1 История и архитектура MySQL

Архитектура MySQL очень отличается от архитектур иных серверов баз данных, что делает эту СУБД полезной для одних целей, но одновременно неудачным выбором для других. MySQL неидеальна, но достаточно гибка для того, чтобы хорошо работать в очень требовательных средах, например в веб-приложениях. В то же время MySQL позволяет применять встраиваемые приложения, хранилища данных, индексирование содержимого, программное обеспечение для доставки, высоконадежные системы с резервированием, обработку транзакций в реальном времени (OLTP) и многое другое.

Для того чтобы максимально эффективно использовать MySQL, нужно разобраться в ее устройстве. Гибкость системы проявляется во многом. Например, вы можете настроить ее для работы на различном оборудовании и поддержки разных типов данных. Однако самой необычной и важной особенностью MySQL является такая архитектура подсистемы хранения, в которой обработка запросов и другие серверные задачи отделены от хранения и извлечения данных. Подобное разделение задач позволяет выбирать способ хранения данных, а также настраивать производительность, ключевые характеристики и др.

В текущей главе сделан краткий обзор архитектуры сервера MySQL, рассматриваются основные различия между подсистемами хранения и говорится о том, почему эти различия важны. В конце главы мы рассмотрим исторический контекст и перечислим эталонные тесты производительности (бенчмарки). Попытаемся объяснить MySQL на примерах и максимально упрощая детали. Этот обзор будет полезен как для новичков в работе с сервером баз данных, так и для читателей, которые являются экспертами в работе с подобными системами.

Логическая архитектура MySQL

Чтобы хорошо понимать работу сервера, нужно иметь представление о взаимодействии его компонентов. На рис. 1.1 представлен логический вид архитектуры MySQL.

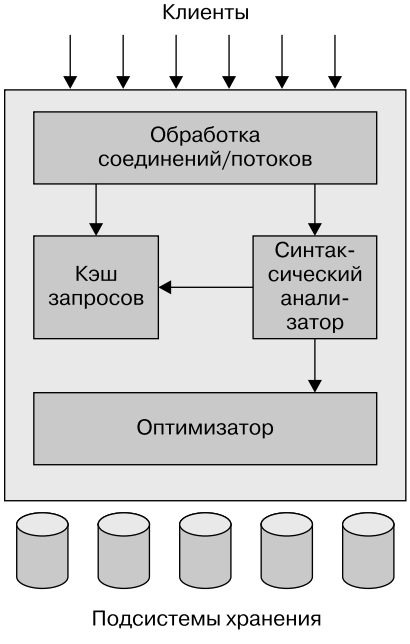

Рис. 1.1. Логический вид архитектуры сервера MySQL

На верхнем уровне располагаются службы, не являющиеся уникальными компонентами MySQL. Они необходимы большинству сетевых клиент-серверных инструментов или серверов: для обслуживания соединений, аутентификации, обеспечения безопасности и т. п.

Второй уровень намного интереснее. Здесь находится бóльшая часть «мозгов» MySQL: код для обработки, анализа, оптимизации и кэширования запросов, а также все встроенные функции (например, функции даты/времени, математические и функции шифрования). Здесь также расположены все инструменты, используемые в подсистемах хранения, например хранимые процедуры, триггеры и представления.

Третий уровень содержит подсистемы хранения данных. Они отвечают за хранение всех данных в MySQL и их извлечение. Подобно различным файловым системам, доступным для GNU/Linux, каждая подсистема хранения данных имеет как сильные, так и слабые стороны. Сервер взаимодействует с ними через API подсистемы хранения данных. Этот интерфейс скрывает различия между такими подсистемами и делает их практически прозрачными на уровне запросов. API содержит пару десятков низкоуровневых функций, выполняющих операции типа «начать транзакцию» или «извлечь строку с таким первичным ключом». Подсистемы хранения не анализируют запросы SQL[1] и не взаимодействуют друг с другом, они просто отвечают на исходящие от сервера запросы.

[1] Единственным исключением является InnoDB, где анализируются определения внешнего ключа, поскольку сам сервер MySQL не делает этого.

