

O'REILLY®

4-е издание

MySQL

по максимуму

Проверенные стратегии

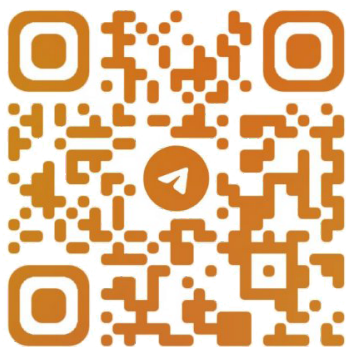
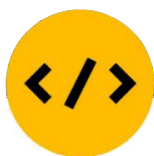


Сильвия Ботрос
Джерemi Тинли

FOURTH EDITION

High Performance MySQL

Proven Strategies for Operating at Scale



Silvia Botros and Jeremy Tinley
Foreword by Jeremy Cole

@CODELIBRARY_IT

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

MySQL по максимуму

Проверенные стратегии

4-е издание

Сильвия Ботрос
Джереми Тинли



Санкт-Петербург • Москва • Минск

2023

ББК 32.988.02-018
УДК 004.738.5
Б86

Ботрос Сильвия, Тинли Джереми

Б86 MySQL по максимуму. 4-е изд. — СПб.: Питер, 2023. — 432 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-2261-5

Хотите выжать из MySQL максимум возможностей? Вам поможет уникальная книга, написанная экспертами для экспертов.

Пора изучать лучшие практики, начиная с постановки целей уровня обслуживания, проектирования схем, индексов, запросов и заканчивая настройкой вашего сервера, операционной системы и оборудования, чтобы реализовать потенциал вашей платформы по максимуму. Администраторы баз данных научатся безопасным и практичным способам масштабирования приложений с помощью репликации, балансировки нагрузки, высокой доступности и отказоустойчивости.

Это издание было обновлено и переработано с учетом последних достижений в области облачного и самостоятельного хостинга MySQL, производительности InnoDB, а также новых функций и инструментов. Вы сможете разработать платформу реляционных данных, которая будет масштабироваться вместе с вашим бизнесом, и узнаете о передовых методах обеспечения безопасности, производительности и стабильности баз данных.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018
УДК 004.738.5

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, такие как Meta Platforms Inc., Facebook, Instagram и др.

ISBN 978-1492080510 англ.

Authorized Russian translation of the English edition of High Performance MySQL 4E, ISBN 9781492080510 © 2022 Silvia Botros and Jeremy Tinley. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-2261-5

© Перевод на русский язык ООО «Прогресс книга», 2023
© Издание на русском языке, оформление ООО «Прогресс книга», 2023
© Серия «Бестселлеры O'Reilly», 2023

Краткое содержание

Отзывы о книге.....	15
Предисловие	16
Введение	18
От издательства	23
Глава 1. Архитектура MySQL	24
Глава 2. Мониторинг в мире проектирования надежности.....	45
Глава 3. Performance Schema	71
Глава 4. Оптимизация операционной системы и оборудования.....	108
Глава 5. Оптимизация настроек сервера	136
Глава 6. Разработка схемы и управление.....	168
Глава 7. Повышение производительности с помощью индексирования.....	201
Глава 8. Оптимизация производительности запросов.....	242
Глава 9. Репликация	284
Глава 10. Резервное копирование и восстановление	317
Глава 11. Масштабирование MySQL	355
Глава 12. MySQL в облаке	384
Глава 13. Соответствие MySQL нормативным требованиям.....	398
Приложение А. Обновление MySQL	419
Приложение Б. MySQL на Kubernetes	425
Об авторах.....	429
Иллюстрация на обложке.....	430

Оглавление

Отзывы о книге.....	15
Предисловие	16
Введение	18
Для кого эта книга	18
Новое в этом издании	19
Используемые в книге соглашения	20
Благодарности к четвертому изданию	21
От Сильвии	21
От Джереми	22
Спасибо рецензентам.....	22
От издательства	23
Глава 1. Архитектура MySQL	24
Логическая архитектура MySQL	24
Управление соединениями и их безопасность	25
Оптимизация и исполнение	26
Управление конкурентным доступом.....	27
Блокировки чтения/записи.....	27
Гранулярность блокировок	28
Транзакции	30
Уровни изолированности.....	32
Взаимоблокировки	34
Ведение журнала транзакций.....	35
Транзакции в MySQL.....	35
Управление конкурентным доступом с помощью многоверсионности	38
Репликация.....	40
Структура файлов данных	41

Подсистема хранения InnoDB	41
Поддержка JSON-документов.....	43
Изменения словаря данных	43
Атомарный DDL	43
Резюме.....	44
Глава 2. Мониторинг в мире проектирования надежности.....	45
Влияние проектирования надежности на DBA группы.....	46
Определение целей уровня обслуживания.....	47
Что нужно, чтобы клиенты были довольны	49
Что измерять.....	50
Определение SLI и SLO.....	50
Решения для мониторинга	51
Мониторинг доступности	53
Мониторинг задержки запросов.....	55
Мониторинг ошибок	56
Проактивный мониторинг.....	57
Измерение долгосрочной эффективности	66
Изучение вашего делового ритма	66
Эффективное отслеживание показателей	67
Применение инструментов мониторинга для контроля производительности	68
Использование SLO для управления общей архитектурой	69
Резюме.....	70
Глава 3. Performance Schema	71
Введение в Performance Schema	71
Список инструментов	72
Организация потребителей.....	74
Потребление ресурсов	75
Ограничения.....	76
Схема sys	77
Кратко о потоках.....	77
Конфигурация.....	78
Включение и отключение Performance Schema	79
Включение и отключение инструментов.....	79
Включение и отключение потребителей.....	81
Настройка мониторинга для конкретных объектов	82

Настройка мониторинга потоков	82
Настройка размера памяти для Performance Schema	84
Значения по умолчанию.....	84
Использование Performance Schema	85
Анализ операторов SQL.....	85
Анализ производительности чтения и записи	95
Анализ блокировок метаданных.....	96
Анализ использования памяти	97
Анализ переменных	100
Анализ наиболее частых ошибок.....	103
Анализ самой Performance Schema.....	104
Резюме.....	107
Глава 4. Оптимизация операционной системы и оборудования.....	108
Что ограничивает производительность MySQL.....	108
Как выбрать процессоры для MySQL	109
Балансировка памяти и дисковых ресурсов	109
Кэширование, чтение и запись	110
Каково ваше рабочее множество	111
Твердотельные хранилища данных	111
Обзор флеш-памяти	112
Сборка мусора	113
Оптимизация производительности с помощью RAID	113
Отказ, восстановление и мониторинг RAID	116
Конфигурация RAID и кэширование.....	118
Конфигурация сети.....	122
Выбор файловой системы	123
Выбор планировщика дисковых очередей.....	126
Память и подкачка	127
Состояние операционной системы	129
Резюме.....	134
Глава 5. Оптимизация настроек сервера	136
Основы конфигурации MySQL	137
Синтаксис, область видимости и динамичность.....	139
Сохраняемые системные переменные.....	140
Побочные эффекты установки переменных.....	141
Планирование изменений ваших переменных.....	142

Чего делать не следует	143
Создание конфигурационного файла MySQL.....	145
Минимальная конфигурация.....	145
Проверка переменных состояния сервера MySQL	147
Настройка использования памяти.....	148
Сколько памяти нужно для соединения.....	148
Резервирование памяти для операционной системы	149
Буферный пул InnoDB.....	149
Кэш потоков	150
Настройка ввода/вывода в MySQL	151
Журнал транзакций InnoDB	153
Буфер журнала.....	154
Табличное пространство InnoDB	156
Прочие параметры настройки ввода/вывода.....	160
Настройка конкурентного доступа в MySQL	160
Настройки безопасности	162
Дополнительные настройки InnoDB	165
Резюме.....	167
Глава 6. Разработка схемы и управление.....	168
Выбор оптимальных типов данных.....	169
Целые числа.....	170
Вещественные числа.....	171
Строковые типы	172
Типы Date и Time	178
Битовые типы данных.....	180
JSON-данные	182
Выбор идентификаторов	186
Специальные типы данных	189
Подводные камни проектирования схемы в MySQL	189
Слишком много столбцов.....	189
Слишком много соединений.....	190
Всемогущий тип ENUM.....	190
Замаскированный тип ENUM	190
NULL изобрели не здесь	190
Управление схемой	191
Управление схемой как часть платформы хранения данных	191
Резюме.....	200

Глава 7. Повышение производительности с помощью индексирования	201
Основы индексирования	202
Типы индексов	203
Преимущества индексов	208
Стратегии индексирования для достижения производительности	209
Префиксные индексы и селективность индексов	209
Многостолбцовые индексы	213
Выбор правильного порядка столбцов	215
Кластерные индексы	218
Покрывающие индексы	227
Использование сканирования индекса для сортировки	229
Избыточные и дублирующие индексы	232
Неиспользуемые индексы	235
Обслуживание индексов и таблиц	236
Поиск и исправление повреждений таблицы	236
Обновление статистики индекса	237
Уменьшение фрагментации индекса и данных	239
Резюме	240
Глава 8. Оптимизация производительности запросов	242
Почему запросы бывают медленными	243
Основная причина медленных запросов — оптимизация доступа к данным	243
Не запрашивает ли вы лишние данные у базы?	244
Не слишком ли много данных анализирует MySQL?	245
Способы реструктуризации запросов	250
Один сложный или несколько простых запросов	251
Разбиение запроса на части	252
Декомпозиция соединения	252
Основные принципы выполнения запросов	254
Клиент-серверный протокол MySQL	255
Состояния запроса	257
Процесс оптимизации запроса	258
Подсистема выполнения запросов	273
Возврат результатов клиенту	274

Ограничения оптимизатора запросов MySQL	274
Ограничения UNION	274
Распространение равенства	275
Параллельное выполнение	276
SELECT и UPDATE для одной и той же таблицы	276
Оптимизация конкретных типов запросов	277
Оптимизация запросов COUNT()	277
Оптимизация запросов с JOIN	279
Оптимизация GROUP BY WITH ROLLUP	280
Оптимизация LIMIT и OFFSET	280
Оптимизация SQL_CALC_FOUND_ROWS	282
Оптимизация UNION	283
Резюме	283
Глава 9. Репликация	284
Обзор репликации	284
Как работает репликация	286
Взгляд на репликацию изнутри	287
Выбор формата репликации	287
Глобальные идентификаторы транзакций	289
Обеспечение безопасности при сбоях репликации	290
Отложенная репликация	291
Многопоточная репликация	292
Полусинхронная репликация	295
Фильтры репликации	296
Отказоустойчивость репликации	298
Запланированные повышения	298
Незапланированные повышения	299
Компромиссы повышения	300
Топологии репликации	300
Активный/пассивный	300
Активный/пул чтения	302
Нерекомендуемые топологии	304
Администрирование и обслуживание репликации	308
Мониторинг репликации	308
Измерение отставания репликации	309
Как определить, согласованы ли реплики с источником	310

Проблемы с репликацией и их решения.....	311
Повреждение двоичных журналов в источнике	312
Неуникальные идентификаторы серверов	312
Неопределенные идентификаторы серверов	312
Отсутствующие временные таблицы	313
Репликация не всех обновлений.....	313
Слишком большое отставание репликации	314
Чрезмерно большие пакеты от источника.....	315
Отсутствие места на диске	315
Ограничения репликации	315
Резюме.....	316
Глава 10. Резервное копирование и восстановление	317
Зачем нужно резервное копирование.....	319
Определение требований к восстановлению	319
Проектирование решения для резервного копирования MySQL.....	321
Оперативное или автономное резервное копирование?	323
Логическое и физическое резервное копирование	324
Что нужно копировать	327
Инкрементное и дифференциальное резервное копирование	328
Репликация.....	330
Управление двоичными журналами и их резервное копирование	331
Инструменты резервного копирования и восстановления	332
MySQL Enterprise Backup	333
Percona XtraBackup.....	333
mysdumper	333
mysqldump.....	334
Резервное копирование данных.....	334
Логические SQL-дампы	334
Снимки файловой системы	336
Percona XtraBackup.....	343
Восстановление из резервной копии.....	347
Восстановление логических резервных копий.....	348
Восстановление физических файлов из моментального снимка	350
Восстановление с помощью Percona XtraBackup.....	351
Запуск MySQL после восстановления физических файлов.....	352
Резюме.....	353

Глава 11. Масштабирование MySQL	355
Что такое масштабируемость	355
Рабочие нагрузки, связанные с чтением и записью	357
Знание рабочей нагрузки	358
Рабочие нагрузки, связанные с чтением	359
Рабочие нагрузки, связанные с записью	359
Функциональное сегментирование	360
Масштабирование чтения с помощью пулов чтения	361
Управление конфигурацией пулов чтения	364
Проверки работоспособности пулов чтения	365
Выбор алгоритма балансировки нагрузки	366
Очередь	368
Масштабирование операций записи с помощью шардирования	369
Выбор схемы сегментирования	370
Несколько ключей сегментирования	372
Запросы в разных шардах	373
Vitess	374
ProxySQL	378
Резюме	383
Глава 12. MySQL в облаке	384
Управляемая MySQL	384
Amazon Aurora для MySQL	385
GCP Cloud SQL	388
MySQL на виртуальных машинах	389
Типы машин в облаке	390
Выбор правильного типа машины	390
Выбор правильного типа диска	392
Дополнительные советы	394
Резюме	397
Глава 13. Соответствие MySQL нормативным требованиям	398
Что такое соответствие	399
Элементы контроля сервисной организацией типа 2	399
Закон Сарбейнса — Оксли	399
Стандарт безопасности данных индустрии платежных карт	400
Закон о переносимости и подотчетности медицинского страхования	400

Федеральная программа управления рисками и авторизацией.....	401
Общий регламент по защите данных	401
Schrems II	401
Выстраивание контроля за соблюдением нормативных требований.....	402
Управление секретами.....	403
Разделение ролей и данных	406
Отслеживание изменений.....	407
Процедуры резервного копирования и восстановления	415
Резюме.....	417
Приложение А. Обновление MySQL	419
Зачем обновлять версию	419
Жизненный цикл обновления	420
Тестирование обновлений.....	421
Тестирование среды разработки	421
Копия производственных данных	422
Реплика.....	422
Инструментарий	422
Масштабное обновление.....	422
Резюме.....	424
Приложение Б. MySQL на Kubernetes	425
Предоставление ресурсов с помощью Kubernetes	425
Тщательно определите свою цель.....	426
Выберите плоскость управления	426
Более тонкие детали	426
Резюме.....	428
Об авторах.....	429
Иллюстрация на обложке.....	430

Отзывы о книге

Мне нравится, что в новом издании книги акценты смещаются на современное прагматическое мышление командных игроков, создающих ценность для бизнеса. Материал о том, как работают базы данных, по-прежнему подробно освещается, но теперь со свежим гуманистическим подходом, который очень необходим.

*Барон Шварц, ведущий автор «MySQL по максимуму»,
2-е и 3-е издания*

«MySQL по максимуму» была одной из основных книг в мире MySQL с момента выхода первого издания 17 лет назад. MySQL постоянно движется вперед, и Сильвия и Джереми проделали отличную работу, приведя эту важную книгу в соответствие с сегодняшним состоянием MySQL.

Джереми Коул

Это последнее издание, обновленное с учетом современных практик, содержит полезные советы для администраторов и разработчиков MySQL.

Шломи Ноах, инженер баз данных, PlanetScale

«MySQL по максимуму» получило новый фокус. Речь больше не идет о том, чтобы выжимать из MySQL каждую унцию мощности. Теперь у нас есть большая экосистема инструментов и поставщиков.

Сильвия и Джереми прекрасно рассказывают, как MySQL вписывается в новую картину. Прочитать эту книгу обязательно, если вы запускаете MySQL на любой платформе.

*Сузу Сугумаране, технический директор PlanetScale, один
из создателей Vitess*

Сильвия и Джереми проделали фантастическую работу, сохранив первоначальный дух книги и обновив ее, чтобы охватить быстро меняющийся мир MySQL.

*Петр Зайцев, основатель и генеральный директор Persona
и соавтор «MySQL по максимуму», 3-е издание*

Предисловие

Новенький экземпляр «MySQL по максимуму» был первой книгой, которая ложилась на стол каждого вновь нанятого администратора баз данных, системного инженера или разработчика баз данных с тех пор, как она вышла почти два десятилетия назад.

Когда Джереми Заводны и Дерек Баллинг решили написать книгу о работе с MySQL на уровне, позволяющем внести ясность и структурировать многолетние загадки, ей суждено было стать классикой в мире MySQL. За прошедшие годы и несколько редакций книги часть содержимого оригинала и последующих обновлений сохранилась, а часть — не очень.

Сама MySQL развилась, сообщество MySQL сильно изменилось, и способы, которыми мы используем MySQL, поменялись. Теперь, в четвертом издании, Сильвия и Джереми берутся за колоссальную неблагодарную работу по обновлению этого классического труда для современной эпохи — и они идеально подходят для этой задачи.

Все время моего знакомства с MySQL (уже более 20 лет!) в сообществе MySQL единственной неизменной вещью была, ну, несогласованность. Все используют MySQL (и базы данных в целом) немного по-разному, и у всех разные ожидания от нее. Каждый принимает несколько хороших решений, несколько исполненных благих намерений, но сомнительных решений и всегда значительную долю плохих. Иногда добиться прогресса легко, но иногда требуется новый взгляд на проблему и мудрый совет, полученный непосредственно от эксперта.

Сильвия и Джереми как раз такие эксперты. Все области, в том числе архитектура MySQL, оптимизация, репликация, резервное копирование и многое другое, выиграют от того, что они поделятся своим обширным опытом работы с MySQL. В этом новом, четвертом издании многие темы получили новую трактовку, было удалено много устаревшего материала, исправлены ошибки, и в материал внесен новый и свежий стиль.

Как и оригинальное (теперь устаревшее и просто маленькое) первое издание, четвертое обещает помочь новейшему поколению разработчиков, администра-

торов баз данных и их боссов войти в новый мир MySQL — иногда с волнением, но, возможно, иногда с пинками и криками.

Спасибо, Сильвия и Джереми, за вашу усердную работу по воспитанию следующего поколения фанатов MySQL, которые будут обеспечивать безопасность мировых данных, благодаря вам лучшие в мире веб-сайты и другие системы, управляемые данными, станут работать на пике своих возможностей.

Поздравляю с тем, что удалось это сделать с помощью COVID и всего остального. А все мы обязательно обеспечим экземпляром этой книги всех новых администраторов баз данных.

*Джереми Коул, окрестности Рино,
Невада, октябрь 2021 года*

Введение

Официальная документация от Oracle дает вам знания, необходимые для установки и настройки MySQL и взаимодействия с ней. Наша книга служит дополнением к этой документации, помогая вам понять, как наилучшим образом использовать MySQL в качестве мощной платформы данных для вашего сценария применения.

В этом издании также раскрывается растущая роль соответствия требованиям и безопасности как части работы с базой данных. Новые реалии, такие как законы о конфиденциальности и суверенитете данных, изменили то, как компании создают свои продукты, и это, естественно, вносит новые сложности в развитие технической архитектуры.

Для кого эта книга

Эта книга в первую очередь предназначена для инженеров, желающих улучшить свой опыт работы с MySQL. Ее авторы предполагают, что аудитория знакома с основными принципами использования системы управления реляционными базами данных (RDBMS). Мы также предполагаем наличие некоторого опыта общего системного администрирования, работы с сетями и операционными системами.

Мы предложим вам проверенные стратегии масштабируемой эксплуатации MySQL с применением современной архитектуры и новейших инструментов и практик.

В конечном счете мы надеемся, что знания о внутреннем устройстве MySQL и стратегиях масштабирования, которые вы получите из этой книги, помогут вам в масштабировании уровня хранения данных в вашей организации. И еще надеемся, что новообретенное понимание поможет вам изучить и применить на практике методический подход к проектированию, поддержке и устранению неполадок в архитектуре, построенной на базе MySQL.

Новое в этом издании

«MySQL по максимуму» была неременным атрибутом сообщества инженеров баз данных в течение многих лет — предыдущие издания были выпущены в 2004, 2008 и 2012 годах. Их цель всегда заключалась в том, чтобы научить разработчиков и администраторов оптимизировать MySQL для любого случая потери производительности, сосредоточившись на глубоком анализе внутреннего устройства, объяснив, что означают различные параметры настройки, и вооружив пользователя знаниями, позволяющими эффективно изменять эти параметры. Это издание преследует ту же цель, но с несколько иной направленностью.

Начиная с третьего издания, экосистема MySQL претерпела множество изменений — были выпущены три новые основные версии. Ландшафт инструментальных средств значительно расширился за пределы сценариев Perl и Bash, и они превратились в полноценные инструментальные решения. Были созданы совершенно новые проекты с открытым исходным кодом, позволяющие организациям радикально изменить управление масштабированием MySQL.

Даже традиционная роль администратора базы данных (DBA) изменилась. В ИТ-отрасли есть старая шутка, гласящая, что DBA означает Don't Bother Asking («Не трудись спрашивать»). У администраторов баз данных была репутация «спящих полицейских» в жизненном цикле разработки программного обеспечения (SDLC), не из-за отрицательного отношения к ним, а просто потому, что базы данных развивались не так быстро, как остальная часть SDLC вокруг них.

Благодаря таким книгам, как *Database Reliability Engineering: Designing and Operating Resilient Database Systems*¹ Лейна Кэмпбелла и Черити Мейджорс (O'Reilly), новой реальностью стало то, что технические организации рассматривают инженеров баз данных больше как средство обеспечения роста бизнеса, а не просто как операторов всех баз данных. Когда-то основными задачами администратора баз данных было проектирование схемы и оптимизация запросов, теперь он отвечает за обучение этим навыкам разработчиков и управление системами, которые позволяют разработчикам быстро и безопасно разворачивать собственные изменения схемы.

Из-за этих изменений основное внимание больше не стоит уделять оптимизации MySQL для ускорения на несколько процентов. Мы думаем, что теперь книга «MySQL по максимуму» предназначена для предоставления людям информации, необходимой для принятия обоснованных решений о том, как лучше всего

¹ Кэмпбелл Л., Мейджорс Ч. Базы данных. Инжиниринг надежности. — Питер, 2020.

использовать MySQL. Все начинается с объяснения того, как устроена MySQL, так что можно понять, что такое MySQL, а что — нет¹. Современные релизы MySQL предлагают разумные значения по умолчанию, и вам нужно настраивать очень немного, если только вы не сталкиваетесь с конкретной проблемой масштабирования. Современные команды теперь чаще имеют дело с изменениями схемы и проблемами соответствия нормативам и сегментирования. Мы хотим, чтобы «MySQL по максимуму» стала исчерпывающим руководством по тому, как современные компании могут задействовать MySQL при масштабируемой эксплуатации.

