жизнь администраторам баз данных MySQL в будущем!

Алфавитный указатель

S

T

U

Г

Д

Л

М

Ф

Х

Ш

Э