Управление соединениями и их безопасность

Для каждого клиентского соединения внутри серверного процесса выделяется отдельный поток. Запросы соединения выполняются только внутри этого потока, который, в свою очередь, выполняется одним ядром или процессором. Сервер кэширует потоки, поэтому их не нужно создавать или уничтожать для каждого нового соединения[1].

Когда клиенты (приложения) подключаются к серверу MySQL, он должен их аутентифицировать. Аутентификация выполняется на основе имени пользователя, адреса хоста, с которого происходит соединение, и пароля. При соединении по протоколу Secure Sockets Layer (SSL) можно использовать сертификаты X.509. После того как клиент подключился, сервер проверяет наличие необходимых привилегий для каждого запроса (например, может ли клиент использовать команду `SELECT` применительно к таблице `Country` базы данных `world`).

Оптимизация и выполнение

MySQL анализирует запросы для создания внутренней структуры (дерева разбора), а затем выполняет оптимизации. Это могут быть переписывание запроса, определение порядка чтения таблиц, выбор используемых индексов и т. п. Через специальные ключевые слова в запросе вы можете передать оптимизатору подсказки и тем самым повлиять на процесс принятия решения. Или обратиться к серверу за объяснением различных аспектов оптимизации. Это позволит вам понять, какие решения принимает сервер, и даст ориентир для изменения запросов, схем и настроек, чтобы добиться максимальной эффективности работы. Оптимизатор мы детально обсудим в главе 6.

Оптимизатору неважно, в какой подсистеме хранения данных находится конкретная таблица, но подсистема хранения данных влияет на то, как сервер оптимизирует запрос. Оптимизатор опрашивает подсистему хранения данных о некоторых ее возможностях, затратах на выполнение определенных операций и статистике по содержащимся в таблицах данным. Например, отдельные подсистемы хранения поддерживают типы индексов, которые могут быть полезны для выполнения определенных запросов. Больше информации об индексировании и схемах оптимизации вы сможете найти в главах 4 и 5.

Прежде чем анализировать запрос, сервер обращается к кэшу запросов, в котором могут храниться только команды `SELECT` вместе с наборами результатов. Если поступает запрос, идентичный уже имеющемуся в кэше, серверу не нужно выполнять анализ, оптимизацию или сам запрос — он может просто отправить в ответ сохраненный набор результатов. Больше об этом можно узнать, прочитав главу 7.

---

[1] MySQL 5.5 и более новые версии поддерживают API, который может принимать плагины для организации пула потоков, поэтому небольшой пул потоков может обслуживать множество соединений.

Управление конкурентным доступом

Задача управления конкурентным доступом возникает в тот момент, когда нескольким запросам необходимо одновременно изменить данные. В рамках текущей главы оговорим, что MySQL должна решать эту задачу на двух уровнях: сервера и подсистемы хранения данных. Управление конкурентным доступом — это обширная тема, которой посвящено множество теоретических исследований. Мы же просто представим обзор того, что MySQL делает с конкурентными запросами на чтение и запись, чтобы вы смогли получить общее представление об этой теме, а это, в свою очередь, позволит разобраться в материале главы.

В качестве примера будем использовать ящик электронной почты в системе UNIX. Классический файл формата mbox очень прост. Все сообщения в почтовом ящике mbox расположены одно за другим, так что читать и анализировать почтовые сообщения очень просто. Это также существенно упрощает доставку почты: достаточно добавить новое сообщение в конец файла.

Но что происходит, когда два процесса пытаются одновременно поместить сообщения в почтовый ящик? Очевидно, что чередование строк этих сообщений приведет к повреждению файла. Чтобы предотвратить это, правильно работающие почтовые системы используют блокировку. Если клиент пытается отправить новое сообщение в тот момент, когда почтовый ящик заблокирован, ему придется подождать, пока он не сможет сам использовать блокировку, чтобы отправить сообщение.

Эта схема довольно хорошо работает, но не поддерживает конкурентный доступ. Поскольку в любой момент времени только один процесс может изменять содержимое почтового ящика, такой подход создает проблемы при работе с большими почтовыми ящиками.