Используемые в книге соглашения

В данной книге применяются следующие шрифтовые соглашения.

Курсив

Отмечает новые термины.

Рубленый шрифт

Выделяет URL и адреса электронной почты.

Моноширинный шрифт

Показывает использованные внутри абзацев имена и расширения файлов, элементы программ, такие как переменные или имена функций, базы данных, переменные окружения, операторы и ключевые слова. Также применяется для листингов программ.



Эта пиктограмма означает совет или указание.



Эта пиктограмма означает общее примечание.



Эта пиктограмма указывает на предупреждение или предостережение.

¹ Известны примеры того, как некоторые люди использовали MySQL в качестве очереди, а затем на собственном горьком опыте поняли, почему это плохо. Наиболее часто упоминаемыми причинами были накладные расходы на выборку новых элементов очереди, управление блокировкой записей при их обработке и громоздкие таблицы очереди, получающиеся по мере роста объема данных с течением времени.

Благодарности к четвертому изданию

От Сильвии

Прежде всего я хотела бы поблагодарить свою семью. Родителей, которые пожертвовали стабильной работой и жизнью в Египте, чтобы привезти меня и моего брата в Соединенные Штаты. Мужа Армеа — за то, что поддерживал меня в этот и все прошлые годы построения карьеры, когда я принимала один вызов за другим, достигнув кульминации в своих достижениях.

Я начала заниматься технологиями, будучи иммигранткой, которая прервала обучение на Ближнем Востоке, чтобы осуществить свою мечту — переехать в Соединенные Штаты. Получив степень в государственном университете в Калифорнии, я устроилась на работу в Нью-Йорке. Второе издание этой книги было самой первой технической книгой, которую я купила на свои деньги и которая не была учебником для колледжа. Я в долгу перед авторами предыдущих изданий, преподавших мне множество фундаментальных уроков, которые подготовили меня к управлению базами данных на протяжении всей моей карьеры. Я благодарна за поддержку очень многим людям, с которыми работала прежде. Их поддержка побудила меня написать новую редакцию этой книги, которая многому меня научила в начале карьеры. Я хотела бы поблагодарить Тима Дженкинса, бывшего технического директора SendGrid, за то, что нанял меня на работу всей моей жизни, хотя я сказала ему в своем интервью, что он неправильно использует репликацию MySQL, и за то, что он доверил мне то, что стало ракетным кораблем.

Я хотела бы поблагодарить всех замечательных женщин, работающих в сфере технологий, которые были моей группой поддержки и болельщиками. Особая благодарность Камилле Фурнье и доктору Николь Форсгрэн за написание двух книг, которые повлияли на последние несколько лет моей карьеры и изменили мой взгляд на повседневную работу.

Спасибо моей команде в Twilio. Шону Килгору — за то, что под его влиянием я стала гораздо лучшим инженером, который заботится не только о базах данных. Джону Мартину — это самый оптимистичный человек, с которым я когда-либо работала. Спасибо Лайне Кэмпбелл и ее команде PalominoDB (позже приобретенной Pythian), которые помогли мне и многому научили в самые трудные годы, а также Барону Шварцу — за то, что он вдохновил меня написать о своем опыте.

Наконец, спасибо Вирджинии Уилсон — за то, что она была превосходным редактором и помогла превратить мой поток идей в предложения, которые имеют смысл, и делала это очень доброжелательно.

От Джереми

Когда Сильвия обратилась ко мне с просьбой помочь с этой книгой, это было в разгар необычайно напряженного периода жизни большинства людей — глобальной пандемии, которая началась в 2020 году. Я не был уверен, что хочу добавить в свою жизнь еще больше стресса. Моя жена Селена сказала, что я пожалею, если не соглашусь, а я знаю, что с ней лучше не спорить. Она всегда поддерживала меня и поощряла быть наилучшим человеком из возможных. Я всегда буду любить ее за все, что она для меня делает.

Моей семье, коллегам и друзьям по сообществу: без вас я бы никогда не добился этого. Вы все научили меня быть тем, кто я есть сегодня. Моя карьера — это сумма опыта общения со всеми вами. Вы научили меня, как принимать критику, как подавать пример, как терпеть неудачи и восстанавливаться и, самое главное, тому, что сумма лучше, чем индивидуальность.

Наконец, хочу поблагодарить Сильвию, которая доверила мне привнести в эту книгу общее понимание, но другую точку зрения. Надеюсь, я оправдал ваши ожидания.

Спасибо рецензентам

Авторы также хотят отметить рецензентов, которые помогли сделать эту книгу именно такой — это Аиша Имран, Эндрю Регнер, Барон Шварц, Дэниел Нихтер, Хейли Андерсон, Иван Мора Перес, Джем Леони, Джарид Ремиллард, Дженнифер Дэвис, Джереми Коул, Кейт Уэллс, Крис Хамуд, Ник Визас, Шубхекша Джалан, Том Крупер и Уилл Ганти. Спасибо всем за ваше время и усилия.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ГЛАВА 1

Архитектура MySQL

Архитектурные характеристики MySQL делают эту СУБД полезной для широкого круга целей. Хотя она и неидеальна, она достаточно гибка для того, чтобы хорошо работать как в малых, так и в больших средах. Они варьируются от личного веб-сайта до крупномасштабных корпоративных приложений. Чтобы получить максимальную отдачу от MySQL, вам нужно понять ее структуру, чтобы вы могли работать с ней, а не против нее.

В этой главе представлен общий обзор архитектуры сервера MySQL, основных различий между подсистемами хранения и того, почему эти различия важны. Мы попытались объяснить MySQL, упростив детали и показав примеры. Это обсуждение будет полезно тем, кто плохо знаком с серверами баз данных, а также читателям, которые являются экспертами в других серверах баз данных.

Логическая архитектура MySQL

Хорошее понимание того, как компоненты MySQL работают вместе, поможет вам понять сервер. На рис. 1.1 представлен логический вид архитектуры MySQL.

На самом верхнем уровне — уровне клиентов — располагаются службы, не являющиеся уникальными для MySQL. Это службы, в которых нуждается большинство сетевых клиент-серверных инструментов или серверов: обработка соединений, аутентификация, безопасность и т. д.

На втором уровне все намного интереснее. Здесь находится большая часть «мозгов» MySQL, включая код для обработки запросов, анализа, оптимизации и всех встроенных функций (например, даты, время, математика и шифрование). На этом же уровне находится любая функциональность, предоставляемая подсистемами хранения, например хранимые процедуры, триггеры и представления.

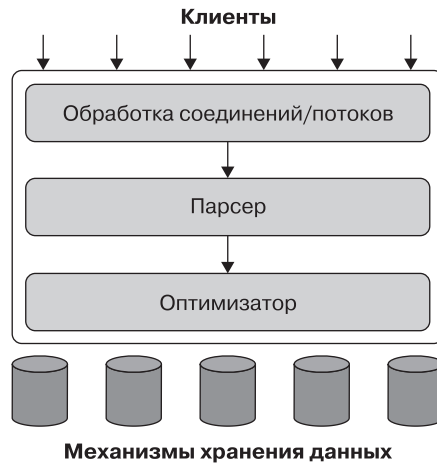


Рис. 1.1. Логический взгляд на архитектуру сервера MySQL

Третий уровень содержит подсистемы хранения данных. Они отвечают за хранение в MySQL всех данных и их извлечение. Подобно различным файловым системам, доступным для GNU/Linux, каждая подсистема хранения имеет свои преимущества и недостатки. Сервер взаимодействует с ними через API подсистемы хранения. Этот API скрывает различия между подсистемами хранения и делает их в значительной степени прозрачными на уровне запросов. Он также содержит пару десятков низкоуровневых функций, которые выполняют такие операции, как «начать транзакцию» или «извлечь строку, содержащую данный первичный ключ». Подсистемы хранения не анализируют SQL¹ и не взаимодействуют друг с другом — они просто отвечают на запросы от сервера.

Управление соединениями и их безопасность

По умолчанию каждое клиентское соединение получает собственный поток внутри серверного процесса. Запросы соединения выполняются в этом единственном потоке, который, в свою очередь, находится на одном ядре или ЦП. Сервер поддерживает кэш готовых к использованию потоков, поэтому их не нужно создавать и уничтожать для каждого нового подключения².

¹ Единственным исключением является InnoDB, который анализирует определения внешнего ключа, потому что сервер MySQL еще не реализует их сам.

² MySQL 5.5 и более поздние версии поддерживают API, который может принимать подключаемые модули объединения потоков, хотя и не часто используемые. Обычно объединение потоков выполняется на уровнях доступа, которые мы обсудим в главе 5.

Когда клиенты (приложения) подключаются к серверу MySQL, сервер должен их аутентифицировать. Аутентификация основана на имени пользователя, адресе хоста, с которого происходит соединение, и пароле. Сертификаты X.509 можно задействовать также через соединение Transport Layer Security (TLS). После подключения клиента сервер проверяет, есть ли у того привилегии для каждого запроса, который он выдает, например, разрешено ли клиенту выполнять оператор `SELECT`, который обращается к таблице `Country` в базе данных `world`.

Оптимизация и исполнение

MySQL анализирует запросы для создания внутренней структуры (дерева разбора), а затем применяет множество оптимизаций. Сюда может входить переписывание запроса, определение порядка чтения таблиц, выбор используемых индексов и т. д. Через специальные ключевые слова в запросе можно передавать оптимизатору подсказки, влияющие на процесс принятия им решения. Вы также можете обратиться к серверу за объяснением различных аспектов оптимизации. Это позволит узнать, какие решения принимает сервер, и даст вам ориентир для изменения запросов, схем и настроек таким образом, чтобы все работало максимально эффективно. Подробнее об этом читайте в главе 8.

Оптимизатору на самом деле неважно, какую подсистему хранения использует конкретная таблица, но подсистема хранения влияет на то, как сервер оптимизирует запрос. Оптимизатор запрашивает у подсистемы хранения некоторые его возможности и стоимость определенных операций, а также статистику по данным таблицы. Например, некоторые подсистемы хранения поддерживают типы индексов, которые могут быть полезны для определенных запросов. Вы сможете больше узнать об оптимизации схемы и индексации в главах 6 и 7.

В более старых версиях MySQL использовала внутренний кэш запросов, чтобы увидеть, сможет ли она обработать оттуда результаты. Однако по мере роста параллелизма кэш запросов стал печально известным узким местом. Начиная с MySQL 5.7.20 кэш запросов официально объявлен устаревшим как функция MySQL, а в версии 8.0 он полностью удален. Несмотря на то что кэш запросов больше не является основной частью сервера MySQL, кэширование часто обслуживаемых наборов результатов является хорошей практикой. Популярным шаблоном проектирования является кэширование данных с помощью `memcached` или `Redis`, но это выходит за рамки книги.

Управление конкурентным доступом

Каждый раз, когда несколько запросов должны одновременно изменить данные, возникает проблема управления конкурентным доступом. Для целей данной главы MySQL должна делать это на двух уровнях — сервера и подсистемы хранения. Мы представим упрощенный обзор того, как MySQL работает с параллельными операциями чтения и записи, чтобы вы получили общее представление об этой теме, позволяющее разобраться в материале остальной части главы.

Чтобы проиллюстрировать, как MySQL обрабатывает параллельную работу с одним и тем же набором данных, используем в качестве примера традиционный файл электронной таблицы. Эта таблица состоит из строк и столбцов, как и таблица базы данных. Предположим, что файл находится на вашем ноутбуке и доступ к нему есть только у вас. В таком случае нет потенциальных конфликтов, ведь только вы можете вносить изменения в файл. Теперь представьте, что вам нужно совместно с коллегой работать над этой электронной таблицей. Теперь она находится на общем сервере, к которому у вас обоих есть доступ. Что происходит, когда вам обоим нужно внести изменения в этот файл одновременно? А что, если есть целая команда людей, активно пытающихся редактировать эту электронную таблицу, добавляя и удаляя ячейки? Можно сказать, что они должны по очереди вносить изменения, но это неэффективно. Нам нужен подход для обеспечения одновременного доступа к электронной таблице большого объема.

Блокировки чтения/записи

Чтение из электронной таблицы не вызывает таких проблем. Нет ничего плохого в том, что несколько клиентов одновременно читают один и тот же файл: поскольку они не вносят изменений, это ничем не грозит. Но что произойдет, если кто-то попытается удалить ячейку с номером A25, пока другие читают электронную таблицу? Это зависит от обстоятельств, но читатель может получить искаженное или противоречивое представление данных. Таким образом, чтобы не возникла опасность, даже читать из электронной таблицы требуется с особой осторожностью.

Если вы думаете об электронной таблице как о таблице базы данных, легко увидеть, что в этом контексте проблема та же самая. Во многих отношениях электронная таблица — это просто обычная таблица базы данных. Изменение строк в таблице базы данных очень похоже на удаление или изменение содержимого ячеек в файле электронной таблицы.

Решение этой классической проблемы управления параллелизмом довольно простое. Системы, имеющие дело с одновременным доступом для чтения/записи, обычно реализуют систему блокировки, состоящую из двух типов блокировки. Эти блокировки обычно известны как *разделяемые блокировки* и *монопольные блокировки*, или *блокировки чтения и записи*.

Не вдаваясь в подробности технологии блокировки, можем описать концепцию следующим образом. Блокировки чтения ресурса являются общими или взаимно не блокирующими: множество клиентов могут читать из ресурса одновременно и не мешать друг другу. В то же время блокировки записи являются эксклюзивными, то есть они исключают возможность установки как блокировки чтения, так и других блокировок записи, потому что единственная безопасная политика — иметь одного клиента, записывающего в ресурс в заданное время, и предотвращать все операции чтения, когда это происходит.

В мире баз данных блокировка происходит постоянно: MySQL должна препятствовать тому, чтобы один клиент считывал часть данных, в то время как другой изменяет ее. Если сервер базы данных работает приемлемым образом, управление блокировками выполняется достаточно быстро, чтобы не быть заметным для клиентов. В главе 8 мы обсудим, как настроить ваши запросы, чтобы избежать проблем с производительностью, вызванных блокировкой.

Гранулярность блокировок

Один из способов улучшить конкурентность разделяемого ресурса — быть более избирательными в отношении того, что вы блокируете. Вместо блокировки всего ресурса заблокируйте только его часть, содержащую данные, которые необходимо изменить. А еще лучше заблокируйте только ту часть данных, которую вы планируете менять. Минимизация объема данных, которые вы блокируете в любой момент времени, позволяет одновременно выполнять несколько изменений одного и того же ресурса, если эти операции не конфликтуют друг с другом.

К сожалению, блокировки не бесплатны — они потребляют ресурсы. Каждая операция блокировки — получение блокировки, проверка на свободную блокировку, освобождение блокировки и т. д. — увеличивает накладные расходы. Если система тратит слишком много времени на управление блокировками вместо хранения и извлечения данных, производительность может пострадать.

Стратегия блокировки — это компромисс между накладными расходами на блокировку и безопасностью данных, он влияет на производительность. Большинство коммерческих серверов баз данных не предоставляют большого выбора: вы получаете то, что известно как блокировка на уровне строк в ваших таблицах, с множеством часто сложных способов, обеспечивающих хорошую производительность при множестве блокировок. Блокировки — это то, как

базы данных реализуют гарантии согласованности данных. Опытному оператору базы данных придется дойти до чтения исходного кода, чтобы определить наиболее подходящий набор настраиваемых конфигураций для оптимизации компромисса между скоростью и безопасностью данных.

В то же время MySQL предлагает выбор. Ее подсистемы хранения могут реализовывать собственные политики блокировки и гранулярность блокировок. Управление блокировками — очень важное решение при проектировании подсистемы хранения: установка гранулярности блокировок на определенном уровне может повысить производительность для определенных целей, но сделать этот движок менее подходящим для других. Поскольку MySQL предлагает несколько подсистем хранения, здесь не требуется единого универсального решения. Рассмотрим две наиболее важные стратегии блокировки.

Табличные блокировки

Самая простая стратегия блокировки, доступная в MySQL и требующая наименьших затрат, — это *блокировка таблицы*. Она аналогична блокировкам электронных таблиц, описанным ранее: блокирует всю таблицу. Когда клиент хочет произвести запись в таблицу (вставить, удалить, обновить и т. д.), он получает блокировку записи. Это тормозит все другие операции чтения и записи. Когда никто не пишет, читатели могут получить блокировки чтения, которые не конфликтуют с другими блокировками чтения.

Табличные блокировки имеют варианты для повышения производительности в определенных ситуациях. Например, блокировка таблицы `READ LOCAL` разрешает некоторые типы параллельных операций записи. Очереди блокировки записи и чтения разделены, при этом очередь записи имеет более высокий приоритет, чем очередь чтения¹.

Построчные блокировки

Стиль блокировки, обеспечивающий наилучший параллелизм и влекущий за собой наибольшие накладные расходы, — это использование построчных блокировок. Возвращаясь к аналогии с электронными таблицами, построчная блокировка будет такой же, как блокировка только строки в электронной таблице. Это позволяет серверу выполнять больше одновременных операций записи, но увеличивает затраты на отслеживание того, кто заблокировал каждую строку, как долго они были открыты и какие применялись блокировки строк, а также на снятие блокировок, когда они больше не нужны.

¹ Рекомендуем вам прочитать документацию об эксклюзивных и разделяемых блокировках, блокировках с намерением и блокировках записи (<https://oreil.ly/EPfwc>).

Построчные блокировки реализованы в подсистеме хранения, а не на сервере. Сервер обычно¹ не знает о блокировках, реализованных в подсистемах хранения, и, как вы увидите далее в этой главе и на протяжении всей книги, все подсистемы хранения 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 операторов SQL, которые обрабатываются *атомарно*, то есть как цельная единица работы. Если механизм базы данных может применить всю группу операторов к базе данных, он делает это, но если какой-либо из операторов не может быть выполнен из-за сбоя или по другой причине, ни один из них не применяется. Всё или ничего.

Немногое из этого раздела характерно именно для MySQL. Если вы уже знакомы с транзакциями ACID, смело переходите к подразделу «Транзакции в MySQL» далее в этом разделе.

Банковское приложение — классический пример, демонстрирующий, почему необходимы транзакции². Представьте базу данных банка с двумя таблицами: *checking* (текущие счета) и *savings* (сберегательные счета). Чтобы перевести 200 долларов с расчетного счета Джейн на ее сберегательный счет, необходимо выполнить как минимум три шага.

1. Убедиться, что остаток на ее текущем счете превышает 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 сбоя любого из шагов можно было выполнить откат всех выполненных.

Вы начинаете транзакцию с помощью команды `START TRANSACTION`, а затем либо сохраняете изменения командой `COMMIT`, либо отменяете их с помощью команды `ROLLBACK`. Итак, SQL для примера транзакции может выглядеть следующим образом:

¹ Существуют блокировки метаданных, которые используются при изменении имени таблицы или изменении схемы, а в версии 8.0 мы познакомились с «функциями блокировки на уровне приложения». При обычных изменениях данных внутренняя блокировка выполняется подсистемой InnoDB.

² Хотя это обычное учебное упражнение, большинство банков на самом деле полагаются на ежедневную сверку, а не на строгие транзакционные операции в течение дня.

```
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, если сервер базы данных выйдет из строя при выполнении строки 4? Кто знает? Клиент, вероятно, только что потерял 200 долларов. А что, если другой процесс вклинится между выполнением строк 3 и 4 и снимет весь остаток с текущего счета? Банк предоставил клиенту кредит в размере 200 долларов, даже не подозревая об этом.

И в этой последовательности операций существует гораздо больше возможностей для отказа. Вы могли видеть обрывы соединения, тайм-ауты или даже сбой сервера базы данных, выполняющего их в середине операций. Обычно именно поэтому существуют очень сложные и медленные системы с двухфазной фиксацией для смягчения последствий всевозможных сценариев сбоев. Транзакций недостаточно, пока система не проходит тест ACID. ACID расшифровывается как *atomicity, consistency, isolation и durability* (атомарность, согласованность, изолированность и долговечность). Это тесно связанные критерии, которым должна соответствовать система обработки транзакций с защитой данных.

- *Атомарность.* Транзакция должна функционировать как единая неделимая единица работы, чтобы вся она была либо выполнена, либо отменена. Не существует такого понятия, как частично завершенная атомарная транзакция: либо всё, либо ничего.
- *Согласованность.* База данных должна всегда переходить из одного согласованного состояния в другое. В нашем примере согласованность гарантирует, что сбой между строками 3 и 4 не приведет к исчезновению 200 долларов с текущего счета. Если транзакция никогда не фиксируется, ни одно из ее изменений никогда не отражается в базе данных.
- *Изолированность.* Результаты транзакции обычно невидимы для других транзакций до тех пор, пока она не будет завершена. В нашем примере это гарантирует, что, если программа суммирования остатков на банковских счетах будет запущена после строки 3, но перед строкой 4, она все равно увидит 200 долларов на текущем счете. Когда позже в этой главе будем обсуждать уровни изолированности, вы поймете, почему мы сказали «обычно невидимы».
- *Долговечность.* После подтверждения изменения транзакции становятся постоянными. Это означает, что изменения должны быть записаны, чтобы данные не были потеряны при сбое системы. Однако долговечность — это немного расплывчатое понятие, потому что на самом деле существует много уровней. Некоторые стратегии обеспечения надежности гарантируют

безопасность более надежно, чем другие, но ничто не может быть на 100 % долговечным (если бы сама база данных была действительно надежной, то как резервные копии могли бы увеличить ее надежность?).

Транзакции ACID и гарантии, предоставляемые ими в подсистеме InnoDB, в частности, — это одна из самых сильных и наиболее зрелых функций MySQL. Несмотря на то что они связаны с определенными компромиссами в отношении пропускной способности, при правильном применении они могут избавить вас от реализации большого количества сложной логики на прикладном уровне.

Уровни изолированности

Изолированность — более сложное понятие, чем кажется на первый взгляд. Стандарт ANSI SQL¹ определяет четыре уровня изолированности. Если вы новичок в мире баз данных, мы настоятельно рекомендуем ознакомиться с общим стандартом ANSI SQL, прежде чем возвращаться к чтению о конкретной реализации MySQL. Цель этого стандарта — определить правила, по которым изменения видны или не видны внутри и вне транзакции. Более низкие уровни изолированности обычно допускают большую степень конкурентного доступа и влекут за собой меньшие накладные расходы.



Каждая подсистема хранения реализует уровни изолированности немного по-разному, и они не обязательно соответствуют вашим ожиданиям, если вы привыкли к другой СУБД (поэтому в этом разделе мы не будем вдаваться в исчерпывающие подробности). Вам следует прочитать руководства для тех подсистем хранения, которые решите использовать.

Кратко рассмотрим четыре уровня изолированности.

- **READ UNCOMMITTED.** На уровне изолированности **READ UNCOMMITTED** транзакции могут просматривать результаты незавершенных транзакций. На этом уровне может возникнуть много проблем, если вы не знаете абсолютно точно, что делаете, и не имеете для этого веской причины. Этот уровень редко используется на практике, потому что его производительность лишь немного лучше, чем у других уровней, которые имеют много преимуществ. Чтение незафиксированных данных известно также как *черновое* или «грязное» чтение (*dirty read*).
- **READ COMMITTED.** Уровень изолированности по умолчанию в большинстве систем баз данных (но не в MySQL!) — **READ COMMITTED**. Он удовлетворяет про-

¹ Для получения дополнительной информации прочитайте обзор ANSI SQL (<https://oreil.ly/joikF>) Адриана Койлера (Adrian Coyle) и объяснение моделей согласованности (<http://jepsen.io/consistency>) Кайла Кингсбери (Kyle Kingsbury).

стому определению изолированности, приведенному ранее: транзакция будет продолжать видеть изменения, которые к моменту ее начала подтверждены другими транзакциями, а произведенные ею изменения останутся невидимыми для других транзакций, пока текущая транзакция не будет подтверждена. Этот уровень по-прежнему допускает так называемое *неповторяющееся чтение* (nonrepeatable read). Это означает, что вы можете выполнить один и тот же оператор дважды и увидеть разные данные.

- REPEATABLE READ позволяет решить проблемы, которые возникают на уровне READ UNCOMMITTED. Он гарантирует, что любые строки, прочитанные транзакцией, будут выглядеть одинаково при последующем чтении в рамках одной и той же транзакции, но теоретически по-прежнему допускает другую сложную проблему, которая называется *фантомным чтением* (phantom reads). Если попросту, фантомное чтение может произойти в случае, когда вы выбираете какой-то 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), которое мы объясним позже в этой главе.

Уровень изолированности REPEATABLE READ устанавливается в MySQL по умолчанию.

- SERIALIZABLE. Самый высокий уровень изолированности, SERIALIZABLE, решает проблему фантомного чтения, заставляя транзакции выполняться в таком порядке, который исключает возможность конфликта. В двух словах: SERIALIZABLE блокирует каждую читаемую строку. На этом уровне может возникнуть множество тайм-аутов и конфликтов из-за блокировки. Нам редко встречались люди, использующие этот уровень, но потребности вашего приложения могут заставить применять его, смирившись с меньшей степенью конкурентного доступа, но обеспечивая стабильность данных.

В табл. 1.1 приведена сводка различных уровней изолированности и указаны недостатки, свойственные каждому из них.

Таблица 1.1. Уровни изолированности ANSI SQL

Уровень изолированности	Возможность чернового чтения	Возможность неповторяющегося 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

Взаимоблокировки возникают тогда, когда две и более транзакции взаимно удерживают и запрашивают блокировки одних и тех же ресурсов, создавая циклическую зависимость. Взаимоблокировки появляются, когда транзакции пытаются заблокировать ресурсы в разном порядке. Они могут возникнуть всякий раз, когда несколько транзакций блокируют одни и те же ресурсы. Для примера рассмотрим две транзакции, обращающиеся к таблице `StockPrice`, которая имеет первичный ключ `(stock_id, date)`.

Транзакция 1:

```
START TRANSACTION;
UPDATE StockPrice SET close = 45.50 WHERE stock_id = 4 and date = '2020-05-01';
UPDATE StockPrice SET close = 19.80 WHERE stock_id = 3 and date = '2020-05-02';
COMMIT;
```

Транзакция 2:

```
START TRANSACTION;
UPDATE StockPrice SET high = 20.12 WHERE stock_id = 3 and date = '2020-05-02';
UPDATE StockPrice SET high = 47.20 WHERE stock_id = 4 and date = '2020-05-01';
COMMIT;
```

Каждая транзакция будет выполнять свой первый запрос и обновлять строку данных, блокируя ее в индексе первичного ключа и любом дополнительном уникальном индексе, частью которого она является в процессе. Затем каждая транзакция попытается обновить свою вторую строку, но обнаружит, что та уже заблокирована. В итоге каждая транзакция будет до бесконечности ожидать окончания другой, пока не произойдет вмешательство извне, которое снимет взаимоблокировку. В главе 7 мы рассмотрим, как индексирование может повысить или снизить производительность ваших запросов по мере развития схемы.

Для борьбы с этой проблемой в системах баз данных реализованы различные формы обнаружения взаимоблокировок и тайм-аутов. Более сложные системы, такие как подсистема хранения InnoDB, заметят циклические зависимости и немедленно вернут ошибку. Это очень хорошо, иначе взаимоблокировки будут проявляться как очень медленные запросы. Другие системы откатывают транзакцию по истечении тайм-аута, что не всегда хорошо. Способ, которым InnoDB в настоящее время обрабатывает взаимоблокировки, заключается в откате транзакции, которая захватила наименьшее количество монопольных блокировок строк (приблизительный показатель, указывающий для какой транзакции будет проще всего выполнить откат).

Поведение и порядок блокировки зависят от подсистемы хранения, поэтому некоторые подсистемы могут заблокировать определенную последовательность

операторов, а другие этого не сделают. Взаимоблокировки имеют двойную природу: некоторые из них неизбежны из-за реальных конфликтов данных, а некоторые вызваны схемой работы конкретной подсистемы хранения¹.

Возникшие взаимоблокировки не получится преодолеть без частичного или полного отката одной из транзакций. Это суровая проза жизни в транзакционных системах, и ваши приложения должны быть рассчитаны на их обработку. Многие приложения могут просто попытаться повторить свои транзакции с самого начала, и если они не столкнутся еще с одним тупиком, то должны быть успешными.

Ведение журнала транзакций

Ведение журнала транзакций помогает сделать их более эффективными. Вместо обновления таблиц на диске каждый раз, когда происходит изменение, подсистема хранения может изменить свою копию данных в памяти. Это очень быстро. Затем подсистема хранения может сделать запись об изменении в журнал транзакций, который находится на диске и поэтому долговечен. Это довольно быстрая операция, поскольку добавление событий в журнал включает последовательный ввод/вывод в одной небольшой области диска, а не произвольный ввод/вывод во многих местах. Затем через какое-то время процесс может обновить таблицу на диске. Таким образом, большинство подсистем хранения, использующих этот метод, известный как *упреждающая запись в журнал*, в итоге дважды записывают изменения на диск.

Если сбой произошел после того, как обновление было записано в журнал транзакций, но до внесения изменений в сами данные, подсистема хранения все еще может восстановить изменения после перезапуска. Метод восстановления зависит от подсистемы хранения.

Транзакции в MySQL

Подсистемы хранения — это программное обеспечение, которое управляет тем, как данные будут храниться на диске и извлекаться с него. В то время как MySQL традиционно предлагает несколько подсистем хранения, поддерживающих транзакции, InnoDB является золотым стандартом и рекомендуемой для использования подсистемой хранения. Описанные здесь примитивы транзакций будут основаны на транзакциях в подсистеме InnoDB.

¹ Как вы увидите далее в главе, одни подсистемы хранения данных блокируют таблицы целиком, а другие реализуют более сложную блокировку на уровне строк. Вся эта логика по большей части находится на уровне подсистемы хранения данных.

AUTOCOMMIT

По умолчанию одиночная инструкция `INSERT`, `UPDATE` или `DELETE` неявно включается в транзакцию и немедленно подтверждается. Она известна как режим `AUTOCOMMIT`. Отключив этот режим, вы можете выполнить серию операторов внутри транзакции и в конце — `COMMIT` или `ROLLBACK`.

Вы можете включить или отключить переменную `AUTOCOMMIT` для текущего соединения с помощью команды `SET`. Значения `1` и `ON` эквивалентны, как и `0` и `OFF`. Когда вы работаете с `AUTOCOMMIT=0`, то всегда находитесь в транзакции, пока не выполните `COMMIT` или `ROLLBACK`. После этого MySQL немедленно начинает новую транзакцию. Кроме того, при включенном `AUTOCOMMIT` вы можете начать транзакцию с несколькими инструкциями, используя ключевое слово `BEGIN` или `START TRANSACTION`. Изменение значения `AUTOCOMMIT` не влияет на нетранзакционные таблицы, которые не имеют представления о подтверждении или отмене изменений.

Некоторые команды, выдаваемые во время открытой транзакции, заставляют MySQL подтвердить транзакцию до их выполнения. Обычно это команды языка определения данных (Data Definition Language, DDL), которые вносят существенные изменения, такие как `ALTER TABLE`, но `LOCK TABLES` и некоторые другие операторы также обладают этим свойством. Проверьте документацию вашей версии для получения полного списка команд, которые автоматически фиксируют транзакцию.

MySQL позволяет вам установить уровень изолированности с помощью команды `SET TRANSACTION ISOLATION LEVEL`, которая вступает в силу при запуске следующей транзакции. Вы можете настроить уровень изолированности для всего сервера в файле конфигурации или только для вашей сессии:

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

Предпочтительно устанавливать уровень изолированности, который вы чаще всего используете, на уровне сервера и изменять ее только в явных случаях. MySQL распознает все четыре стандартных уровня изолированности ANSI, и *InnoDB* поддерживает их все.

Применение нескольких подсистем хранения данных в транзакциях

MySQL не управляет транзакциями на уровне сервера. Вместо этого базовые подсистемы хранения сами реализуют транзакции. Это означает, что вы не можете надежно сочетать разные подсистемы в одной транзакции.

Если вы используете транзакционные и нетранзакционные таблицы (например, таблицы *InnoDB* и *MyISAM*) в одной транзакции, она будет работать правильно, когда все в порядке. Однако, если потребуется выполнить откат, изменения

в нетранзакционной таблице нельзя будет отменить. Это оставляет базу данных в несогласованном состоянии, из которого ее может быть трудно восстановить, и ставит под сомнение идею транзакций в целом. Вот почему очень важно выбрать правильную подсистему хранения для каждой таблицы и любой ценой избегать смешивания подсистем хранения в логике вашего приложения.

MySQL обычно не станет предупреждать вас или вызывать ошибки, если вы выполняете транзакционные операции над нетранзакционной таблицей. Иногда откат транзакции будет генерировать предупреждение «Не удалось откатить некоторые нетранзакционные измененные таблицы», но в большинстве случаев не будет никаких указаний на то, что вы работаете с нетранзакционными таблицами.



Лучше всего не смешивать подсистемы хранения в своем приложении. Неудачные транзакции могут привести к противоречивым результатам, поскольку некоторые части могут откатиться, а другие — нет.

Неявная и явная блокировка

InnoDB использует двухфазный протокол блокировки. Она может устанавливать блокировки в любой момент транзакции, но не снимает их до тех пор, пока не будет выполнена команда **COMMIT** или **ROLLBACK**. Все блокировки снимаются одновременно. Все механизмы блокировки, описанные ранее, являются неявными. *InnoDB* автоматически обрабатывает блокировки в соответствии с вашим уровнем изоляции.

Однако *InnoDB* также поддерживает явную блокировку, чего нет в стандарте SQL^{1, 2}.