Блокировки чтения/записи

Чтение из почтового ящика не вызывает таких проблем. Ничего страшного, если несколько клиентов одновременно считывают информацию из одного и того же почтового ящика. Раз они не вносят изменений, ничего плохого случиться не должно. Но что произойдет, если кто-нибудь попытается удалить сообщение № 25 в тот момент, когда программы читают письма из почтового ящика? Возможны различные варианты развития ситуации, но программа чтения может получить почтовый ящик в поврежденном или неструктурированном виде. Поэтому для обеспечения безопасности даже чтение информации из почтового ящика требует определенных предосторожностей.

Если представить, что почтовый ящик — это таблица базы данных, а каждое почтовое сообщение — строка, легко увидеть, что и в этом контексте актуальна та же проблема. Во многих смыслах почтовый ящик является простой таблицей базы данных. Модификация строк в такой базе очень похожа на удаление или изменение содержимого сообщений в файле почтового ящика.

У классической задачи управления конкурентным доступом довольно простое решение. В системах, которые имеют дело с конкурентным доступом для чтения/записи, чаще всего реализуется система блокирования, содержащая два типа блокировок. Обычно их называют *разделяемыми блокировками* и *монопольными блокировками*, или *блокировками чтения* и *блокировками записи*.

Не вдаваясь в подробности технологии блокирования, данную концепцию можно описать следующим образом. Блокировки чтения ресурса являются разделяемыми или взаимно неблокирующими: множество клиентов могут производить считывание из ресурса в одно и то же время, не мешая друг другу. Блокировки записи, напротив, являются эксклюзивными. Другими словами, они исключают возможность установки блокировки чтения и других блокировок записи, поскольку единственной безопасной политикой является наличие только одного клиента, выполняющего запись в данный момент времени, и предотвращение во время этого всех операций чтения.

В базах данных блокировки происходят постоянно: MySQL запрещает одному клиенту считывать данные, когда другой клиент их изменяет. Управление блокировками осуществляется внутри СУБД в соответствии с принципами, которые достаточно прозрачны для клиентов.

Детальность блокировок

Одним из способов улучшения конкурентного доступа к разделяемому ресурсу является увеличение избирательности блокировок. Вместо того чтобы блокировать весь ресурс, можно заблокировать только ту его часть, которая содержит изменяемые данные. Еще лучше заблокировать лишь тот фрагмент данных, который будет изменен. Минимизация объема данных, которые вы блокируете в каждый момент времени, позволяет одновременно выполнять несколько изменений одного и того же ресурса, если эти операции не конфликтуют друг с другом.

Определенная проблема возникает из-за того, что блокировки потребляют ресурсы. Каждая операция блокирования — получение возможности блокировки, проверка того, можно ли применить блокировку, снятие блокировки и т. п. — влечет за собой издержки. Если система тратит слишком много времени на управление блокировками вместо того, чтобы расходовать его на сохранение и извлечение данных, то это может повлиять на производительность.

Стратегия блокирования является компромиссом между неизбежностью издержек на реализацию блокировок и безопасностью данных, причем подобный компромисс влияет на производительность. Большинство коммерческих серверов баз данных не предоставляют особого выбора: вы получаете возможность блокировки таблиц на уровне строки, при этом нередко в сочетании с множеством сложных способов обеспечить хорошую производительность при большом количестве блокировок.

MySQL также предоставляет выбор. Подсистемы хранения данных MySQL могут реализовывать собственные стратегии блокировки и уровни детализации блокировок. Управление блокировками является очень важным решением при проектировании подсистем хранения данных. Установка детализации на определенном уровне в ряде случаев может улучшить производительность, но сделать эту подсистему менее подходящей для других целей. Поскольку MySQL предлагает несколько подсистем хранения данных, нет необходимости принимать единственное решение на все случаи жизни. Рассмотрим две наиболее важные стратегии блокировок.

Табличные блокировки

Табличная блокировка является базовой стратегией блокировки в MySQL. Кроме того, у нее самые низкие издержки. Табличная блокировка аналогична рассмотренным ранее блокировкам почтового ящика — блокируется вся таблица. Когда клиент хочет сделать запись в таблицу (вставку, удаление, обновление и т. п.), он получает блокировку на запись для всей таблицы. Это предотвращает все остальные операции чтения и записи. Когда никто не выполняет запись, любой клиент может получить блокировку на чтение, которая не конфликтует с другими подобными блокировками.

У табличных блокировок есть вариации для обеспечения высокой производительности в различных ситуациях. Например, табличные блокировки `READ LOCAL` разрешают выполнять некоторые типы параллельных операций записи. Кроме того, блокировки записи имеют более высокий приоритет, чем блокировки чтения. Поэтому запрос блокировки записи будет помещен в очередь перед уже находящимися там запросами блокировки чтения (блокировки записи могут отодвигать в очереди блокировки чтения, а блокировки чтения не могут отодвигать блокировки записи).

Подсистемы хранения также могут управлять собственными блокировками.

MySQL использует множество блокировок, эффективных на уровне таблиц, для различных целей. Например, для таких операторов, как `ALTER TABLE`, сервер применяет табличную блокировку вне зависимости от подсистемы хранения данных.

Построчные блокировки

Построчные блокировки обеспечивают лучшее управление конкурентным доступом (и влекут максимальные издержки). Блокировка на уровне строк доступна, в частности, в подсистемах хранения InnoDB и XtraDB. Построчные блокировки реализуются подсистемами хранения данных, а не сервером (если нужно, еще раз взгляните на рис. 1.1). Сервер ничего не знает о блокировках, реализованных подсистемой хранения данных, и, как вы увидите далее, все подсистемы хранения данных реализуют блокировки по-своему.

Транзакции

До тех пор пока не познакомитесь с *транзакциями*, вы не сможете изучать более сложные функции СУБД.

Транзакция представляет собой группу запросов SQL, обрабатываемых *атомарно*, то есть как единое целое. Если подсистема базы данных может выполнить всю группу запросов, она делает это, но если какой-либо запрос не может быть выполнен в результате сбоя или по иной причине, ни один запрос группы не будет выполнен. Все или ничего.

Немногое в этом разделе характерно именно для MySQL. Если вы уже знакомы с транзакциями ACID, можете спокойно перейти к подразделу «Транзакции в MySQL».

Банковское приложение является классическим примером, демонстрирующим необходимость транзакций. Представьте банковскую базу данных с двумя таблицами: `checking` (текущие счета) и `savings` (сберегательные счета). Чтобы перевести 200 долларов с текущего счета Джейн на ее сберегательный счет, вам нужно сделать по меньшей мере три шага.

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

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

Вы начинаете транзакцию командой `START TRANSACTION`, а затем либо сохраняете изменения командой `COMMIT`, либо отменяете их командой `ROLLBACK`. Код SQL для 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? Кто знает... Клиент, вероятно, потеряет 200 долларов. А если другой процесс вклинится между выполнением строк 3 и 4 и снимет весь остаток с текущего счета? Банк предоставит клиенту кредит 200 долларов, даже не зная об этом.

Транзакций недостаточно, пока система не прошла *тест ACID*. Аббревиатура ACID расшифровывается как atomicity, consistency, isolation и durability (атомарность, согласованность, изолированность и долговечность). Это тесно связанные критерии,

которым должна соответствовать правильно функционирующая система обработки транзакций.

- **Атомарность.** Транзакция должна функционировать как единая неделимая рабочая единица таким образом, чтобы вся она была либо выполнена, либо отменена. Для атомарных транзакций не существует такого понятия, как частичное выполнение: все или ничего.
- **Согласованность.** База данных всегда должна переходить из одного согласованного состояния в другое. В нашем примере согласованность гарантирует, что сбой между строками 3 и 4 не приведет к исчезновению с текущего счета 200 долларов. Поскольку транзакция не будет подтверждена, ни одно из изменений не отразится в базе данных.
- **Изолированность.** Результаты транзакции обычно невидимы другим транзакциям, пока она не подтверждена. В нашем примере это гарантирует, что, если программа суммирования остатков на банковских счетах будет запущена после третьей строки перед четвертой, она по-прежнему увидит 200 долларов на текущем счете. Когда будем рассматривать уровни изолированности, вы поймете, почему здесь сказано «обычно невидимы».
- **Долговечность.** После подтверждения внесенные в ходе транзакции изменения становятся постоянными. Это значит, что они должны быть записаны так, чтобы данные не потерялись при сбое системы. Долговечность, однако, является несколько расплывчатой концепцией, поскольку у нее довольно много уровней. Некоторые стратегии обеспечения долговечности дают более высокие гарантии безопасности, чем другие, и ни одна из них не является надежной на 100 % (если база данных долговечна сама по себе, то каким образом резервное копирование повышает долговечность?). В последующих главах мы еще обсудим, что же на самом деле в MySQL означает долговечность.