```
SELECT ... FOR SHARE
SELECT ... FOR UPDATE
```

MySQL также поддерживает команды **LOCK TABLES** и **UNLOCK TABLES**, которые реализованы на сервере, а не в подсистемах хранения. Если вам нужны транзакции, используйте транзакционную подсистему хранения. Команда **LOCK TABLES** не нужна, поскольку *InnoDB* поддерживает блокировку на уровне строк.



Взаимодействие между **LOCK TABLES** и транзакциями сложное, и в некоторых версиях сервера наблюдается непредвиденное поведение. Поэтому мы рекомендуем никогда не применять **LOCK TABLES**, если вы не находитесь в транзакции и **AUTOCOMMIT** не отключен, независимо от того, какой подсистемой хранения пользуетесь.

¹ Этими блокирующими подсказками часто злоупотребляют, и их обычно следует избегать.

² **SELECT ... FOR SHARE** — это функция MySQL 8.0, которая заменяет **SELECT...LOCK IN SHARE MODE** предыдущих версий.

Управление конкурентным доступом с помощью многоверсионности

Большинство транзакционных подсистем хранения в MySQL не используют простой механизм построчной блокировки. Вместо этого они задействуют блокировку на уровне строк в сочетании с методом повышения уровня конкурентности, известным как *управление конкурентным доступом с помощью многоверсионности* (multiversion concurrency control, MVCC). MVCC не уникален для MySQL: Oracle, PostgreSQL и некоторые другие системы баз данных также используют его, хотя есть существенные различия, поскольку не существует стандарта работы MVCC.

Вы можете воспринимать MVCC как разновидность блокировки на уровне строк — во многих случаях вообще нет необходимости блокировки и можно существенно снизить накладные расходы. В зависимости от способа реализации MVCC может разрешать чтение без блокировки, блокируя только необходимые строки во время операций записи.

MVCC сохраняет моментальные снимки данных в том виде, в каком они существовали в какой-то момент времени. Это означает, что транзакции могут видеть согласованное представление данных независимо от того, как долго они выполняются. Это также означает, что разные транзакции могут одновременно видеть разные данные в одних и тех же таблицах! Если вы никогда не сталкивались с этим раньше, это может сбивать вас с толку, но по мере знакомства с данной технологией все становится понятнее.

Каждая подсистема хранения реализует MVCC по-разному. Некоторые варианты включают оптимистическое и пессимистическое управление конкурентным доступом. Мы проиллюстрируем один из способов работы MVCC, объяснив поведение InnoDB¹ в виде диаграммы последовательности на рис. 1.2.

InnoDB реализует MVCC, назначая идентификатор транзакции для каждой начавшейся транзакции. Он назначается при первом чтении транзакцией каких-либо данных. Когда запись изменяется в этой транзакции, запись отмены (undo record), которая объясняет, как отменить это изменение, вносится в журнал отмены (undo log), а указатель отката транзакции указывает на эту запись журнала отмены. Таким образом транзакция может найти способ отката, если это необходимо.

Когда другой сеанс считывает запись ключа кластерного индекса, InnoDB сравнивает идентификатор транзакции записи с представлением чтения этого сеанса. Если запись в ее текущем состоянии не должна быть видимой (транзакция, изменившая ее, еще не зафиксирована), записи журнала отмены считываются

¹ Рекомендуем прочитать эту запись в блоге Джереми Коула, чтобы получить более глубокое представление о структуре записей в InnoDB (<https://oreil.ly/jbljq>).

и применяются до тех пор, пока сеанс не достигнет идентификатора транзакции, который должен быть видимым. Этот процесс может заикливиться до записи отмены, которая полностью удаляет эту строку, сигнализируя представлению чтения, что строки не существует. Записи в транзакции удаляются с помощью установки бита «удалено» в информационных флагах записи. Это действие также помечается в журнале отмены как «удалить метку удаления».

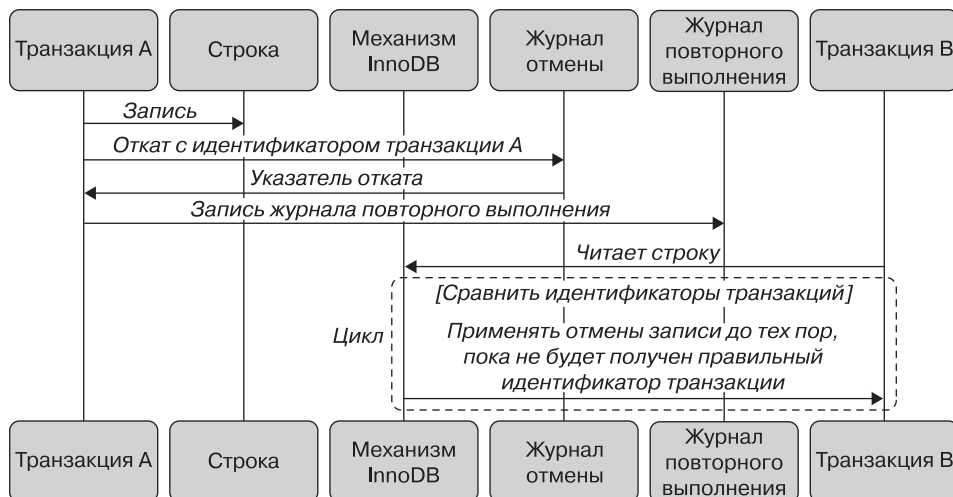


Рис. 1.2. Диаграмма последовательности обработки нескольких версий строки в разных транзакциях

Стоит отметить, что все записи, сделанные в журнале отмены, заносятся также в журнал повторов, потому что записи в журнале отмены — это часть процесса восстановления после сбоя сервера и они являются транзакционными¹. Размер журналов повторов и отмены играет важную роль в том, как выполняются транзакции с высокой степенью конкурентности. Мы рассмотрим их конфигурацию более подробно в главе 5.

Результатом этого дополнительного учета является то, что большинство запросов на чтение никогда не получают блокировки. Они просто считывают данные так быстро, как только могут, выбирая лишь строки, соответствующие указанным критериям. Недостатки подхода заключаются в том, что подсистема хранения должна хранить больше данных с каждой строкой, выполнять больше работы при проверке строк и реализовывать дополнительные вспомогательные операции.

¹ Для получения более подробной информации о том, как InnoDB обрабатывает несколько версий своих записей, прочтите этот пост в блоге Джереми Коула (Jeremy Cole): <https://oreil.ly/exaaL>.

MVCC работает только с уровнями изоляции REPEATABLE READ и READ COMMITTED. READ UNCOMMITTED несовместим с MVCC¹, поскольку запросы не считывают версию строки, подходящую для их версии транзакции, — они читают новейшую версию, несмотря ни на что. SERIALIZABLE несовместим с MVCC, потому что чтение блокирует каждую возвращаемую строку.

Репликация

MySQL разработана для приема записей на одном узле в любой момент времени. Это дает преимущества при управлении согласованностью данных, но требует компромиссов, когда вам нужны данные, записанные на нескольких серверах или в нескольких местах. MySQL предлагает собственный способ рассылки записей, которые один узел передает другим узлам. Этот способ называется репликацией. В MySQL узел — источник данных использует поток для каждой реплики, зарегистрированный как клиент репликации, который просыпается, когда происходит запись, отправляя новые данные. На рис. 1.3 показан простой пример этой системы, которую обычно называют деревом топологии нескольких серверов MySQL в структуре источника и реплик.

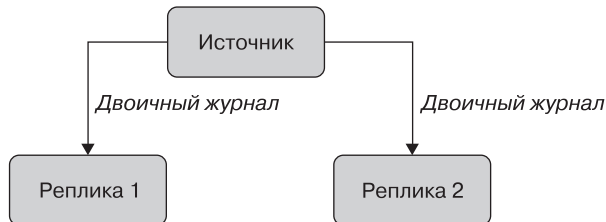


Рис. 1.3. Упрощенное представление топологии репликации сервера MySQL

Для любых данных, с которыми вы работаете в производственной среде, необходимо использовать репликацию и иметь как минимум еще три реплики, в идеале размещенные в разных местах (в облачных средах, известных как регионы) для планирования аварийного восстановления.

С годами репликация в MySQL стала более сложной. Глобальные идентификаторы транзакций, репликация с несколькими источниками, параллельная репликация на репликах и полусинхронная репликация — вот некоторые из основных обновлений. Мы подробно рассмотрим репликацию в главе 9.

¹ Не существует формального стандарта, определяющего MVCC, поэтому разные механизмы и базы данных реализуют его очень по-разному и никто не может сказать, что какой-то из них неправильный.

Структура файлов данных

В версии 8.0 MySQL метаданные таблиц были преобразованы в словарь данных, содержащийся в таблицах, размещенных в табличном пространстве с именем `mysql.ibd`. Это позволяет информации о структуре таблицы поддерживать транзакции и изменения определения атомарных данных. Вместо того чтобы полагаться только на `information_schema` для извлечения определения таблицы и метаданных во время операций, нам предлагается кэш объектов словаря, который представляет собой структуру в памяти для сохранения последних использованных объектов (применяет стратегию LRU): определений разделов, таблиц, хранимых процедур и функций, кодовой страницы и информации о сопоставлении. Это серьезное изменение в том, как сервер обращается к метаданным о таблицах, сокращает количество операций ввода/вывода и повышает эффективность, особенно если подмножество таблиц является наиболее активным и, следовательно, чаще всего находится в кэше. Файлы `.ibd` и `.frm` заменены файлами `.sdi` с сериализованной словарной информацией для каждой таблицы.

Подсистема хранения InnoDB

InnoDB — это транзакционная подсистема хранения по умолчанию для MySQL, а также самая важная и в целом широко используемая подсистема хранения. Она была создана для обработки многих краткосрочных транзакций, которые успешно выполняются намного чаще, чем откатываются. Ее производительность и автоматическое восстановление после сбоев делают ее популярной и для нетранзакционных хранилищ. Если вы хотите изучить подсистемы хранения, стоит потратить время на углубленное рассмотрение InnoDB, чтобы узнать о ней как можно больше, а не изучать все подсистемы хранения в равной степени.



Лучше всего использовать InnoDB в качестве подсистемы хранения по умолчанию для любого приложения. MySQL упростила эту задачу, сделав InnoDB подсистемой хранения по умолчанию несколько основных версий назад.

InnoDB — это подсистема хранения общего назначения MySQL по умолчанию. Она сохраняет данные в одном или нескольких файлах данных, которые называются *табличным пространством* (tablespace). Табличное пространство — это, по сути, черный ящик, объектами которого управляет сама InnoDB.

InnoDB использует MVCC для достижения высокого уровня конкурентности и реализует все четыре стандартных уровня изолированности SQL. По умолчанию

применяется уровень REPEATABLE READ, и у него есть стратегия *блокировки следующего ключа*, которая предотвращает возможность фантомного чтения на этом уровне безопасности: вдобавок к блокировке строк, затронутых в запросе, InnoDB блокирует также пропуски в структуре индекса, предотвращая вставку фантомных строк.

Таблицы InnoDB построены на *кластеризованных индексах*, которые мы подробно рассмотрим в главе 8, когда будем обсуждать дизайн схемы. Структуры индексов InnoDB сильно отличаются от структур индексов большинства других подсистем хранения MySQL. В результате они обеспечивают очень быстрый поиск по первичному ключу. Однако *вторичные индексы* (индексы, не являющиеся первичным ключом) содержат столбцы первичного ключа, поэтому, если ваш первичный ключ большой, другие индексы также будут большими. Вы должны стремиться к небольшому первичному ключу, если у вас будет много индексов в таблице.

InnoDB поддерживает множество внутренних оптимизаций. К ним относятся упреждающее чтение для предварительной выборки данных с диска, адаптивный хеш-индекс, который автоматически создает хеш-индексы в памяти для очень быстрого поиска, и буфер вставок для ускорения операций вставки. Мы рассмотрим эти вопросы в главе 4 этой книги.

Поведение InnoDB очень сложное, и мы настоятельно рекомендуем прочитать раздел «Модель блокировки и транзакций InnoDB» в руководстве по MySQL, если вы используете InnoDB. Из-за наличия архитектуры MVCC есть много тонкостей, о которых вы должны узнать, прежде чем создавать приложение с InnoDB. Работа с подсистемой хранения, поддерживающей согласованные представления данных для всех пользователей, даже когда некоторые из них изменяют данные, может быть сложной.

В качестве транзакционной подсистемы хранения InnoDB поддерживает по-настоящему горячее онлайн-резервное копирование с помощью различных механизмов, включая MySQL Enterprise Backup, запатентованную Oracle, и Percona XtraBackup с открытым исходным кодом. Мы подробно рассмотрим резервное копирование и восстановление в главе 10.

Начиная с MySQL 5.6, InnoDB предоставила онлайн-DDL, который сначала имел ограниченные сценарии использования, в дальнейшем расширенные в выпусках 5.7 и 8.0. Изменения схемы онлайн позволяют вносить определенные изменения в таблицу без полной ее блокировки и без применения внешних инструментов, что значительно улучшает работу таблиц MySQL InnoDB. В главе 6 мы рассмотрим варианты онлайн-изменения схемы с помощью как собственных, так и внешних инструментов.

Поддержка JSON-документов

Тип JSON, впервые представленный в InnoDB как часть версии 5.7, включал автоматическую проверку JSON-документов и оптимизированное хранилище, обеспечивающее быстрый доступ для чтения, что являлось значительным улучшением по сравнению со старым хранилищем больших бинарных объектов (BLOB), которое использовалось для хранения JSON-документов прежде. Наряду с поддержкой нового типа данных InnoDB предоставила также функции SQL для поддержки расширенных операций с JSON-документами. В дальнейшем в MySQL 8.0.7 была добавлена возможность определять многозначные индексы для JSON-массивов. Эта функция может обеспечить способ еще больше ускорить выполнение запросов для чтения JSON-типов с помощью сопоставления общих шаблонов доступа с функциями, которые отображают значения JSON-документа. Мы рассмотрим применение типа данных JSON и его влияние на производительность в подразделе «JSON-данные» в главе 6.

Изменения словаря данных

Еще одним важным изменением в MySQL 8.0 является удаление основанного на файлах хранилища метаданных таблиц и переход к словарю данных с использованием табличного пространства InnoDB. Это изменение обеспечивает все транзакционные преимущества InnoDB по восстановлению после сбоев в таких операциях, как изменение таблиц.

Это значительно улучшает управление определениями данных в MySQL, а также требует серьезных изменений в работе сервера MySQL. В частности, процессы резервного копирования, которые раньше полагались на файлы метаданных таблиц, теперь должны запрашивать новый словарь данных для извлечения определений таблиц.

Атомарный DDL

Наконец, в MySQL 8.0 представлены атомарные изменения определения данных. Это означает, что операторы определения данных теперь могут либо полностью завершиться успешно, либо полностью откатиться. Это становится возможным благодаря созданию специального журнала отмены (undo log) и повторного выполнения (redo log) для DDL, который InnoDB использует для отслеживания изменений, — еще одно место, где проверенный дизайн InnoDB был распространен на работу сервера MySQL.

Резюме

MySQL имеет многоуровневую архитектуру, на верхнем уровне которой находятся серверные службы и функции выполнения запросов, а под ними — подсистемы хранения. Существует множество различных подключаемых API, а наиболее важным является API подсистемы хранения. Если вы понимаете, что MySQL выполняет запросы, передавая строки туда и обратно через API подсистемы хранения, то вы усвоили основы архитектуры сервера.

В нескольких последних крупных выпусках MySQL InnoDB была выбрана в качестве основного направления разработки, на нее даже перенесли внутренний учет метаданных таблиц, аутентификацию и авторизацию после многих лет работы в MyISAM. Увеличение инвестиций Oracle в подсистему InnoDB привело к значительным улучшениям, таким как атомарные DDL, более надежные онлайн-DDL, повышенная устойчивость к сбоям и улучшенная работоспособность для развертываний, ориентированных на безопасность.

InnoDB — это подсистема хранения по умолчанию, которая должна охватывать почти все сценарии использования. Поэтому в следующих главах, когда речь идет о функциях, производительности и ограничениях, основное внимание уделяется подсистеме хранения InnoDB, и лишь изредка мы будем касаться какой-либо другой подсистемы хранения.

Мониторинг в мире проектирования надежности

Системы мониторинга — это обширная тема, которая за последние несколько лет получила широкое распространение благодаря основополагающей работе *Site Reliability Engineering: How Google Runs Production Systems*¹ (O'Reilly) и ее продолжению *The Site Reliability Workbook: Practical Ways to Implement SRE*² (O'Reilly). С момента выхода этих двух книг проектирование надежности и безотказности сайтов (SRE) стало популярной тенденцией в открытых списках вакансий. Некоторые компании зашли так далеко, что переименовали существующие должности в некие разновидности «инженера по надежности».

Проектирование надежности сайтов изменило то, что команды думают о работе по эксплуатации систем. Это привело к формулированию набора принципов, которые позволяют нам легко ответить на следующие вопросы.

- Обеспечиваем ли мы приемлемое качество обслуживания клиентов?
- Должны ли мы сосредоточиться на работе по обеспечению надежности и устойчивости?
- Как мы уравниваем новые функции с высокой нагрузкой?

В этой главе предполагается, что читатель понимает, что представляют собой эти принципы. Если вы не читали ни одну из упомянутых книг, мы

¹ Бейер Б., Джоунс К., Петофф Дж., Мерфи Р. *Site Reliability Engineering*. Надежность и безотказность как в Google. — Питер, 2019.

² Бейер Б., Рензин Д., Кавахара К., Торн С. *Site Reliability Workbook*: практическое применение. — Питер, 2021.

рекомендуем следующие главы из *The Site Reliability Workbook* в качестве ускоренного курса.

- Глава 1 предлагает углубленное рассмотрение философии перехода к управлению производительностью на основе соглашений об уровне обслуживания (SLA) в производственной среде.
- В главе 2 рассказывается о том, как реализовать цели уровня обслуживания (SLO).
- Глава 5 посвящена оповещению о SLO.

Некоторые могут возразить, что реализация SRE, строго говоря, не является частью архитектуры высокопроизводительной MySQL, но мы с этим не согласны. В своей книге *Accelerate*¹ доктор Николь Форсгрэн говорит: «Наша оценка должна быть сосредоточена на результатах, а не на выводах». Ключевым аспектом эффективного управления MySQL является хороший мониторинг работоспособности ваших баз данных. Традиционный мониторинг — это довольно хорошо проторенный путь. Поскольку SRE — это новая область, не очень понятно, как реализовать принципы SRE в отношении MySQL. По мере того как принципы SRE продолжают получать признание, традиционная роль администратора баз данных будет изменяться и включать требование, что администраторы баз данных не должны забывать о мониторинге своих систем.

Влияние проектирования надежности на DBA группы

В течение многих лет мониторинг производительности базы данных основывался на глубоком анализе производительности отдельного сервера. Это по-прежнему имеет большое значение, но, как правило, больше связано с измерениями, реагирующими на конкретную проблему, такими как профилирование сервера, который работает плохо. Это была стандартная рабочая процедура во времена групп администраторов баз данных, когда никто другой не имел права знать, как работает база данных.

Прочтите введение Google в область проектирования надежности. Роль администратора базы данных стала более сложной, и он превратился в инженера по надежности сайта (SRE) или инженера по надежности базы данных (DBRE). Команды должны были оптимизировать свое время. Уровни обслуживания

¹ Forsgren N. *Accelerate: The Science of Lean Software and DevOps*. — IT Revolution Press, 2018. <https://oreil.ly/Bfvda>.

помогают вам определить, когда клиенты недовольны, и позволяют лучше распределить свое время между решением таких вопросов, как проблемы с производительностью и масштабированием, и работой над внутренними инструментами. Обсудим различные способы мониторинга MySQL, необходимые для обеспечения успешного обслуживания клиентов.

Определение целей уровня обслуживания

Прежде чем перейти к вопросу о том, как измерить, довольны ли клиенты производительностью ваших кластеров баз данных, вы должны сначала узнать, каковы ваши цели, и определить общий язык для их описания. Вот несколько вопросов, которые в вашей организации могут послужить началом разговора для определения указанных целей.

- Какие показатели подходят для измерения успеха?
- Какие значения этих показателей приемлемы для клиентов и потребностей нашего бизнеса?
- Какой момент мы будем считать деградировавшим состоянием?
- Когда мы полностью несостоятельны и нуждаемся в максимально быстром восстановлении?

Существуют сценарии с очевидными ответами на эти вопросы, например: исходная база данных не работает, мы не делаем никаких записей, поэтому бизнес останавливается. Некоторые из них менее очевидны, например: периодическая задача иногда перегружает весь дисковый ввод/вывод базы данных, и внезапно все остальные операции замедляются. Наличие в организации общего понимания того, что мы измеряем и почему, может помочь при обсуждении приоритетов. Достижение общего понимания в ходе непрерывных обсуждений в рамках всей организации помогает понять, можете ли вы тратить инженерные усилия на новые функции, или нужно больше инвестировать в повышение производительности или стабильности. В практике SRE эти обсуждения удовлетворенности клиентов помогают команде понять, что полезно для бизнеса с точки зрения индикаторов уровня обслуживания (SLI), целевых показателей качества обслуживания (SLO) и соглашений об уровне обслуживания (SLA). Начнем с определения значения этих терминов.

- *Показатели уровня качества обслуживания (service level indicators, SLI).* Если просто, то SLI отвечают на вопрос «Как мне измерить, довольны ли мои клиенты?». Ответ подтверждает, что система находится в рабочем состоянии,

с точки зрения пользователей. SLI могут быть индикаторами бизнес-уровня, такими как «время отклика для клиентского API» или более фундаментальное «обслуживание доступно». Вы можете понять, что вам нужны разные индикаторы или показатели, в зависимости от контекста данных и их отношения к продукту.

- *Целевые показатели уровня качества обслуживания (service level objectives, SLO).* SLO отвечают на вопрос «Какой минимум допустим для SLI, чтобы мои клиенты были довольны?». SLO устанавливают целевой диапазон, в котором должны находиться SLI, соответствующие работоспособному сервису. Если вы считаете, что время безотказной работы — это SLI, то количество девяток доступности, которое хотите получить для сервиса, работающего нормально в течение заданного промежутка времени, — это SLO. SLO должны быть определены как значение за некий период времени, чтобы гарантировать, что все согласны с тем, что означают SLO. SLI вместе с SLO формулируют базовое уравнение для определения того, довольны ли ваши клиенты.
- *Соглашения об уровне качества обслуживания (service level agreements, SLA).* SLA дают ответ на вопрос «На какие SLO, имеющие значение, я готов согласиться?». SLA — это SLO, которые были включены в соглашение с одним или несколькими клиентами компании (плательщиками, а не внутренними заинтересованными сторонами), предусматривающее финансовые или иные санкции в случае несоблюдения этих SLA. Важно отметить, какие соглашения об уровне обслуживания необязательны.

Мы не будем подробно рассматривать SLA в этой главе, так как они, как правило, требуют скорее делового обсуждения, чем инженерного. Решение такого рода в основном зависит от того, какие продажи ожидает получить компания, если она пообещает SLA в контрактах, и стоит ли это риска для доходов в случае нарушения SLA. Будем надеяться: решение будет основано на том, что мы здесь рассказываем о выборе как SLI, так и соответствующих SLO.

Определение SLI, SLO и SLA влияет не только на состояние бизнеса, но и на планирование внутри инженерных групп. Если команда не выполняет свои согласованные SLO, она не должна приступать к работе над новой функциональностью. То же самое верно и для команд разработки баз данных. Если один из потенциальных SLO, которые мы обсуждаем в этой главе, не выполняется, это должно стимулировать обсуждение того, почему так происходит. Когда вы вооружитесь данными, которые позволят объяснить, почему качество обслуживания клиентов неоптимально, то сможете более содержательно говорить о приоритетах команды.

Что нужно, чтобы клиенты были довольны

После выбора набора показателей в качестве SLI может возникнуть соблазн достичь абсолютно всех целей. Однако вы должны бороться с этим желанием. Помните, что выбор показателей и целей обеспечит возможность в любое время оценить с помощью объективных данных, может ли ваша команда внедрять инновации с помощью новых функций, или существует риск падения стабильности ниже приемлемого уровня для клиентов и, следовательно, требуется больше внимания и ресурсов. Цель состоит в том, чтобы определить, что является *абсолютным минимумом*, которого вам нужно добиться, чтобы клиенты были довольны. Если клиент доволен загрузкой страниц за 2 с, нет необходимости устанавливать цель загрузки страниц за 750 мс. Это может привести к необоснованной нагрузке на инженерные группы.

Взяв, например, время безотказной работы в качестве показателя уровня качества обслуживания и целевых значений для него, мы можем заявить, что у нас не будет простоев. Но что это означает для реализации и отслеживания достижения целей? Получение трех девяток показателя доступности — немалый подвиг. Три девятки за целый год составляют немногим более 8 ч, то есть всего 10 мин в неделю. Чем больше девяток вы обещаете, тем сложнее этого добиться и тем больше времени придется потратить инженерной команде, чтобы выполнить обещание. В табл. 2.1 представлены полезные данные от Amazon Web Services, иллюстрирующие проблему с помощью простых чисел.

Таблица 2.1. Время доступности в девятках

Доступность, %	Время простоя в год	Время простоя в месяц	Время простоя в неделю	Время простоя в день
99,999	5 мин 15,36 с	26,28 с	6,06 с	0,14 с
99,995	26 мин 16,8 с	2 мин 11,4 с	30,30 с	4,32 с
99,990	52 мин 33,6 с	4 мин 22,8 с	1 мин 0,66 с	8,64 с
99,950	4 ч 22 мин 48 с	31 мин 54 с	5 мин 3 с	43,00 с
99,900	8 ч 45 мин 36 с	43 мин 53 с	10 мин 6 с	1 мин 26 с
99,500	43 ч 48 мин 36 с	3 ч 39 мин	32 мин 17 с	7 мин 12 с
99,250	65 ч 42 мин	5 ч 34 мин 30 с	1 ч 15 мин 48 с	10 мин 48 с
99,000	3 дня 15 ч 54 мин	7 ч 18 мин	1 ч 41 мин, 5 с	14 мин 24 с

Поскольку время разработки — ограниченный ресурс, вы должны быть внимательны и не стремиться к совершенству при выборе SLO. Не все функции вашего продукта требуют всех этих девяток, чтобы удовлетворить клиентов. Поэтому вы обнаружите, что по мере роста набора функций продукта у вас будут разные SLI и SLO в зависимости от влияния конкретной функции или дохода, который она обеспечивает. Это вполне ожидаемо и является признаком обдуманного процесса. При этом у вас есть критическая задача: определить, когда набор данных становится узким местом для разных профилей запросов разными заинтересованными сторонами, ставящим под угрозу производительность системы. Это также показывает необходимость поиска способа разделения различных потребностей заинтересованных сторон, позволяющего предоставить им разумные SLI и SLO.

Определение показателей и целей уровня качества обслуживания является эффективным способом поиска общего языка между командой развития продукта и разработчиками, что позволит находить компромисс между задачами «тратить время на разработку новых функций» и «тратить время на отказоустойчивость и устранение проблем». Они также помогают, основываясь на опыте клиентов, выбрать наиболее важные из списка задач, которые необходимо выполнить. Вы можете использовать SLI и SLO как ориентиры, чтобы говорить о приоритетах работ, которые в противном случае бывает трудно согласовать.

Что измерять

Представим себе компанию, продуктом которой является интернет-магазин. В результате увеличения количества онлайн-покупок компания получает значительно больше трафика, поэтому группе инфраструктуры ставится задача гарантировать, что база данных может справиться с возросшей нагрузкой. В этом разделе мы поговорим о том, что необходимо измерять нашей вымышленной инфраструктурной команде.

Определение SLI и SLO

Определение хороших SLI и соответствующих SLO требует краткого объяснения того, как обеспечить замечательный пользовательский опыт вашим клиентам. Мы не будем тратить массу времени на абстрактное объяснение того, как создавать осмысленные SLI и SLO¹.

В контексте MySQL это должно быть утверждение, определяющее три основных параметра: доступность, задержку и отсутствие критических ошибок.

¹ Настоятельно рекомендуем книгу: *Hidalgo A. Implementing Service Level Objectives.* — O'Reilly, 2020.

Для примера с интернет-магазином это означает, что страницы загружаются быстро, менее чем за несколько сотен миллисекунд, по крайней мере для 99,5 % случаев в течение месяца. Это также означает надежный процесс оформления заказа, в котором непредвиденные сбои допускаются только в 1 % случаев в течение данного календарного месяца. Обратите внимание на то, как определяются эти показатели и цели. Мы не устанавливаем 100 % как требование, потому что работаем в мире, где отказы неизбежны. Мы используем временной интервал, позволяющий команде тщательно распределить время работы между созданием новых функций и обеспечением надежности системы.

«Я рассчитываю на то, что 99,5 % моих запросов к базе данных будут обслуживаться менее чем за 2 мс без ошибок» — это одновременно и достаточные SLI с четкими SLO, и не тривиальные. Вы не можете определить все это одним показателем. Здесь одним предложением сформулировано ваше представление о том, как слой базы данных будет вести себя, чтобы обеспечить приемлемое качество обслуживания клиентов.

Итак, что может быть хорошими примерами показателей нашего интернет-магазина, на которых может основываться картина клиентского опыта? Начните с общих тестов, таких как загрузка страниц в производственной среде, с примерами скорости загрузки. Они полезны в качестве ясного сигнала о том, что «все в порядке». Но это только начало. Обсудим различные аспекты сигналов, которые нужно отслеживать для построения общей картины. По мере рассмотрения различных примеров мы свяжем их с нашим интернет-магазином, чтобы помочь вам понять, как различные показатели создают картину хорошего обслуживания клиентов. Вначале поговорим об отслеживании времени ответа на запрос.

Решения для мониторинга

Анализ запросов и отслеживание их задержки в контексте SLI и SLO должны фокусироваться на клиентском опыте. Это означает что вам необходимы инструменты, которые могут максимально быстро предупредить, когда время ответа на запрос становится больше, чем согласованный порог. Обсудим несколько способов, которыми вы можете воспользоваться для достижения такого уровня мониторинга.