Транзакции ACID гарантируют, что банк не потеряет ваши деньги. Вообще очень сложно, а то и невозможно сделать это с помощью логики приложения. Сервер базы данных, поддерживающий ACID, должен выполнить множество сложных операций, о которых вы, возможно, даже не подозреваете, чтобы обеспечить гарантии ACID.

Как и в случае увеличения детализации блокировок, оборотной стороной усиленной безопасности является увеличение объема работы сервера базы. Сервер базы данных с транзакциями ACID также требует бóльших мощности процессора, объема памяти и дискового пространства, чем сервер без них. Как мы уже отмечали, это тот самый случай, когда архитектура подсистем хранения данных MySQL является вашим союзником. Вы сами можете решить, требует ли приложение использования транзакций. Если они не нужны, вы можете добиться большей производительности, выбрав для некоторых типов запросов нетранзакционную подсистему хранения данных. С помощью команды `LOCK TABLES` можно установить нужный уровень защиты без использования транзакций. Все в ваших руках.

Уровни изолированности

Изолированность — более сложное понятие, чем кажется на первый взгляд. Стандарт SQL определяет четыре уровня изолированности с конкретными правилами, устанавливающими, какие изменения видны внутри и за пределами транзакции, а какие — нет. Более низкие уровни изолированности обычно допускают бóльшую степень конкурентного доступа и влекут за собой меньшие издержки.

Все подсистемы хранения данных реализуют уровни изолированности немного по-разному, и они не всегда будут соответствовать вашим ожиданиям, если вы привыкли к другой СУБД (здесь не будем вдаваться в подробности). Следует ознакомиться с руководствами по тем подсистемам хранения данных, которые вы решите использовать.

Вкратце рассмотрим четыре уровня изолированности.

- `READ UNCOMMITTED`. На этом уровне изолированности транзакции могут видеть результаты незавершенных транзакций. Вы можете столкнуться с множеством проблем, если не знаете абсолютно точно, что делаете. Используйте этот уровень, только если у вас есть на то веские причины. На практике этот уровень применяется редко, поскольку в этом случае производительность лишь немного выше, чем на других уровнях, имеющих множество преимуществ. Чтение незавершенных данных называют еще *черновым*, или *«грязным» чтением* (dirty read).
- `READ COMMITTED`. Это уровень изолированности, который устанавливается по умолчанию в большинстве СУБД (но не в MySQL!). Он соответствует приведенному ранее простому определению изолированности: транзакция увидит только те изменения, которые к моменту ее начала подтверждены другими транзакциями, а произведенные ею изменения останутся невидимыми для других транзакций, пока текущая не будет подтверждена. На этом уровне возможно так называемое *неповторяющееся чтение* (nonrepeatable read). Это означает, что вы можете выполнить одну и ту же команду дважды и получить разный результат.
- `REPEATABLE READ`. Этот уровень изолированности позволяет решить проблемы, которые возникают на уровне `READ UNCOMMITTED`. Он гарантирует, что любые строки, которые считываются транзакцией, будут выглядеть одинаково при последовательных операциях чтения в пределах одной транзакции, однако теоретически на этом уровне возможна другая проблема, которая называется *фантомным чтением* (phantom reads). Проще говоря, фантомное чтение может произойти в случае, если вы выбираете некоторый диапазон строк, затем другая транзакция вставляет в него новую строку, после чего вы снова выбираете тот же диапазон. В результате вы увидите новую, фантомную строку. InnoDB и XtraDB решают проблему фантомного чтения с помощью многоверсионного управления конкурентным доступом (multiversion concurrency control), о котором мы расскажем далее в этой главе.