Коммерческие инструменты

Рассмотрим пример, когда плата поставщику, чьим конкурентным преимуществом является конкретная задача профилирования производительности MySQL, может принести вашей организации ощутимую пользу. Такие инструменты, как программное обеспечение для управления производительностью

базы данных компании SolarWinds, могут здорово помочь автоматизировать профилирование производительности запросов и обеспечить его доступность для большого числа ваших инженеров.

Варианты инструментов с открытым исходным кодом

Хорошо зарекомендовавший себя вариант инструмента с открытым исходным кодом — Persona Monitoring and Management (PMM). Он работает в архитектуре «клиент/сервер». Вы устанавливаете на свои экземпляры базы данных клиент, который собирает данные и отправляет на серверную часть. На стороне сервера имеется набор панелей мониторинга, которые позволяют просматривать графики, относящиеся к производительности. Одним из основных преимуществ PMM является то, что организация информационных панелей основана на многолетнем опыте сообщества Persona в области мониторинга производительности MySQL. Это делает его отличным инструментом для отслеживания производительности MySQL инженерами, плохо знакомыми с MySQL.

Еще один способ проверки производительности, который вы можете выбрать, — отправить журнал медленных запросов вашей базы данных и выходные данные MySQL Performance Schema в централизованное место, где можете использовать известные инструменты, такие как *pt-query-digest* (часть пакета Persona Toolkit), для анализа журналов и получения дополнительной информации о том, на что экземпляры вашей базы данных тратят свое время. Хотя этот процесс эффективен, он может быть медленным и способен отрицательно сказаться на клиентах, если не применяется должным образом. В идеале вам желательно обнаруживать проблемы до того, как их заметят клиенты. Проверяя журналы после возникновения проблемы, вы рискуете подорвать доверие клиентов, так как потребуется много времени для анализа снижения производительности и изучения всех видов постфакт-артефактов при определении того, что произошло.

Наконец, для профилирования производительности MySQL может быть очень полезно использовать Performance Schema, о чем более подробно рассказывается в главе 3. Вы сможете задействовать Performance Schema для поиска узких мест, что позволит вашим экземплярам базы данных выполнять больше запросов при неизменных технических условиях, экономить на затратах на инфраструктуру или отвечать на вопрос «Почему это занимает так много времени?». Поскольку Performance Schema глубоко встроена в MySQL, она не является инструментом для определения исключительно того, выполняете ли вы свои обещания относительно надежности службы. Для оценки производительности на уровне качества обслуживания нам нужен новый взгляд на производительность.

ПРИМЕЧАНИЯ О ТЕСТИРОВАНИИ В ПРОИЗВОДСТВЕННОЙ СРЕДЕ

Мы часто слышим о тестировании в производственной среде, и это заставляет многих людей съеживаться. Реальность такова, что тестирование в производственной среде может иметь большое значение. Производственная среда — это место, где вы узнаете, как какое-либо изменение влияет на остальную часть системы (с учетом масштаба) с реальным клиентским трафиком. Это позволяет также увидеть влияние изменения на другие системы. Отвечая на главный вопрос «Довольны ли клиенты?», вы сможете увидеть следующее.

- Когда обратная связь от производственной среды быстрая и сильно привязана к изменению, откат изменения и повторная проверка конкретного изменения, которое внедрялось, становятся намного быстрее.
- Этот метод способствует более тесному сотрудничеству между функциональными группами и инженерами баз данных. Задача измерения производительности становится общей, когда все заинтересованные стороны согласны в отношении конкретных показателей, которые необходимо отслеживать, а также их допустимых значений.
- В случае снижения производительности системы усилия, затрачиваемые за пределами производственной среды для исследования того, что произошло, гораздо более специфичны, чем попытки создать набор эталонных тестов, который эмулирует общие сценарии выполнения кода. Время инженеров, затрачиваемое на отладку, используется гораздо более целенаправленно.

Теперь углубимся в дополнительные показатели, которые помогут вам лучше понять опыт клиентов вашего интернет-магазина. Вы должны думать о показателях, которые можете получить от MySQL, в терминах результатов, а не выводов. Мы также рассмотрим примеры вещей, которые вы не можете измерить только с помощью показателей MySQL.

Мониторинг доступности

Интернет-магазин, который периодически отключается, рискует тем, что доверие к нему покупателей будет подорвано. Вот почему так важна доступность как отдельный показатель и как часть вашего взгляда на качество обслуживания клиентов.

Доступность — это возможность безошибочно реагировать на запросы клиентов. Если сформулировать в стандартных терминах HTTP, это может быть ответ, который является явным успехом, например ответ с кодом 200, или успешное принятие запроса с обещанием завершить соответствующую работу асинхронно, например принятие ответа с кодом 202. Во времена монолитных систем с одним хостом доступность была простым показателем. Сейчас же большинство архитектур намного сложнее. Концепция доступности также превратилась в более

тонкое отражение отказа распределенных систем. При попытке превратить доступность в SLI и SLO для вашей архитектуры базы данных рассмотрите возможность обсуждения не только примеров, касающихся нашего интернет-магазина, но и дополнительных тонкостей, включая следующие.

- Когда речь идет о неизбежных катастрофических сбоях, какие функции не подлежат обсуждению, а какие приятно иметь? Например, могут ли покупатели продолжать использовать существующие корзины покупок и проверять их, но, возможно, не добавлять новые товары во время этого сбоя?
- Какие типы сбоев мы определяем как катастрофические? Например, сбой поиска по списку может не быть катастрофическим, а сбой операций оформления заказа — может.
- Как выглядит ухудшенная функциональность? Например, можем ли мы при необходимости загружать общие рекомендации вместо индивидуальных, основанных на истории прошлых покупок?
- Какое минимально возможное среднее время восстановления (mean time to recovery, MTTR) основных функций мы можем обещать с учетом набора вероятных сценариев отказа? Например, если в базе данных, на которой работает система проверки корзины покупок, произошел сбой записи, как быстро мы можем безопасно перейти на новый исходный узел.

Выбирая набор показателей для представления доступности, вы хотите определить ожидания со своей службой поддержки клиентов, исходя из того, что требовать обеспечить 100%-ное время безотказной работы неразумно в мире, *понимающем и принимающем* тот факт, что неудачи неизбежны и что основное внимание здесь должно уделяться обеспечению наилучшего возможного обслуживания клиентов.

Предпочтительный метод проверки доступности — со стороны клиента или удаленной конечной точки. Это можно сделать пассивно, если у вас есть доступ к журналам доступа клиента к базе данных. В явном виде это означает, что если ваше приложение представлено в виде PHP-кода и вы работаете под Apache, то нужен доступ к журналам Apache, чтобы определить, выдает ли PHP-код какие-либо ошибки при подключении к вашей базе данных. Вы также можете активно проверять доступность. Если ваша среда изолирована и нельзя получить доступ к журналам клиентов, рассмотрите возможность настройки удаленного кода, который действует в вашей базе данных, чтобы убедиться в ее доступности. Это может быть что-то простое, например запрос `SELECT 1`, который показывает, что MySQL получает и анализирует ваш запрос, но не обращается к уровню хранения. Или что-то более сложное, например чтение фактических данных из таблицы или выполнение записи и последующее ее

чтение для проверки успешности этого действия. Такого рода искусственная транзакция из какого-нибудь места в сети может дать представление о том, доступно ли ваше приложение.

Удаленная проверка доступности полезна для отслеживания цели доступности. Она не поможет вам получить информацию *до того*, как возникнет проблема. Одним из показателей MySQL, который можно использовать в качестве ведущего индикатора проблем с доступностью, является счетчик состояния MySQL `Threads_running`. Он отслеживает, сколько запросов в настоящее время выполняется на данном хосте базы данных. Если количество запущенных потоков быстро растет и нет никаких признаков замедления, значит, запросы завершаются недостаточно быстро и, следовательно, накапливаются и потребляют ресурсы. Увеличение этого показателя обычно приводит к тому, что узел базы данных вызывает либо полную блокировку процессора, либо интенсивную нагрузку на память, что может привести к завершению всего процесса MySQL операционной системой. Очевидно, что это серьезный сбой, если он происходит на исходном узле, и вам следует стремиться иметь предупреждающие индикаторы. Отправной точкой для мониторинга этого индикатора является проверка того, сколько ядер ЦП в наличии, и, *если* `Threads_running` превышает данное значение, это может быть признаком того, что ваш сервер находится в опасном состоянии. Вместе с этим можно отслеживать, как близко вы подходите к параметру `max_connections`, в качестве еще одной точки данных для проверки перегрузки для выполняемой работы.

Раздел «Настройки безопасности» в главе 5 дает представление о том, как можно установить тормоза для неуправляемых потоков MySQL.

Мониторинг задержки запросов

MySQL представил ряд давно требующихся улучшений для отслеживания того, сколько времени занимает выполнение запросов, и вы обязаны использовать свой стек мониторинга для отслеживания этих индикаторов по мере изменения кода вашего приложения. Однако это все еще не дает полной картины клиентского опыта, особенно с учетом того, как спроектирована архитектура современного программного обеспечения. В дополнение к отслеживанию внутренней задержки вам также необходимо представлять, как задержка влияет на ваши приложения и что происходит, когда она увеличивается. Это означает, что, помимо непосредственного отслеживания задержки запросов с сервера базы данных, вам будет полезно настроить клиентов так, чтобы они сообщали о времени завершения запроса и тем самым вы могли максимально точно оценить клиентский опыт. Для анализа этих выборочных индикаторов, получаемых от клиентов, особенно при увеличении размера вашей инфраструктуры, могут потребоваться специальные

инструменты — либо платные, такие как Datadog или SolarWinds Database Performance Monitor, либо с открытым исходным кодом, такие как PMM. Это область, где тесное сотрудничество вашей организации с разработчиками приложений имеет первостепенное значение. Вам необходимо знать, как команда разработчиков приложения измеряет выполнение запросов с точки зрения приложения, и обеспечить больше информации для резко отличающихся значений, используя инструменты трассировки, такие как Honeycomb или Lightstep.

Мониторинг ошибок

Нужно ли вам отслеживать каждую когда-либо возникшую ошибку и предупреждать о ней? Это зависит от ошибки.

Само наличие ошибок в работающей службе не является для клиента MySQL признаком того, что что-то определенно неисправно. В мире распределенных систем существует множество сценариев, в которых клиенты могут столкнуться с периодически появляющимися ошибками, которые во многих случаях устраняются простым повторением неудачного запроса. Тем не менее частота ошибок, которые возникают во множестве сервисов, обрабатывающих запросы к базе данных в вашей инфраструктуре, может быть важным индикатором назревающих проблем. Вот несколько примеров ошибок на стороне клиента, которые обычно могут быть просто шумом, но свидетельствуют о проблеме, если частота их появления увеличивается.

- *Время ожидания блокировки.* Сообщения ваших клиентов о резком увеличении частоты появления этой ошибки могут быть признаком нарастания конфликта блокировки строк на узле источника данных, когда транзакции повторяются и оказываются неудачными. Это может быть предвестником ошибки записи данных.
- *Прерванные соединения.* Сообщения клиентов о непредвиденном увеличении числа прерванных подключений могут быть индикатором проблем на любом уровне доступа между клиентами и экземплярами базы данных. Отсутствие отслеживания этого параметра может привести к большому количеству повторных попыток подключений на стороне клиента, которые потребляют избыточные ресурсы.

Единственная вещь, отслеживаемая сервером MySQL, которая может вам помочь, — это набор серверных переменных состояния с именем `Connection_errors_xxx`, где `xxx` — это различные типы ошибок соединения. Внезапное увеличение показаний любого из этих счетчиков может быть важным индикатором того, что что-то новое и необычное в настоящее время нарушено.

Есть ли ошибки, которые даже в единственном числе означают наличие проблемы, которую необходимо устранить? Да.

Например, получение сообщений о том, что экземпляр MySQL работает в режиме «только для чтения», говорит о проблеме, даже если эти ошибки случаются не очень часто. Это может означать, что вы только что повысили реплику до исходной, но она все еще работает в режиме «только для чтения» (вы запускаете реплики в режиме «только для чтения», не так ли?), то есть запись данных для вашего кластера не выполняется. Или это может означать, что на вашем уровне доступа возникла проблема с отправкой трафика записи на реплику. В любом из случаев это не является признаком периодически возникающей проблемы, решаемой с помощью повторной попытки.

Еще одна ошибка на стороне сервера, которая является признаком серьезной проблемы, — это либо сообщение «Слишком много подключений», либо сообщение на уровне операционной системы «Невозможно создать новый поток». Эти сообщения говорят о том, что прикладной уровень создал и оставил открытыми больше соединений, чем позволяют настройки сервера базы данных, либо в серверной переменной `max_connections`, либо в количестве потоков, которые разрешено открывать процессу MySQL. Эти ошибки немедленно транслируются в ваше приложение как ошибки 5xx и в зависимости от дизайна приложения могут повлиять на работу ваших клиентов.

Как видите, измерение производительности и выбор ошибок, на которые следует ориентироваться в ваших SLI, — это не только техническая, но и коммуникационная и социальная проблема, поэтому вы должны быть к этому готовы.

Проактивный мониторинг

Как мы уже говорили, мониторинг SLO направлен на выяснение того, довольны ли ваши клиенты. Это помогает вам сосредоточиться на улучшении пользовательского опыта в тех случаях, когда они недовольны, и на других задачах, таких как сокращение трудоемких работ, когда довольны. При этом упускается ключевая область: проактивный мониторинг.

Вернувшись к примеру с интернет-магазином и к тому, как мы представляем себе мониторинг опыта клиентов, мы сможем развить его. Представьте, что вы не сталкиваетесь с серьезными сбоями каких-то компонентов, но замечаете, что растет поток запросов в службу поддержки клиентов: сообщают о «медленности» или случайных ошибках, которые, кажется, исчезают сами по себе. Как вы отслеживаете подобное поведение? Это может оказаться очень сложной задачей, если у вас еще нет четкого представления о базовой производительности ряда показателей. Панели мониторинга и сценарии, которые вы используете для запуска оповещений по требованию, можно назвать *мониторингом устойчивого состояния*. Они сообщают вам, что с данной системой происходит что-то неожиданное, независимо от того, были ли изменения. Это важный инструмент для получения вами упреждающих индикаторов до того, как клиенты столкнутся с проблемой.

Баланс, который следует соблюдать при мониторинге, заключается в том, что сигнал всегда должен действительно предупреждать о проблеме. Оповещать об исчерпании дискового пространства для базы данных при ее заполнении на 100 % слишком поздно, поскольку сервис уже не работает, а оповещение о 80%-ном заполнении может быть слишком ранним или неэффективным, если скорость роста дискового пространства не так уж высока.

Поговорим о полезных индикаторах, которые вы можете отслеживать и которые не связаны напрямую с фактическим влиянием на клиента.

Рост использования дискового пространства

Рост занятого дискового пространства — это такой показатель, о котором вы можете не думать, пока он не станет проблемой. Когда это действительно становится проблемой, ее решение может занять много времени и повлиять на ваш бизнес. Определенно, лучше понимать, как вы отслеживаете этот параметр, иметь план по устранению проблемы и знать, какие пороговые значения показателя являются подходящими для предупреждения.

Существует ряд стратегий, которые можно использовать для мониторинга роста занятого дискового пространства. Разберем их от самых идеальных до абсолютно минимальных.

Если ваш инструментарий мониторинга позволяет отслеживать темпы роста использования дискового пространства, это может быть чрезвычайно полезным. Всегда существуют сценарии, когда доступное дисковое пространство может сгорать довольно быстро, что ставит под угрозу доступность вашей системы. Такие операции, как длительные транзакции с большими журналами отката или изменения таблиц, являются примерами того, почему вы можете слишком быстро заполнить диск. Во многих случаях чрезмерное журналирование или изменение шаблона вставки для данного набора данных остаются незамеченными до тех пор, пока база данных не исчерпает дисковое пространство. Только после этого срабатывают всевозможные оповещения.

Если отслеживание темпов роста заполнения дискового пространства невозможно (не все инструменты мониторинга предоставляют эту возможность), вы можете установить несколько пороговых значений, по достижении которых будет выдано предупреждение, — более низких, срабатывающих только в рабочее время, и более высоких, действующих в нерабочее время. Это позволяет команде получать предупреждение в рабочее время, до того, как ситуация станет достаточно серьезной, чтобы кто-то ее устранил.

Если вы не можете ни отслеживать темпы роста, ни определять несколько пороговых значений для одного и того же параметра, то вам необходимо определить по крайней мере одно пороговое значение используемого дискового пространства, при достижении которого вы вызываете дежурных инженеров. Этот порог должен быть достаточно низким, чтобы дать время для выполнения действий по освобождению места на диске, пока команда оценивает причины срабатывания и рассматривает долгосрочные меры по устранению последствий переполнения. Попробуйте оценить максимальную пропускную способность записи на ваш диск (Мбайт/с) и использовать ее, чтобы рассчитать, сколько времени потребуется для его заполнения при максимальной пропускной способности трафика. Это позволяет оценить, столько нужно времени, чтобы избежать события заполнения диска.

В главе 4 мы обсудим конфигурации операционной системы и аппаратного обеспечения, связанные с использованием дискового пространства в MySQL, и то, какие компромиссы следует учитывать, принимая решения в связи с ростом задействования дискового пространства. Следует ожидать, что в какой-то момент ваш бизнес разрастется настолько, что вы не сможете хранить все свои данные в одном кластере серверов. Даже если работаете в облачной среде, которая может расширять тома, вам все равно нужно планировать это, поэтому всегда следует определить, сколько свободного места должно оставаться на диске, что даст вам время без паники спланировать и расширить дисковое пространство до нужного размера.

Вывод здесь такой: нужно убедиться, что у вас есть какой-то индикатор роста заполнения дискового пространства, даже если вы думаете, что находитесь в начале процесса и пока в нем не нуждаетесь. Это одна из осей роста, которая почти всех застает врасплох.

Рост числа подключений

По мере роста вашего бизнеса прикладной уровень вашего приложения растет линейно. Вам понадобится больше экземпляров объектов для поддержки входа в систему, корзины покупок, обработки запросов или любого другого контекста продукта. Все эти дополнительные экземпляры объектов начинают открывать все больше и больше соединений с хостами вашей базы данных. На какое-то время вы можете притормозить этот рост, добавив реплики базы данных, задействуя репликацию в качестве средства масштабирования или даже используя уровни промежуточного программного обеспечения, такие как ProxySQL, чтобы отделить прирост количества внешних интерфейсов от нагрузки на подключения непосредственно к базе данных.

Когда трафик растёт, сервер базы данных может поддерживать конечный пул соединений, который настраивается параметром сервера `max_connections`. Как только общее количество подключений к серверу достигает этого максимума, база данных перестаёт разрешать новые подключения. Это распространённая причина инцидентов, когда вы больше не можете открывать новые подключения к базе данных, что, в свою очередь, приводит к увеличению количества ошибок для ваших пользователей.

Мониторинг роста числа подключений заключается в том, чтобы убедиться: ваши ресурсы не исчерпаны до такой степени, что это может поставить под угрозу доступность базы данных. Такой риск может возникнуть по двум причинам.

- Прикладной уровень приложения открывает множество соединений, которые он не использует, и создаёт риск исчерпания максимального количества соединений без реальной причины. Явный признак этой ситуации — большое количество подключений (`threads_connected`) при низком значении `threads_running`.
- Прикладной уровень приложения активно использует все большее количество подключений и рискует перегрузить базу данных. Вы можете заметить это состояние, увидев, что `threads_connected` и `threads_running` имеют высокие значения (сотни? тысячи?) и увеличиваются.

При настройке мониторинга количества подключений следует учитывать проценты, а не абсолютные числа. Процентное соотношение `threads_connected/max_connections` показывает, насколько близко рост числа узлов вашего приложения подводит вас к максимальному пулу соединений, который база данных может разрешить. Это поможет вам с ростом числа соединений отслеживать первый тип проблем из указанных ранее.

Отдельно вы должны отслеживать, насколько занят узел базы данных, что, как мы уже объясняли, отображается в виде значения `threads_running`, и предупреждать об этом. Как правило, если это значение превышает сотню потоков, вы начинаете замечать повышенную загрузку процессора и увеличенное использование памяти, что является общим признаком высокой нагрузки на хост базы данных. Это начинает мешать получать доступ к вашей базе данных, так как может привести к тому, что процесс MySQL будет остановлен операционной системой. Обычное быстрое решение — использование команды `kill process` или инструмента, который автоматизирует её применение, например `pt-kill`, тактически, чтобы уменьшить нагрузку, а затем выяснять, почему база данных пришла в такое состояние, с помощью анализа запросов, который мы описали ранее.



Штормы подключений — это ситуации в производственных системах, когда прикладной уровень воспринимает увеличение задержки запросов и отвечает открытием дополнительных подключений к уровню базы данных. Это может привести к росту нагрузки на базу данных, поскольку она обрабатывает большой поток новых подключений, отнимающий ресурсы, необходимые для выполнения запросов. Штормы подключений могут привести к внезапному уменьшению количества доступных подключений в `max_connections` и увеличить риск недоступности вашей базы данных.

Отставание репликации

MySQL имеет встроенную функцию репликации, которая отправляет данные с *сервера-источника* на один или несколько дополнительных серверов, называемых *репликами*. Задержка между записью данных в источник и их доступностью в репликах называется *отставанием репликации*. Если ваше приложение считывает данные из реплик, задержка может создать впечатление, что данные имеют несоответствия, поскольку вы отправляете операции чтения на реплики, которые еще не получили все изменения. В примере с социальной сетью пользователь может прокомментировать то, что опубликовал кто-то другой. Эти данные записываются в источник, а затем отсылаются на реплики. Если приложение отправляет запрос на сервер-реплику, когда пользователь пытается просмотреть свой ответ, из-за задержки реплика может еще не иметь данных. Это способно вызвать недоумение у пользователя, считающего, что его комментарий не был сохранен. В главе 9 мы более подробно рассмотрим стратегии борьбы с отставанием репликации.

Задержка — один из тех показателей, которые могут быть актуальными для SLI и спровоцировать инциденты. Если отставание репликации становится продолжительным, это указывает на необходимость дополнительных архитектурных изменений. В долгосрочной перспективе, даже если вы никогда не столкнетесь с отставанием репликации, влияющим на качество обслуживания клиентов, это все равно будет признаком того, что при текущей конфигурации по крайней мере периодически объем операций записи с исходных узлов превосходит возможности записи реплик. Это может быть индикатором и ранним предупреждением о проблемах с возможностью записи у вас. Если к нему прислушаться, можно предотвратить полномасштабные инциденты в будущем.



Будьте осторожны, предупреждая кого-либо об отставании репликации. Немедленное действенное исправление ситуации не всегда возможно. Аналогично, если вы не читаете реплики, подумайте, насколько агрессивно ваша система мониторинга предупреждает кого-либо об этом состоянии. Оповещения, которые кто-то получает, особенно в нерабочее время, всегда должны быть действенными.

Отставание репликации — один из тех показателей, которые могут повлиять как на непосредственные, так и на тактические решения, а отслеживание ее тенденций в долгосрочной перспективе может помочь вам избежать хлопот, связанных с более масштабным воздействием на бизнес, и опережать кривую роста.

Использование ввода/вывода

Инженер базы данных всегда стремится выполнять как можно больше обработки в памяти, потому что это быстрее. Бесспорно, это верно, но мы также знаем, что не можем поступать так в 100 % случаев, поскольку это означало бы, что данные полностью помещаются в памяти и масштабирование еще не стало задачей, на которую нужно тратить усилия.

По мере того как инфраструктура базы данных масштабируется и данные больше не помещаются в памяти, вы начинаете понимать, что лучше бы не считывали столько данных с диска, чтобы запросы зависали в ожидании своей очереди для драгоценных циклов ввода/вывода. Это остается верным и в нашу эпоху, когда почти все работает на твердотельных накопителях. По мере роста объема данных и необходимости сканирования все большего их количества для выполнения запросов вы обнаружите, что ожидание ввода/вывода может стать узким местом для роста трафика.

Мониторинг операций дискового ввода/вывода поможет вам предотвратить снижение производительности до того, как это станет проблемой для клиентов. Есть несколько вещей, которые вы можете отслеживать, чтобы добиться этого. Такие инструменты, как `iostat`, могут помочь контролировать ожидание ввода/вывода. Вы хотите отслеживать и предупреждать, если на вашем сервере базы данных много потоков, находящихся в `IOWait`, что указывает на то, что они находятся в очереди, ожидая освобождения некоторых дисковых ресурсов. Вы обнаружите это, отслеживая `IOWait` в виде графика в течение значимого периода времени, например дня, двух дней или даже недели. `IOWait` указывается в процентах от общей пропускной способности доступа к диску системы. Если в течение продолжительных периодов времени этот показатель близок к 100 % на хосте, на котором не выполняется резервное копирование, это может указывать на полное сканирование таблиц и неэффективные запросы. Вы также хотите отслеживать общее использование пропускной способности дискового ввода/вывода в процентах, поскольку это может предупредить о том, что доступ к диску станет в будущем узким местом для производительности вашей базы данных.

Автоинкрементный диапазон

Одной из менее известных проблем при использовании MySQL является то, что первичные ключи с автоинкрементом по умолчанию создаются как целые числа со знаком и диапазон возможных значений для ключа может быть исчерпан.

Это происходит, когда вы сделали много вставок и ключ автоинкремента достиг максимально возможного значения для своего типа данных. При планировании того, какие показатели вы должны отслеживать на долгосрочной основе, однозначно следует включить в него мониторинг оставшегося целочисленного пространства для всех таблиц, в которых для первичного ключа применяются автоинкременты, — это почти наверняка избавит вас от некоторых серьезных проблем в будущем, потому что вы сможете заранее спрогнозировать необходимость в большем пространстве ключей.

Как вы контролируете это пространство ключей? Есть несколько вариантов. Если вы уже используете РММ, интегрированный с Prometheus exporter, то эта функциональность уже встроена и все, что нужно сделать, — включить флаг `-collect.auto_increment.columns`. Если ваша команда не использует Prometheus, можете задействовать следующий запрос, который можно преобразовать либо в средство создания показателей, либо в предупреждение, которое сообщит о том, что какая-либо из ваших таблиц приближается к максимально возможному значению ключа. Этот запрос основан на `information_schema`, в которой есть все метаданные о таблицах в вашем экземпляре базы данных:

```
SELECT
  t.TABLE_SCHEMA AS `schema`,
  t.TABLE_NAME AS `table`,
  t.AUTO_INCREMENT AS `auto_increment`,
  c.DATA_TYPE AS `pk_type`,
  (
    t.AUTO_INCREMENT /
    (CASE DATA_TYPE
      WHEN 'tinyint'
        THEN IF(COLUMN_TYPE LIKE '%unsigned',
          255,
          127
        )
      WHEN 'smallint'
        THEN IF(COLUMN_TYPE LIKE '%unsigned',
          65535,
          32767
        )
      WHEN 'mediumint'
        THEN IF(COLUMN_TYPE LIKE '%unsigned',
          16777215,
          8388607
        )
      WHEN 'int'
        THEN IF(COLUMN_TYPE LIKE '%unsigned',
          4294967295,
          2147483647
        )
      WHEN 'bigint'
```

```

        THEN IF(COLUMN_TYPE LIKE '%unsigned',
                18446744073709551615,
                9223372036854775807
        )
    END / 100)
) AS `max_value`
FROM information_schema.TABLES t
INNER JOIN information_schema.COLUMNS c
    ON t.TABLE_SCHEMA = c.TABLE_SCHEMA
    AND t.TABLE_NAME = c.TABLE_NAME
WHERE
    t.AUTO_INCREMENT IS NOT NULL
    AND c.COLUMN_KEY = 'PRI'
    AND c.DATA_TYPE LIKE '%int'
;

```

Имеется много нюансов и сопутствующих факторов, о которых нужно подумать при выборе первичного ключа в целом и при управлении автоинкрементами в частности, и мы рассмотрим эти вопросы в главе 6.

Время создания/восстановления резервной копии

Долгосрочное планирование — это не только рост, пока бизнес работает в обычном режиме, но и восстановление в допустимые сроки. В главе 10 мы более подробно обсудим, как продумывать аварийное восстановление, а в главе 13 разберем, почему оно является частью ваших обязанностей по контролю за соблюдением требований. Здесь же говорим об этом, чтобы подчеркнуть: хороший план аварийного восстановления работает только тогда, когда вы регулярно пересматриваете его и корректируете его цели.

ФУНКЦИОНАЛЬНОЕ СЕГМЕНТИРОВАНИЕ И ГОРИЗОНТАЛЬНОЕ РАЗДЕЛЕНИЕ

В этой и других главах книги вы увидите, что мы упоминаем сегментирование или секционирование как разные способы разделения ваших данных на отдельные экземпляры с целью масштабирования. Мы хотим объяснить, что подразумеваем под ними и чем они различаются, чтобы вы не путались, когда будете читать следующие главы.

- *Функциональное сегментирование (functional sharding)* означает размещение определенных таблиц, которые обслуживают определенную бизнес-функцию, на выделенных кластерах, чтобы отдельно управлять временем безотказной работы этого набора данных, производительностью или даже контролем доступа.
- *Горизонтальное сегментирование (horizontal sharding)* — это когда у вас есть набор данных, объем которых превысил размер, который вы можете надежно обслуживать на одном кластере, поэтому вы разделяете его на несколько кластеров и обслуживаете данные с нескольких узлов, полагаясь на некоторый механизм поиска данных, чтобы найти нужное их подмножество.

Если ваши базы данных достигают размера, при котором восстановление из резервной копии займет больше времени, чем приемлемо для восстановления критически важных функций бизнеса, тогда, даже если все остальное работает нормально, вам необходимо рассмотреть возможность корректировки этой цели MTTR, изменения определения «критическая функциональность» или поиска способа сокращения времени резервного копирования и восстановления. Вот кое-что, о чем следует подумать при планировании аварийного восстановления.

- Будьте *очень конкретны* при определении того, какие функциональные возможности относятся к целевому объекту восстановления, и, если необходимо, изучите, должны ли данные, обеспечивающие это подмножество функциональных возможностей, находиться в отдельном кластере, чтобы ожидания оправдались.
- Если функциональное разделение данных на несколько более мелких частей невозможно, весь набор данных теперь становится целью для восстановления с помощью резервных копий. Набор данных, восстановление которого из резервных копий занимает больше всего времени, будет определять время завершения восстановления.
- Убедитесь, что у вас есть автоматизированные методы тестирования (мы рассмотрим некоторые примеры в главе 10). Отслеживайте, сколько времени требуется для восстановления резервной копии из файла в работающую базу данных, которая успела выполнить репликацию всех изменений с момента создания резервной копии, и сохраните этот показатель, задав срок хранения таким, чтобы увидеть долгосрочные тенденции (не менее года). Это один из тех показателей, который может ускользнуть от внимания, и это время может стать на удивление длительным, если не автоматизировать его мониторинг.

Вы увидите, что во многих примерах долгосрочных показателей, которые мы кратко опишем, почти всегда указывается на необходимость либо функционального, либо горизонтального разделения ваших данных. Цель здесь — явно обозначить тот факт, что если вы анализируете возможность сегментирования при возникновении инцидентов, основной причиной которых являются проблемы с производительностью, то, вероятно, подумали об этом слишком поздно. Работа по разделению данных на управляемые части начинается не тогда, когда они стали слишком большими для одного кластера, а задолго до этого, когда вы все еще определяете, каковы ваши цели по обеспечению успешного обслуживания клиентов.

Понимание того, сколько времени вам потребуется для восстановления данных, может помочь определиться относительно того, что делать в случае реальной аварии. Это также может дать вам понять, когда все может занять больше времени, чем этого хочет бизнес. Это также предшествует осознанию необходимости сегментирования.

Измерение долгосрочной эффективности

Выбор SLI и SLO для ежедневных операций — это только начало. Вам нужно убедиться, что вы не путаете лес с деревьями и не фокусируетесь на конкретных показателях хоста вместо того, чтобы проверять общую производительность системы и результаты пользовательского опыта. В этом разделе мы рассмотрим стратегии, которые можно применить для общей оценки долгосрочной работоспособности системы.

Изучение вашего делового ритма

Важно быть в курсе роста трафика вашего бизнеса, поскольку именно в это время все ваши SLO подвергаются наибольшим испытаниям и получают наибольшее внимание со стороны ваших самых важных клиентов. Интенсивность бизнеса может означать, что пиковое время трафика на несколько порядков превышает среднее, и это может иметь серьезные последствия, если инфраструктура базы данных не подготовлена. В контексте инфраструктуры базы данных это может привести к увеличению на несколько порядков количества запросов, выполняемых в секунду, создать значительную нагрузку на подключение с ваших серверов приложений или серьезно повлиять на доход, если возникнут периодические сбои в операциях записи. Вот несколько примеров бизнес-ритма, которые должны помочь вам понять, в рамках какого бизнес-цикла работает ваша компания.

- *Сайт электронной коммерции.* С конца ноября до конца года — самая оживленная пора в интернет-магазинах многих стран, в это время продажи могут увеличиться на несколько порядков. Это означает гораздо больше тележек для покупок, гораздо больше одновременных продаж и гораздо большее влияние на доход при том же количестве сбоев, что и в любое другое время года.
- *Программное обеспечение для управления персоналом.* Обычно в Соединенных Штатах ноябрь — это период «открытой регистрации», когда сотрудники компаний и организаций, включая правительство, могут вносить изменения в выбранные ими варианты льгот для сотрудников, такие как медицинское страхование.
- *Интернет-магазин свежих цветов.* День святого Валентина будет самым оживленным временем года, когда гораздо больше людей, чем обычно, будут заказывать доставку букетов.

Как можно заметить, эти бизнес-циклы могут сильно различаться в зависимости от потребностей клиентов, которые удовлетворяет ваш бизнес. Для вас крайне важно знать о цикле своего бизнеса, его воздействии на доход бизнеса, его репутацию и, следовательно, о том, насколько вы должны подготовиться, чтобы

удовлетворить спрос, не влияя на стабильность систем, которые вам поручено эксплуатировать.

Когда дело доходит до измерения производительности инфраструктуры базы данных, лежащей в основе бизнеса, важно не определять производительность изолированно от других важных показателей, которые отслеживает ваша инженерная структура.

Производительность базы данных должна быть частью более широкого разговора о производительности технического стека, а не рассматриваться как отдельный вопрос. Начните с максимально возможного использования тех же инструментов, которые применяет остальная часть вашей инженерной структуры. Желательно, чтобы показатели и панель мониторинга, на которые вы полагаетесь при определении производительности уровня базы данных, были такими же доступными, как показатели уровня приложения или даже те же панели мониторинга. Такое мышление, независимо от того, какую технологию или какого поставщика вы задействуете, будет иметь большое значение для создания среды, где каждый вкладывается в производительность полного стека, и уменьшения пресловутой стены, которую инженеры зачастую ощущают между функциями, которые они пишут, и базами данных, которые их поддерживают.

Эффективное отслеживание показателей

Есть ряд моментов, которые необходимо учитывать, когда речь заходит о долгосрочном планировании бизнеса (это далеко не полный список):

- планирование будущей производительности;
- прогнозирование того, когда необходимы серьезные улучшения, а когда достаточно постепенных изменений;
- планирование увеличения затрат на эксплуатацию инфраструктуры.

Вы должны иметь возможность не только оценивать работоспособность инфраструктуры хранилища данных в определенный момент времени, но и отслеживать улучшение или ухудшение производительности в долгосрочной перспективе. Это означает определение не только SLI и SLO, но и того, какие SLI и SLO остаются важными показателями для долгосрочных тенденций. Скорее всего, вы обнаружите, что не все показатели, которые можно использовать для принятия краткосрочных решений по запросу, подходят и для долгосрочного бизнес-планирования.

Прежде чем углубиться в то, какие показатели важны для долгосрочного планирования, поговорим об инструментах, которые позволяют расширить возможности мониторинга долгосрочных тенденций.

Применение инструментов мониторинга для контроля производительности

Измерение производительности важно как для того, чтобы понять, находимся ли мы в настоящее время в ситуации инцидента, так и для отслеживания и выявления тенденций на длительную перспективу. Инструмент, который содержит нужные вам показатели, так же важен, как и сами показатели. Какой смысл выбирать хорошие SLI, если вы не можете увидеть их динамику с течением времени таким образом, чтобы соотнести их с остальными показателями организации? Область инструментов мониторинга быстро растет, и существует множество весомых мнений о том, как они должны быть сделаны. Целью мониторинга является повышение прозрачности и акцентирование внимания на отслеживании результатов, а не выводимых показателей. Для достижения успеха в создании успешного стека инфраструктуры мониторинга требуются коллективные усилия.

Вместо того чтобы говорить здесь о конкретных инструментах, перечислим важные функции и аспекты, о которых следует подумать, решая, подходит ли инструмент для выявления долгосрочных тенденций.

Средние показатели неприемлемы

Независимо от того, управляете ли вы своим решением для мониторинга показателей самостоятельно как инженерная структура или используете программное обеспечение как услугу (SaaS), будьте осторожны с тем, как это решение для мониторинга нормализует данные для долгосрочного хранения. Многие решения по умолчанию преобразуют долгосрочные данные в средние значения (Graphite — один из первых, кто сделал это), и это большая проблема. Если вам нужно посмотреть на изменение показателя за период, превышающий несколько недель, среднее значение *сглаживает* пики, а это означает, что, если вы захотите увидеть, удвоится ли использование дискового ввода/вывода в следующем году, график средних точек данных, скорее всего, даст ложное чувство безопасности. Всегда обращайте внимание на пики при анализе данных за месяцы, чтобы ясно видеть случайные всплески.

Процентили — ваши друзья

Процентили основаны на упорядочении точек данных за определенный промежуток времени и удалении точек с наивысшим значением в зависимости от целевого процентиля (то есть если вы ищете 95-е место, удалите верхние 5 %). Это отличный способ сделать просматриваемые данные визуально более похожими на то, какими мы видим SLI и SLO. Если вы можете сделать так, чтобы

график, показывающий время ответа на ваш запрос, отображал 95-й процентиль, будет гораздо проще сопоставить его с SLO, которых вы хотите достичь для выполнения запроса приложения, и сделать показатели базы данных понятными для таких специалистов, как служба поддержки клиентов и инженеры, а не только для команды разработчиков баз данных.

Длительный период хранения и производительность

Может показаться очевидным, что производительность инструмента мониторинга при отображении длительных временных интервалов имеет большое значение. Если вы оцениваете решения для отслеживания тенденций бизнес-показателей, вам следует проверить, как меняется пользовательский опыт при запросе данных за все более и более длительные временные интервалы. Решение для показателей настолько хорошо, насколько это возможно для обеспечения доступа к этим данным, а не только для скорости приема или продолжительности их хранения.

Теперь, когда мы описали, как должен выглядеть инструмент долгосрочного мониторинга, обсудим, как все, что рассматривалось до сих пор при выборе SLI и SLO, может повлиять на архитектуру данных.

Использование SLO для управления общей архитектурой

Поддержание стабильного и хорошего качества обслуживания клиентов, в то время как бизнес растет, — немалый подвиг. По мере роста бизнеса сохранять даже существующие SLO, не говоря уже о том, чтобы устанавливать более амбициозные, становится все труднее и труднее. Возьмем, к примеру, что-то вроде доступности: каждый хочет как можно больше девяток времени безотказной работы как для чтения, так и для записи для всех своих данных. Но чем более строгих SLO вы хотите достичь, тем дороже становится работа, поскольку число пиковых транзакций с базой данных в секунду или ее размер также растут на порядки.

Используя SLI и SLO, которые обсуждались ранее, вы можете найти точки роста, в которых имеет смысл начать разбивать данные либо на функциональные сегменты, либо на разделы данных. Более подробно мы обсудим масштабирование MySQL с помощью сегментирования в главе 11, а здесь важно отметить, что те же SLI и SLO, которые сообщают вам, как работает система сейчас, могут помочь понять, когда пора инвестировать в масштабирование MySQL, чтобы отдельные кластеры оставались управляемыми в пределах границ SLO, которые поддерживают впечатление клиентов от работы с вами.

Наличие решения для мониторинга показателей, которое может обрабатывать как краткосрочные, так и долгосрочные показатели и обеспечивает удобство

отслеживания изменений, — очень важная часть отслеживания тактических показателей производительности вашей инфраструктуры базы данных и долгосрочных тенденций.

Резюме

В процессе применения концепций проектирования надежности для мониторинга инфраструктуры базы данных важно постоянно улучшать и пересматривать свои показатели и цели. Они не предназначены для того, чтобы быть высеченными на камне после того, как вы впервые определите некоторые SLI и SLO. По мере роста бизнеса вы станете лучше понимать, какой видят клиенты работу с вами, и это должно способствовать улучшению ваших SLI и SLO.

Выбирая показатели и назначая им цели, на которых будете концентрироваться, помните, что ваша главная цель — обеспечить качество обслуживания клиентов. Кроме того, не сосредотачивайте все свои усилия на параметрах, которые показывают, когда происходит инцидент, а потратьте некоторое время на мониторинг того, что может помочь *предотвратить* инциденты. Все это нацелено на проактивную работу по созданию положительного имиджа у клиентов.

Мы рекомендуем заранее установить цели в трех ключевых областях: задержка, доступность и ошибки. Эти области могут обеспечить массу информации о том, довольны ли ваши клиенты. Помимо этого, обеспечьте проактивный мониторинг в таких областях, как рост числа подключений, дисковое пространство, дисковый ввод/вывод и задержки.

Надеемся, что эта глава поможет вам понять, как применять проектирование надежности для успешного мониторинга MySQL по мере масштабирования вашей компании.

Performance Schema

Предоставила Света Смирнова

Настройка производительности баз данных при высокой нагрузке представляет собой итеративный цикл. Каждый раз, когда вы вносите изменения для настройки производительности базы данных, вам необходимо понять, дало ли это изменение какой-либо эффект. Ваши запросы выполняются быстрее, чем раньше? Замедляют ли блокировки работу приложения, или они полностью исчезли? Изменилось ли использование памяти? Изменилось ли время, затраченное на ожидание на диске? Как только поймете, как ответить на эти вопросы, вы сможете оценивать повседневные ситуации и реагировать на них быстрее и увереннее.

Performance Schema — это база данных, в которой хранятся данные, необходимые для ответа на эти вопросы. Эта глава поможет вам понять, как работает Performance Schema, каковы ее ограничения и как лучше всего ее использовать (вместе с сопутствующей схемой `sys`) для получения общей информации о том, что происходит внутри MySQL.

Введение в Performance Schema

Performance Schema предоставляет низкоуровневые показатели операций, выполняемых внутри сервера MySQL. Чтобы объяснить, как работает Performance Schema, мне нужно представить две концепции.

Первая — это *инструмент*. Он относится к любой части кода MySQL, о которой мы хотим получить информацию. Например, если хотим собрать информацию о блокировках метаданных, нам нужно включить инструмент `wait/lock/meta data/sql/mdl`.

Второй концепцией является *потребитель*, представляющий собой просто таблицу, в которой хранится информация о том, какой код был инструментирован. Если мы инструментируем запросы, потребитель будет записывать такую информацию, как общее количество выполнений, сколько раз индекс не использовался, затраченное время и т. д. Потребитель — это то, что у большинства людей ассоциируется с Performance Schema.

Общее функционирование Performance Schema показано на рис. 3.1.

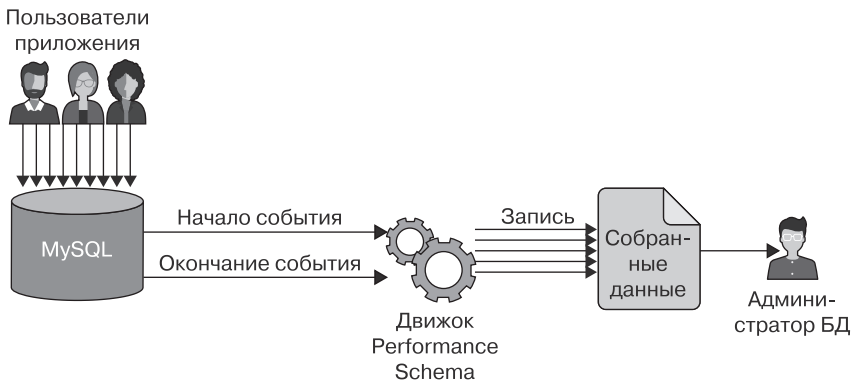


Рис. 3.1. Поток запросов, выполняемых в базе данных, показывающий, как `performance_schema` собирает и агрегирует данные, а затем представляет их администратору базы данных

Когда пользователи приложения подключаются к MySQL и выполняют инструментальную инструкцию, `performance_schema` инкапсулирует каждый проверенный вызов в два макроса, а затем записывает результаты в соответствующую таблицу потребителя. Вывод здесь заключается в том, что включение инструментов вызывает дополнительный код, а это, в свою очередь, означает, что инструменты потребляют ресурсы ЦП.

Список инструментов

В `performance_schema` таблица `setup_instruments` содержит список всех поддерживаемых инструментов. Названия всех инструментов состоят из частей, разделенных косой чертой. Я буду использовать следующие примеры, чтобы помочь вам разобраться, как они называются:

- `statement/sql/select;`
- `wait/synch/mutex/innodb/autoinc_mutex.`

Крайняя левая часть названия инструмента обозначает его тип. Таким образом, `statement` указывает, что инструмент является оператором, `wait` — что это ожидание и т. д.

Остальные элементы в поле имени слева направо обозначают подсистему от общего к частному. В предыдущем примере `select` является частью подсистемы `sql`, которая имеет тип `statement`. Или `autoinc_mutex` принадлежит `innodb`, который является частью более общего класса инструмента `mutex`, который, в свою очередь, является частью более общего инструмента `sync` типа `wait`.

Большинство названий инструментов говорят сами за себя. Как и в приведенных примерах, `statement/sql/select` — это запрос `SELECT`, `wait/synch/mutex/innodb/autoinc_mutex` — это мьютекс, который InnoDB устанавливает для столбца с автоинкрементом. В таблице `setup_instruments` есть также столбец `DOCUMENTATION`, который может содержать более подробную информацию:

```
mysql> SELECT * FROM performance_schema.setup_instruments
-> WHERE DOCUMENTATION IS NOT NULL LIMIT 5, 5\G
***** 1. row *****
NAME: statement/sql/error
ENABLED: YES
TIMED: YES PROPERTIES:
VOLATILITY: 0
DOCUMENTATION: Invalid SQL queries (syntax error).
***** 2. row *****
NAME: statement/abstract/Query
ENABLED: YES
TIMED: YES
PROPERTIES: mutable
VOLATILITY: 0
DOCUMENTATION: SQL query just received from the network. At this point,
the real statement type is unknown, the type will be refined
after SQL parsing.
***** 3. row *****
NAME: statement/abstract/new_packet
ENABLED: YES
TIMED: YES
PROPERTIES: mutable
VOLATILITY: 0
DOCUMENTATION: New packet just received from the network. At this point,
the real command type is unknown, the type will be refined after reading
the packet header.
***** 4. row *****
NAME: statement/abstract/relay_log
ENABLED: YES
TIMED: YES
PROPERTIES: mutable
VOLATILITY: 0
```

```
DOCUMENTATION: New event just read from the relay log. At this point, the real
statement type is unknown, the type will be refined after parsing the event.
***** 5. row *****
NAME: memory/performance_schema/mutex_instances
ENABLED: YES
TIMED: NULL
PROPERTIES: global_statistics
VOLATILITY: 1
DOCUMENTATION: Memory used for table performance_schema.mutex_instances
5 rows in set (0,00 sec)
```

К сожалению, столбец `DOCUMENTATION` может содержать значение `NULL` для многих инструментов, поэтому необходимо использовать имя инструмента, интуицию и знание исходного кода MySQL, чтобы понять, что исследует конкретный инструмент.

Организация потребителей

Как я упоминала ранее, потребитель — это место назначения, куда инструмент отправляет информацию. Performance Schema хранит результаты измерений во многих таблицах (на самом деле MySQL Community 8.0.25 содержит 110 таблиц в `performance_schema`). Чтобы понять, для чего они предназначены, стоит разделить их на группы.

Текущие и исторические данные

События помещаются в таблицы, имена которых заканчиваются следующим образом:

- `*_current` — события, происходящие на сервере в данный момент;
- `*_history` — последние 10 завершенных событий на поток;
- `*_history_long` — последние 10 000 завершенных событий на поток по всему миру.

Размеры таблиц `*_history` и `*_history_long` настраиваются. Текущие и исторические данные доступны для следующих событий:

- `events_waits` — низкоуровневые ожидания сервера, такие как получение мьютексов;
- `events_statements` — операторы SQL;
- `events_stages` — информация о профиле, например создание временных таблиц или отправка данных;
- `events_transactions` — транзакции.

Сводные таблицы и дайджесты

Сводная таблица содержит агрегированную информацию обо всем, что предлагается в таблице. Например, таблица `memory_summary_by_thread_by_event_name` содержит агрегированные значения использования памяти каждым потоком MySQL для пользовательских подключений или любого фонового потока.

Дайджесты — это способ агрегирования запросов путем удаления из них вариаций. Рассмотрим следующие примеры запросов:

```
SELECT user,birthdate FROM users WHERE user_id=19;  
SELECT user,birthdate FROM users WHERE user_id=13;  
SELECT user,birthdate FROM users WHERE user_id=27;
```

Дайджест для этого запроса будет таким:

```
SELECT user,birthdate FROM users WHERE user_id=?
```

Это позволяет Performance Schema отслеживать такие показатели, как задержка для дайджеста, не сохраняя каждый вариант запроса отдельно.

Экземпляры

Экземпляры ссылаются на экземпляры объектов, доступные при установке MySQL. Например, таблица `file_instances` содержит имена файлов и количество потоков, обращающихся к этим файлам.

Настройка

Таблицы настройки используются для настройки `performance_schema` во время выполнения.

Другие таблицы

Имеются и другие таблицы, имена которых не соответствуют строгому шаблону. Например, таблица `metadata_locks` содержит данные о блокировках метаданных. Я представлю некоторые из них позже в этой главе при обсуждении проблем, которые может помочь решить `performance_schema`.

Потребление ресурсов

Данные, собранные Performance Schema, хранятся в памяти. Вы можете ограничить объем используемой памяти, установив максимальный размер потребителей. Некоторые таблицы в `performance_schema` поддерживают автоматическое

масштабирование. Это означает, что они выделяют минимальный объем памяти при запуске и регулируют свой размер по мере необходимости. Однако эта память никогда не освобождается после выделения, даже если вы отключили определенные инструменты и выполнили усечение (`truncate`) этой таблицы.

Как я упоминала ранее, каждый инструментальный вызов добавляет еще два вызова макроса для сохранения данных в `performance_schema`. Это означает, что чем больше инструментов вы используете, тем выше будет загрузка процессора. Фактическое влияние на загрузку процессора зависит от конкретного инструмента. Например, инструмент, связанный с оператором (`statement`), может вызываться только один раз во время запроса, а инструмент, связанный с ожиданием (`wait`), — гораздо чаще. Например, для сканирования таблицы InnoDB с миллионом строк движку потребуется установить и снять блокировку 1 млн строк. Если вы задействуете инструмент для блокировки (`lock`), загрузка ЦП может значительно возрасти. Однако для того же оператора запроса потребуется один вызов, чтобы выяснить, является ли он `statement/sql/select`. Таким образом, вы не заметите увеличения нагрузки на ЦП, если включите инструментарий операторов. То же самое относится и к инструментам блокировки памяти или метаданных.

Ограничения

Прежде чем обсуждать, как настроить и использовать `performance_schema`, важно понять ее ограничения.

- *Инструментарий должен поддерживаться компонентом MySQL.* Например, предположим, что вы применяете инструментарий памяти, чтобы вычислить, какой компонент или поток MySQL использует наибольшую часть памяти. Вы обнаружите, что компонент, занимающий больше всего памяти, — это подсистема хранения, которая не поддерживает инструментарий памяти. В этом случае вы не сможете найти, куда делась память.
- *Она собирает данные только после включения конкретного инструмента и потребителя.* Например, если вы запустили сервер со всеми отключенными инструментами, а затем решили инструментировать использование памяти, то не сможете узнать точный объем, выделенный глобальным буфером, таким как пул буферов InnoDB, потому что он был выделен до того, как вы включили инструментарий памяти.
- *Трудно освободить память.* Вы можете ограничить размер потребителей при запуске или довериться им. В последнем случае они выделяют память

не при запуске, а только при сборе разрешенных данных. Однако, даже если вы позже отключите определенные инструменты или потребители, память не будет освобождена, пока вы не перезапустите сервер.

В оставшейся части главы я предполагаю, что вы знаете об этих ограничениях, поэтому не буду специально заострять на них внимание.

Схема sys

Начиная с версии 5.7, стандартный дистрибутив MySQL включает сопутствующую схему для данных `performance_schema`, называемую схемой `sys`. Она состоит только из представлений и хранимых подпрограмм над `performance_schema`. Хотя она предназначена для того, чтобы работа с `performance_schema` протекала более гладко, сама по себе она не хранит никаких данных.



Схема `sys` очень удобна, но нужно помнить, что она обращается только к данным, хранящимся в таблицах `performance_schema`. Если вам нужны данные, недоступные в схеме `sys`, проверьте, существуют ли они в базовой таблице в `performance_schema`.

Кратко о потоках

Сервер MySQL является многопоточным программным обеспечением. Каждый из его компонентов задействует потоки. Это может быть фоновый поток, созданный, например, основным потоком или подсистемой хранения, или приоритетный поток, созданный для подключения пользователя. Каждый из потоков имеет по крайней мере два уникальных идентификатора: идентификатор потока операционной системы, который виден, например, в выводе команды Linux ``ps -elf``, и внутренний идентификатор потока MySQL. Последний идентификатор называется `THREAD_ID` в большинстве таблиц `performance_schema`. Кроме того, каждому потоку переднего плана назначен идентификатор `PROCESS_LIST_ID` — идентификатор подключения, который отображается в выводе команды `SHOW PROCESSLIST` или в строке `Your MySQL connection id is` при подключении с помощью клиента командной строки MySQL.



`THREAD_ID` не аналогичен `PROCESSLIST_ID`!

Таблица `threads` в `performance_schema` содержит все потоки, существующие на сервере:

```
mysql> SELECT NAME, THREAD_ID, PROCESSLIST_ID, THREAD_OS_ID
-> FROM performance_schema.threads;
```

NAME	THREAD_ID	PROCESSLIST_ID	THREAD_OS_ID
thread/sql/main	1	NULL	797580
thread/innodb/io_ib...	3	NULL	797583
thread/innodb/io_lo...	4	NULL	797584
...			
thread/sql/slave_io	42	5	797618
thread/sql/slave_sql	43	6	797619
thread/sql/event_sc...	44	7	797620
thread/sql/signal_h...	45	NULL	797621
thread/mysqlx/accep...	46	NULL	797623
thread/sql/one_conn...	27823	27784	797695
thread/sql/compress...	48	9	797624

44 rows in set (0.00 sec)

Помимо информации о номере потока, таблица `threads` содержит те же данные, что и выходные данные `SHOW PROCESSLIST`, и несколько дополнительных столбцов, таких как `RESOURCE_GROUP` или `PARENT_THREAD_ID`.



`performance_schema` использует `THREAD_ID` везде, пока `PROCESSLIST_ID` доступен только в таблице `threads`. Если вам нужно получить `PROCESSLIST_ID` — например, чтобы прервать соединение, удерживающее блокировку, — то необходимо запросить таблицу `threads`, чтобы получить ее значение.

Таблицу `threads` можно соединить со многими другими таблицами для предоставления дополнительной информации о выполняющемся запросе, например данных запроса, блокировок, мьютексов или открытых экземпляров таблиц.

В остальной части главы я предполагаю, что вы знакомы с этой таблицей и значением `THREAD_ID`.

Конфигурация

Некоторые части Performance Schema можно изменить только при запуске сервера: включение или отключение самой Performance Schema и переменных, связанных с использованием памяти и ограничениями для собираемых данных.

Инструменты и потребители Performance Schema могут быть включены или отключены динамически.



Можно запустить Performance Schema со всеми отключенными потребителями и инструментами и включить только необходимые для решения конкретных проблем непосредственно перед ожидаемым возникновением проблемы. Таким образом вы не будете тратить ресурсы на Performance Schema там, где это не нужно, и не рискуете перегрузить систему из-за избыточного инструментария.

Включение и отключение Performance Schema

Чтобы включить или отключить Performance Schema, установите для переменной `performance_schema` значение `ON` или `OFF` соответственно. Это переменная, доступная только для чтения, которую можно изменить лишь в файле конфигурации или с помощью параметра командной строки при запуске сервера MySQL.

Включение и отключение инструментов

Инструменты могут быть как включены, так и отключены. Чтобы увидеть состояние инструмента, можно запросить таблицу `setup_instruments`:

```
mysql> SELECT * FROM performance_schema.setup_instruments
-> WHERE NAME='statement/sql/select'\G
***** 1. row *****
NAME: statement/sql/select
ENABLED: NO
TIMED: YES
PROPERTIES:
VOLATILITY: 0
DOCUMENTATION: NULL
1 row in set (0.01 sec)
```

Как видите, `ENABLED` имеет значение `NO`, это говорит о том, что в настоящее время мы не обрабатываем запросы `SELECT`.

Существует три варианта включения или отключения инструментов `performance_schema`:

- использовать таблицу `setup_instruments`;
- вызвать хранимую процедуру `ps_setup_enable_instrument` в схеме `sys`;
- применить параметр запуска `performance-schema-instrument`.

Оператор UPDATE

Первый метод заключается в использовании оператора UPDATE для изменения значения столбца:

```
mysql> UPDATE performance_schema.setup_instruments
      -> SET ENABLED='YES' WHERE NAME='statement/sql/select';
Query OK, 1 rows affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

Поскольку это стандартный SQL, можно использовать подстановочные знаки, чтобы включить все инструменты операторов SQL:

```
mysql> UPDATE performance_schema.setup_instruments
      -> SET ENABLED='YES' WHERE NAME LIKE statement/sql/%';
Query OK, 167 rows affected (0.00 sec)
Rows matched: 167 Changed: 167 Warnings: 0
```

Этот метод не обеспечивает сохранения состояния между перезапусками.

Хранимая процедура sys

Схема sys предоставляет две хранимые процедуры — `ps_setup_enable_instrument` и `ps_setup_disable_instrument`, которые включают и отключают инструменты, передаваемые им в качестве параметров. Обе подпрограммы поддерживают подстановочные знаки. Если вы хотите включить или отключить все поддерживаемые инструменты, используйте подстановочный знак %:

```
mysql> CALL sys.ps_setup_enable_instrument('statement/sql/select');
+-----+
| summary                               |
+-----+
| Enabled 1 instrument                  |
+-----+
1 row in set (0.01 sec)
```

Этот метод точно такой же, как и предыдущий, в том числе то, что состояние не сохраняется между перезапусками.

Параметры запуска

Как упоминалось ранее, оба метода позволяют вам изменить конфигурацию `performance_schema` онлайн, но не сохраняют это изменение между перезапусками сервера. Если вы хотите сохранить параметры для определенных инструментов между перезапусками, используйте параметр конфигурации `performance-schema-instrument`.

Эта переменная поддерживает синтаксис `performance-schema-instrument='instrument_name=value'`, где `instrument_name` — это имя инструмента, а `value`

равно ON, TRUE или 1 для включенных инструментов, OFF, FALSE или 0 — для отключения и COUNTED — для тех, которые учитываются вместо TIMED. Вы можете указать эту опцию несколько раз, чтобы включить или отключить разные инструменты. Опция поддерживает подстановочные знаки:

```
performance-schema-instrument='statement/sql/select=ON'
```



Если указаны несколько параметров, более длинная строка инструмента имеет приоритет над более короткой независимо от порядка.

Включение и отключение потребителей

Как и инструменты, потребители могут быть включены или отключены путем:

- обновления таблицы `setup_consumers` в Performance Schema;
- использования хранимых процедур `ps_setup_enable_consumer` и `ps_setup_disable_consumer` в схеме системы;
- установки параметра конфигурации `performance-schema-consumer`.

Есть 15 возможных потребителей. У некоторых из них имена говорят сами за себя, но у нескольких нуждаются в дополнительных пояснениях (табл. 3.1).

Таблица 3.1. Потребители и их цели

Потребитель	Описание
events_stages_[current history history_long]	Детали профилирования, такие как Creating tmp table, statistics или buffer pool load
events_statements_[current history history_long]	Статистика операторов
events_transactions_[current history history_long]	Транзакции
events_waits_[current history history_long]	Ожидания
global_instrumentation	Включает или отключает глобальную инструментовку. Если этот параметр отключен, никакие отдельные параметры не проверяются и никакие глобальные данные или данные для каждого потока не сохраняются. Ни одно отдельное событие не собирается
thread_instrumentation	Инструментарий для каждого потока. Проверяется только в том случае, если включена глобальная инструментовка. Если этот параметр отключен, данные о каждом потоке или отдельных событиях не собираются
statements_digest	Дайджесты операторов

Примеры, приведенные для инструментов, можно повторить для потребителей с использованием указанных методов.

Настройка мониторинга для конкретных объектов

Performance Schema позволяет включать и отключать мониторинг для определенных типов объектов, схем и имен. Это делается в таблице `setup_objects`.

Столбец `OBJECT_TYPE` может иметь одно из пяти значений: `EVENT`, `FUNCTION`, `PROCEDURE`, `TABLE` и `TRIGGER`. Кроме того, вы можете указать `OBJECT_SCHEMA` и `OBJECT_NAME`. Поддерживаются подстановочные знаки.

Например, чтобы отключить `performance_schema` для триггеров в тестовой базе данных, используйте следующую инструкцию:

```
mysql> INSERT INTO performance_schema.setup_objects
-> (OBJECT_TYPE, OBJECT_SCHEMA, OBJECT_NAME, ENABLED)
-> VALUES ('TRIGGER', 'test', '%', 'NO');
```

Если хотите создать исключение для триггера с именем `my_trigger`, добавьте его с помощью оператора:

```
mysql> INSERT INTO performance_schema.setup_objects
-> (OBJECT_TYPE, OBJECT_SCHEMA, OBJECT_NAME, ENABLED)
-> VALUES ('TRIGGER', 'test', 'my_trigger', 'YES');
```

Когда `performance_schema` решает, нужно ли инструментировать конкретный объект, она сначала ищет более конкретное правило, а затем возвращается к менее конкретному. Например, если пользователь выполняет запрос к таблице, которая запускает `test.my_trigger`, он будет проверять операторы, запускаемые триггером. Но если пользователь выполняет запрос к таблице, которая запускает триггер с именем `test.some_other_trigger`, то последний не будет проверен.

Для объектов нет опции файла конфигурации. Если вам нужно сохранить изменения в этой таблице во время перезапуска, то следует записать эти операторы `INSERT` в файл `SQL` и применить параметр `init_file` для загрузки файла `SQL` при запуске.

Настройка мониторинга потоков

Таблица `setup_threads` содержит список фоновых потоков, которые можно отслеживать. Столбец `ENABLED` указывает, включен ли инструментарий для конкретного потока. Столбец `HISTORY` указывает, должны ли инструментиро-

ванные события для конкретного потока также храниться в таблицах `_history` и `_history_long`.

Например, чтобы отключить ведение журнала для планировщика событий (`thread/sql/event_scheduler`), запустите:

```
mysql> UPDATE performance_schema.setup_threads
      -> SET HISTORY='NO'      -> WHERE NAME='thread/sql/event_scheduler';
```

В таблице `setup_threads` не хранятся настройки для пользовательских потоков. Для этой цели предназначена таблица `setup_actors`, содержащая столбцы, описанные в табл. 3.2.

Таблица 3.2. Столбцы, содержащиеся в таблице `setup_actors`

Имя столбца	Описание
HOST	Хост, например <code>localhost</code> , <code>%</code> , <code>my.domain.com</code> или <code>199.27.145.65</code>
USER	Имя пользователя, например <code>sveta</code> или <code>%</code>
ROLE	Не используется
ENABLED	Если поток включен
HISTORY	Если включено сохранение данных в таблицах <code>_history</code> и <code>_history_long</code>

Чтобы указать правила для определенных учетных записей, примените команду, подобную этой:

```
mysql> INSERT INTO performance_schema.setup_actors
      -> (HOST, USER, ENABLED, HISTORY)
      -> VALUES ('localhost', 'sveta', 'YES', 'NO'),
      -> ('example.com', 'sveta', 'YES', 'YES'),
      -> ('localhost', '%', 'NO', 'NO');
```

Этот оператор позволяет применять инструменты для `sveta@localhost` и `sveta@example.com`, отключает историю для `sveta@localhost` и отключает как инструментарий, так и историю для всех других пользователей, подключенных с локального хоста.

Как и в случае мониторинга объектов, нет файла конфигурации для потоков и факторов. Если вам нужно сохранить изменения в этой таблице во время перезапуска, то следует записать эти операторы `INSERT` в файл `SQL` и использовать параметр `init_file` для загрузки файла `SQL` при запуске.

Настройка размера памяти для Performance Schema

Performance Schema хранит данные в таблицах, которые задействуют механизм PERFORMANCE_SCHEMA. Этот движок хранит данные в памяти. Размер некоторых таблиц `performance_schema` автоматически изменяется по умолчанию, другие имеют фиксированное количество строк. Вы можете настроить эти параметры, изменив переменные запуска. Имена переменных соответствуют шаблону `performance_schema_object_[size|instances|classes|length|handles]`, где объектом является либо потребитель, либо таблица настройки, либо инструментированный экземпляр определенного события. Например, переменная конфигурации `performance_schema_events_stages_history_size` определяет количество этапов для каждого потока, которое будет храниться в таблице `performance_schema_events_stages_history`. Переменная `performance_schema_max_memory_classes` определяет максимальное количество инструментов памяти, которые могут быть использованы.

Значения по умолчанию

Значения параметров по умолчанию для разных частей MySQL меняются от версии к версии, поэтому стоит обратиться к справочному руководству пользователя, прежде чем полагаться на значения, описанные здесь. Однако для Performance Schema они влияют на общую производительность сервера, поэтому я хочу осветить наиболее важные из них.

Начиная с версии 5.7, Performance Schema включена по умолчанию, а большинство инструментов отключено. Включены только глобальные инструменты и инструментальные средства потоков, операторов и транзакций. Начиная с версии 8.0, блокировка метаданных и инструментарий памяти дополнительно включены по умолчанию.

Базы данных `mysql`, `information_schema` и `performance_schema` не инструментированы. Все остальные объекты, потоки и акторы инструментированы.

Размеры большинства экземпляров, дескрипторов и таблиц настройки изменяются автоматически. Для таблиц `_history` хранятся последние десять событий каждого потока. Для таблиц `_history_long` сохраняются последние 10 000 событий на поток. Максимальная длина сохраненного текста SQL — 1024 байта. Максимальная длина дайджеста SQL — также 1024 байта. Все более длинные тексты обрезаются справа.

Использование Performance Schema

Теперь, когда я рассказала о том, как настраивается Performance Schema, хочу привести примеры, которые помогут вам устранять распространенные неполадки.

Анализ операторов SQL

Как я упоминала в разделе «Список инструментов» ранее в этой главе, Performance Schema поддерживает богатый набор инструментов для проверки производительности SQL-операторов. Вы найдете инструменты для стандартных подготовленных операторов и хранимых процедур. С помощью `performance_schema` легко сможете определить, какой запрос вызывает проблемы с производительностью и по какой причине.

Чтобы включить инструментирование операторов, необходимо включить инструменты типа `statement`, как описано в табл. 3.3.

Таблица 3.3. Инструменты типа `statement` и их описания

Класс инструмента	Описание
<code>statement/sql</code>	Операторы SQL, такие как <code>SELECT</code> или <code>CREATE TABLE</code>
<code>statement/sp</code>	Контроль хранимых процедур
<code>statement/scheduler</code>	Планировщик событий
<code>statement/com</code>	Такие команды, как <code>quit</code> , <code>KILL</code> , <code>DROP DATABASE</code> или <code>Binlog Dump</code> . Некоторые недоступны для пользователей и вызываются самим процессом <i>mysqld</i>
<code>statement/abstract</code>	Класс четырех команд: <code>clone</code> , <code>Query</code> , <code>new_packet</code> и <code>relay_log</code>

Обычные операторы SQL

Performance Schema хранит показатели операторов в таблицах `events_statements_current`, `events_statements_history` и `events_statements_history_long`. Все три таблицы имеют одинаковую структуру.

Использование `performance_schema` напрямую. Вот пример записи `event_statement_history`:

```
THREAD_ID: 3200
EVENT_ID: 22
END_EVENT_ID: 23
EVENT_NAME: statement/sql/select
```

```
SOURCE: init_net_server_extension.cc:94
TIMER_START: 878753511280779000
TIMER_END: 878753544491277000
TIMER_WAIT: 33210498000
LOCK_TIME: 657000000
SQL_TEXT: 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)
DIGEST: 2fdac27c4a9434806da3b216b9fa71aca738f70f1e8888a581c4fb00a349224f
DIGEST_TEXT: SELECT `film`.`film_id`, `film`.`description` FROM `sakila`.`
film` INNER JOIN ( SELECT `film_id` FROM `sakila`.`film` ORDER BY
`title` LIMIT?, ... ) AS `lim` USING ( `film_id` )
CURRENT_SCHEMA: sakila
OBJECT_TYPE: NULL
OBJECT_SCHEMA: NULL
OBJECT_NAME: NULL
OBJECT_INSTANCE_BEGIN: NULL
MYSQL_ERRNO: 0
RETURNED_SQLSTATE: NULL
MESSAGE_TEXT: NULL
ERRORS: 0
WARNINGS: 0
ROWS_AFFECTED: 0
ROWS_SENT: 5
ROWS_EXAMINED: 10
CREATED_TMP_DISK_TABLES: 0
CREATED_TMP_TABLES: 1
SELECT_FULL_JOIN: 0
SELECT_FULL_RANGE_JOIN: 0
SELECT_RANGE: 0
SELECT_RANGE_CHECK: 0
SELECT_SCAN: 2
SORT_MERGE_PASSES: 0
SORT_RANGE: 0
SORT_ROWS: 0
SORT_SCAN: 0
NO_INDEX_USED: 1
NO_GOOD_INDEX_USED: 0
NESTING_EVENT_ID: NULL
NESTING_EVENT_TYPE: NULL
NESTING_EVENT_LEVEL: 0
STATEMENT_ID: 25
```

Эти столбцы описаны в официальной документации, поэтому я не буду рассматривать каждый из них. В табл. 3.4 перечислены столбцы, которые могут быть использованы в качестве индикаторов для определения запросов, требующих оптимизации. Не все такие столбцы равны. Например, `CREATED_TMP_DISK_TABLES` в большинстве случаев является признаком плохо оптимизированного запроса, а четыре столбца, связанных с сортировкой, могут просто указывать на то, что результаты запроса требуют сортировки. Столбец «Важность» показывает, насколько важен индикатор.

Таблица 3.4. Инструменты типа statement и их описания

Столбец	Описание	Важность
CREATED_TMP_DISK_TABLES	Запрос создал указанное количество временных таблиц на диске. У вас есть два варианта решения этой проблемы: оптимизировать запрос или увеличить максимальный размер временных таблиц в памяти	High
CREATED_TMP_TABLES	Запрос создал указанное количество временных таблиц в памяти. Использование временных таблиц в памяти само по себе неплохо. Однако, если базовая таблица увеличивается, они могут быть преобразованы в таблицы, хранимые на диске. К таким ситуациям лучше подготовиться заранее	Medium
SELECT_FULL_JOIN	JOIN выполнил полное сканирование таблицы, потому что в противном случае нет хорошего индекса для разрешения запроса. Вам нужно пересмотреть свои индексы, если только таблица не очень маленькая	High
SELECT_FULL_RANGE_JOIN	Если JOIN применил поиск по диапазону ссылочной таблицы	Medium
SELECT_RANGE	Если JOIN использовал поиск по диапазону для разрешения строк в первой таблице. Обычно это небольшая проблема	Low
SELECT_RANGE_CHECK	Если JOIN без индексов, то ключи проверяются после каждой строки. Это очень плохой симптом, и вам нужно пересмотреть свои табличные индексы, если это значение больше нуля	High
SELECT_SCAN	Если JOIN выполнил полное сканирование первой таблицы. Это проблема, если таблица большая	Medium
SORT_MERGE_PASSES	Количество проходов слияния, которые должна выполнить сортировка. Если значение больше нуля и производительность запроса низкая, вам может потребоваться увеличить sort_buffer_size	Low
SORT_RANGE	Если сортировка производилась с использованием диапазонов	Low
SORT_ROWS	Количество отсортированных строк. Сравните со значением возвращенных строк. Если количество отсортированных строк больше, может потребоваться оптимизировать запрос	Medium (см. описание)

Продолжение ➤

Таблица 3.4 (продолжение)

Столбец	Описание	Важность
<code>SORT_SCAN</code>	Если сортировка производилась путем сканирования таблицы. Это очень плохой знак, если вы намеренно не выбираете все строки из таблицы без применения индекса	High
<code>NO_INDEX_USED</code>	Для разрешения запроса индекс не использовался	High, если таблицы не маленькие
<code>NO_GOOD_INDEX_USED</code>	Индекс, используемый для разрешения запроса, не самый лучший. Вам необходимо пересмотреть свои индексы, если это значение больше нуля	High

Чтобы узнать, какие операторы требуют оптимизации, можете выбрать любой из указанных столбцов и сравнить его с нулем. Например, чтобы найти все запросы, которые не используют хороший индекс, выполните следующий запрос:

```
SELECT THREAD_ID, SQL_TEXT, ROWS_SENT, ROWS_EXAMINED, CREATED_TMP_TABLES,
NO_INDEX_USED, NO_GOOD_INDEX_USED
FROM performance_schema.events_statements_history_long
WHERE NO_INDEX_USED > 0 OR NO_GOOD_INDEX_USED > 0;
```

Чтобы найти все запросы, создавшие временные таблицы, выполните запрос:

```
SELECT THREAD_ID, SQL_TEXT, ROWS_SENT, ROWS_EXAMINED, CREATED_TMP_TABLES,
CREATED_TMP_DISK_TABLES
FROM performance_schema.events_statements_history_long
WHERE CREATED_TMP_TABLES > 0 OR CREATED_TMP_DISK_TABLES > 0;
```

Вы можете использовать значения в этих столбцах, чтобы показать потенциальные проблемы по отдельности. Например, чтобы найти все запросы, которые возвращали ошибки, примените условие `WHERE ERRORS > 0`, чтобы найти все запросы, выполняемые более 5 с, — условие `WHERE TIMER_WAIT > 5000000000` и т. д.

В качестве альтернативы можете создать запрос, который найдет все операторы с проблемами, используя сложные условия, следующим образом:

```
WHERE ROWS_EXAMINED > ROWS_SENT
OR ROWS_EXAMINED > ROWS_AFFECTED
OR ERRORS > 0
OR CREATED_TMP_DISK_TABLES > 0
OR CREATED_TMP_TABLES > 0
OR SELECT_FULL_JOIN > 0
OR SELECT_FULL_RANGE_JOIN > 0
OR SELECT_RANGE > 0
```

```

OR SELECT_RANGE_CHECK > 0
OR SELECT_SCAN > 0
OR SORT_MERGE_PASSES > 0
OR SORT_RANGE > 0
OR SORT_ROWS > 0
OR SORT_SCAN > 0
OR NO_INDEX_USED > 0
OR NO_GOOD_INDEX_USED > 0

```

Использование схемы sys. Схема `sys` обеспечивает представления, которые можно задействовать для поиска проблемных операторов. Например, представление `statements_with_errors_or_warnings` выдает список всех операторов с ошибками и предупреждениями, а представление `statements_with_full_table_scans` — список всех операторов, требующих полного сканирования таблицы. Схема `sys` задействует текст дайджеста вместо текста запроса, поэтому вы получите текст дайджеста запроса вместо SQL или дайджеста, как при доступе к необработанным (raw) таблицам `performance_schema`:

```

mysql> SELECT query, total_latency, no_index_used_count, rows_sent,
-> rows_examined
-> FROM sys.statements_with_full_table_scans
-> WHERE db='employees' AND
-> query NOT LIKE '%performance_schema%\G
***** 1. row *****
query: SELECT COUNT ( 'emp_no' ) FROM ... 'emp_no' )
WHERE 'title' = ?
total_latency: 805.37 ms
no_index_used_count: 1
rows_sent: 1
rows_examined: 397774
...

```

Другие представления, с помощью которых можно искать операторы, требующие оптимизации, описаны в табл. 3.5.

Таблица 3.5. Представления, которые можно использовать для поиска операторов, требующих оптимизации

Представление	Описание
<code>statement_analysis</code>	Представление нормализованных операторов с агрегированной статистикой, упорядоченной по общему времени выполнения нормализованного оператора. Аналогично таблице <code>events_statements_summary_by_digest</code> , но менее детализировано

Продолжение ⇨

Таблица 3.5 (продолжение)

Представление	Описание
statements_with_errors_or_warnings	Все нормализованные операторы, вызвавшие ошибки или предупреждения
statements_with_full_table_scans	Все нормализованные операторы, которые выполнили полное сканирование таблицы
statements_with_runtimes_in_95th_percentile	Все нормализованные операторы, среднее время выполнения которых находится в верхнем 95-м процентиле
statements_with_sorting	Все нормализованные операторы, в которых выполнена сортировка. Представление включает в себя все виды сортировки
statements_with_temp_tables	Все нормализованные операторы, использовавшие временные таблицы

Подготовленные операторы

Таблица `prepare_statements_instances` содержит все подготовленные операторы, существующие на сервере. Она имеет ту же статистику, что и таблицы `events_statements_[current|history|history_long]`, и, кроме того, информацию о потоке, которому принадлежит подготовленный оператор, и о том, сколько раз оператор был выполнен.

В отличие от таблиц `events_statements_[current|history|history_long]` данные статистики суммируются и таблица содержит общее количество всех выполнений операторов.



Столбец `COUNT_EXECUTE` содержит количество выполнений оператора, поэтому вы можете получить среднюю статистику по операторам, разделив общее значение на число в этом столбце. Однако обратите внимание на то, что любая усредненная статистика может быть неточной. Например, если вы выполнили инструкцию десять раз и значение в столбце `SUM_SELECT_FULL_JOIN` равно 10, в среднем будет одно полное соединение на инструкцию. Если вы затем добавите индекс и выполните оператор еще раз, `SUM_SELECT_FULL_JOIN` останется равным 10, поэтому среднее значение будет $10 / 11 = 0,9$. Это не означает, что проблема теперь решена.

Чтобы включить инструментарий для подготовленных операторов, вам необходимо включить инструменты, описанные в табл. 3.6.

Таблица 3.6. Инструменты, позволяющие использовать инструментарий для подготовленных операторов

Представление	Описание
statement/sql/prepare_sql	Оператор PREPARE в текстовом протоколе (при запуске через MySQL CLI)
statement/sql/execute_sql	Оператор EXECUTE в текстовом протоколе (при запуске через MySQL CLI)
statement/com/Prepare	Оператор PREPARE в бинарном протоколе (при доступе через MySQL C API)
statement/com/Execute	Оператор EXECUTE в бинарном протоколе (при доступе через MySQL C API)

После включения можете подготовить оператор и выполнить его несколько раз:

```
mysql> PREPARE stmt FROM
-> 'SELECT COUNT(*) FROM employees WHERE hire_date > ?';
Query OK, 0 rows affected (0.00 sec)
Statement prepared

mysql1> SET @hd='1995-01-01';
Query OK, 0 rows affected (0.00 sec)
```

```
mysql1> EXECUTE stmt USING @hd;
+-----+
| count(*) |
+-----+
| 34004    |
+-----+
1 row in set (1.44 sec)
```

```
-- Execute a few more times with different values
```

Затем можете проверить диагностику:

```
mysql2> SELECT statement_name, sql_text, owner_thread_id,
-> count_reprepare, count_execute, sum_timer_execute
-> FROM prepared_statements_instances\G
***** 1. row *****
statement_name: stmt
sql_text: select count(*) from employees where hire_date > ?
owner_thread_id: 22
count_reprepare: 0
count_execute: 3
sum_timer_execute: 4156561368000
1 row in set (0.00 sec)
```

Обратите внимание на то, что вы увидите операторы в таблице `prepare_statements_instances` только в том случае, если они существуют на сервере.

После того как они будут удалены, вы больше не можете получить доступ к их статистике:

```
mysql1> DROP PREPARE stmt;
Query OK, 0 rows affected (0.00 sec)

mysql2> SELECT * FROM prepared_statements_instances\G
Empty set (0.00 sec)
```

Хранимые процедуры

С помощью `performance_schema` вы можете получить информацию о том, как выполнялись ваши хранимые процедуры: например, какая из ветвей оператора управления потоком `IF ... ELSE` была выбрана или был ли вызван обработчик ошибок.

Чтобы включить инструментарий хранимых подпрограмм, вам необходимо включить инструменты, которые соответствуют шаблону `'statement/sp/%'`. Инструкция инструмента `/sp/stmt` отвечает за операторы, вызываемые внутри процедуры, а другие инструменты — за отслеживание событий, таких как вход в процедуру, цикл или в любую другую команду управления либо выход из них.

Чтобы продемонстрировать, как работает инструментарий хранимых подпрограмм, создайте хранимую процедуру:

```
CREATE DEFINER='root'@'localhost' PROCEDURE 'sp_test'(val int)
BEGIN
  DECLARE CONTINUE HANDLER FOR 1364, 1048, 1366
  BEGIN
    INSERT IGNORE INTO t1 VALUES('Some string');
    GET STACKED DIAGNOSTICS CONDITION 1 @stacked_state = RETURNED_SQLSTATE;
    GET STACKED DIAGNOSTICS CONDITION 1 @stacked_msg = MESSAGE_TEXT;
  END;
  INSERT INTO t1 VALUES(val);
END
```

Затем вызовите его с разными значениями:

```
mysql> CALL sp_test(1);
Query OK, 1 row affected (0.07 sec)

mysql> SELECT THREAD_ID, EVENT_NAME, SQL_TEXT
-> FROM EVENTS_STATEMENTS_HISTORY
-> WHERE EVENT_NAME LIKE 'statement/sp%';
+-----+-----+-----+
| THREAD_ID | EVENT_NAME | SQL_TEXT |
+-----+-----+-----+
| 24 | statement/sp/hpush_jump | NULL |
| 24 | statement/sp/stmt | INSERT INTO t1 VALUES(val) |
| 24 | statement/sp/hpop | NULL |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

В этом случае обработчик ошибки не вызывался, а процедура вставляла в таблицу значение аргумента (1):

```
mysql> CALL sp_test(NULL);
Query OK, 1 row affected (0.07 sec)
```

```
mysql> SELECT THREAD_ID, EVENT_NAME, SQL_TEXT
        -> FROM EVENTS_STATEMENTS_HISTORY
        -> WHERE EVENT_NAME LIKE 'statement/sp%';
```

THREAD_ID	EVENT_NAME	SQL_TEXT
24	statement/sp/hpush_jump	NULL
24	statement/sp/stmt	INSERT INTO t1 VALUES(val)
24	statement/sp/stmt	INSERT IGNORE INTO t1 VALUES('Some str...
24	statement/sp/stmt	GET STACKED DIAGNOSTICS CONDITION 1 @s...
24	statement/sp/stmt	GET STACKED DIAGNOSTICS CONDITION 1 @s...
24	statement/sp/hreturn	NULL
24	statement/sp/hpop	NULL

7 rows in set (0.00 sec)

Однако во втором вызове содержимое таблицы `events_statements_history` другое: оно содержит вызовы из обработчика ошибок и оператор SQL, который заменил ошибочный.

Хотя возвращаемое значение самой процедуры не изменилось, мы ясно видим, что она была выполнена по-другому. Понимание таких различий в потоке выполнения процедуры может помочь понять, почему одна и та же процедура может завершиться почти сразу, если вызывается один раз, и занять гораздо больше времени при повторном вызове.

Профилирование операторов

Таблица `events_stages_[current|history|history_long]` содержит информацию о профилировании, например, сколько времени MySQL потратила на создание временной таблицы, обновление или ожидание блокировки. Чтобы включить профилирование, необходимо включить указанные потребители, а также инструменты, которые соответствуют шаблону `'stage/%'`. После включения вы можете найти ответы на такие вопросы, как «Какой этап выполнения запроса занял критически много времени?». В следующем примере выполняется поиск этапов, которые заняли более 1 с:

```
mysql> SELECT esh1.event_name, sql_text,
        -> esh1.timer_wait/1000000000 w_s
        -> FROM performance_schema.events_stages_history_long esh1
```

```

-> JOIN performance_schema.events_statements_history_long esth1
-> ON (esh1.nesting_event_id = esth1.event_id)
-> WHERE esh1.timer_wait > 1*1000000000\G
***** 1. row *****
event_name: stage/sql/Sending data
sql_text: SELECT COUNT(emp_no) FROM employees JOIN salaries
USING(emp_no) WHERE hire_date=from_date
w_s: 81.7
1 row in set (0.00 sec)

```

Другой метод использования таблиц `events_stages_[current|history|history_long]` заключается в том, чтобы обратить внимание на те операторы, на выполнение которых израсходовали больше определенного времени на этапах, вызывающих проблемы с производительностью. В табл. 3.7 перечислены эти этапы.

Таблица 3.7. Этапы, которые являются индикаторами проблем с производительностью

Класс (классы) этапов	Описание
stage/sql/%tmp%	Все, что связано с временными таблицами
stage/sql/%lock%	Все относящиеся к блокировкам
stage/%/Waiting for%	Все, что ждет ресурса
stage/sql/Sending data	Этот этап следует сравнить с количеством <code>ROWS_SENT</code> в статистике операторов. Если <code>ROWS_SENT</code> невелико, оператор, который тратит много времени на этом этапе, может означать, что он должен создать временный файл или таблицу для разрешения промежуточных результатов. За этим часто следует фильтрация строк перед отправкой данных клиенту. Обычно это симптом плохо оптимизированного запроса
stage/sql/freeing items, stage/sql/cleaning up, stage/sql/closing tables, stage/sql/end	Это этапы, которые очищают ресурсы. К сожалению, они недостаточно детализованы, и каждый из них включает в себя более одной задачи. Если вы видите, что ваши запросы занимают много времени на этих этапах, то, скорее всего, столкнулись с конкуренцией за ресурсы из-за высокого параллелизма. Вам необходимо проверить использование процессора, ввода/вывода и памяти, а также то, могут ли ваше оборудование и применяемые параметры MySQL обрабатывать параллелизм, создаваемый приложением

Очень важно отметить, что профилирование доступно только для общих этапов сервера. Подсистемы хранения не поддерживают профилирование с помощью `performance_schema`. В результате такие этапы, как `stage/sql/update`, означают, что задание находится внутри подсистемы хранения и может включать в себя не только выполнение оператора `update`, но и ожидание блокировок, специфичных для подсистемы хранения.

Анализ производительности чтения и записи

Инструментарий операторов в Performance Schema может быть очень полезен, чтобы понять, связана ли ваша рабочая нагрузка с чтением или записью. Вы можете начать с подсчета типов операторов:

```
mysql> SELECT EVENT_NAME, COUNT(EVENT_NAME)
-> FROM events_statements_history_long
-> GROUP BY EVENT_NAME;
```

EVENT_NAME	COUNT(EVENT_NAME)
statement/sql/insert	504
statement/sql/delete	502
statement/sql/select	6987
statement/sql/update	1007
statement/sql/commit	500
statement/sql/begin	500

```
6 rows in set (0.03 sec)
```

В этом примере запросов `SELECT` больше, чем любых других. Это показывает, что большинство запросов в данной настройке — запросы на чтение.

Если вы хотите узнать задержку своих операторов, агрегируйте по столбцу `LOCK_TIME`:

```
mysql> SELECT EVENT_NAME, COUNT(EVENT_NAME),
-> SUM(LOCK_TIME/1000000) AS latency_ms
-> FROM events_statements_history
-> GROUP BY EVENT_NAME ORDER BY latency_ms DESC;
```

EVENT_NAME	COUNT(EVENT_NAME)	latency_ms
statement/sql/select	194	7362.0000
statement/sql/update	33	1276.0000
statement/sql/insert	16	599.0000
statement/sql/delete	16	470.0000
statement/sql/show_status	2	176.0000
statement/sql/begin	4	0.0000
statement/sql/commit	2	0.0000
statement/com/Ping	2	0.0000
statement/sql/show_engine_status	1	0.0000

```
9 rows in set (0.01 sec)
```

Вы также можете захотеть узнать количество прочитанных и записанных байтов и строк. Для этого используйте глобальные переменные состояния `Handler_*`:

```
mysql> WITH rows_read AS (SELECT SUM(VARIABLE_VALUE) AS rows_read
-> FROM global_status
```

```

-> WHERE VARIABLE_NAME IN ('Handler_read_first', 'Handler_read_key',
-> 'Handler_read_next', 'Handler_read_last', 'Handler_read_prev',
-> 'Handler_read_rnd', 'Handler_read_rnd_next')),
-> rows_written AS (SELECT SUM(VARIABLE_VALUE) AS rows_written
-> FROM global_status
-> WHERE VARIABLE_NAME IN ('Handler_write'))
-> SELECT * FROM rows_read, rows_written\G
***** 1. row *****
rows_read: 169358114082
rows_written: 33038251685
1 row in set (0.00 sec)

```

Анализ блокировок метаданных

Блокировки метаданных используются для защиты определений объектов базы данных от модификации.

Блокировки общих метаданных устанавливаются для любого оператора SQL: SELECT, UPDATE и т. д. Они не влияют на другие операторы, для которых требуются общие блокировки метаданных. Однако они предотвращают выполнение операторов, которые изменяют определение объекта базы данных, таких как ALTER TABLE или CREATE INDEX, с момента запуска оператора SQL до снятия блокировки. Хотя большинство проблем, вызванных конфликтами блокировок метаданных, затрагивают таблицы, сами блокировки устанавливаются для любого объекта базы данных, например SCHEMA, EVENT, TABLESPACE и т. д.

Блокировки метаданных удерживаются до завершения транзакции. Это усложняет устранение неполадок, если вы используете несколько операторов транзакций. Какой оператор ожидает блокировки, обычно ясно: операторы DDL неявно фиксируют транзакции, поэтому они являются единственным оператором в новой транзакции, и вы найдете их в списке процессов со статусом «Ожидание блокировки метаданных». Однако оператор, который удерживает блокировку, может исчезнуть из списка процессов, если он является частью все еще открытой транзакции с несколькими операторами.

Таблица `metadata_locks` в `performance_schema` содержит информацию о блокировках, в настоящее время установленных различными потоками, а также о запросах на блокировку, которые ожидают блокировки. Таким образом, вы можете легко определить, какой поток не разрешает запустить ваш DDL-запрос, и решить, хотите ли вы принудительно завершить выполнение этого оператора или подождать, пока он сам завершит выполнение.

Чтобы включить инструментарий анализа блокировки метаданных, вам необходимо включить инструмент `wait/lock/metadata/sql/mdl`.

В следующем примере показано, что поток, видимый в списке процессов с идентификатором 5, удерживает блокировку, которую ожидает поток с идентификатором `processlist_id=4`:

```
mysql> SELECT processlist_id, object_type,
-> lock_type, lock_status, source
-> FROM metadata_locks JOIN threads ON (owner_thread_id=thread_id)
-> WHERE object_schema='employees' AND object_name='titles'\G
***** 1. row *****
processlist_id: 4
object_type: TABLE
lock_type: EXCLUSIVE
lock_status: PENDING -- waits
source: mdl.cc:3263
***** 2. row *****
processlist_id: 5
object_type: TABLE
lock_type: SHARED_READ
lock_status: GRANTED -- holds
source: sql_parse.cc:5707
```

Анализ использования памяти

Чтобы включить инструмент анализа использования памяти в `performance_schema`, включите инструменты класса `memory`. После этого вы сможете найти подробную информацию о том, как именно память задействуется внутренними структурами MySQL.

Применение `performance_schema` напрямую

Performance Schema хранит статистику использования памяти в сводных таблицах, имена которых начинаются с префикса `memory_summary_`. Параметры агрегации для использования памяти описаны в табл. 3.8.

Таблица 3.8. Параметры агрегации для использования памяти

Параметр агрегации	Описание
<code>global</code>	Глобально по имени события
<code>thread</code>	По потоку: включает как фоновые, так и пользовательские потоки
<code>account</code>	Учетная запись пользователя
<code>host</code>	Хост
<code>user</code>	Имя пользователя

Например, чтобы найти структуры InnoDB, которые задействуют большую часть памяти, выполните следующий запрос:

```
mysql> SELECT EVENT_NAME,
-> CURRENT_NUMBER_OF_BYTES_USED/1024/1024 AS CURRENT_MB,
-> HIGH_NUMBER_OF_BYTES_USED/1024/1024 AS HIGH_MB
-> FROM performance_schema.memory_summary_global_by_event_name
-> WHERE EVENT_NAME LIKE 'memory/innodb/%'
-> ORDER BY CURRENT_NUMBER_OF_BYTES_USED DESC LIMIT 10;
```

EVENT_NAME	CURRENT_MB	HIGH_MB
memory/innodb/buf_buf_pool	130.68750000	130.68750000
memory/innodb/ut0link_buf	24.00006104	24.00006104
memory/innodb/buf0dblwr	17.07897949	24.96951294
memory/innodb/ut0new	16.07891273	16.07891273
memory/innodb/sync0arr	6.25006866	6.25006866
memory/innodb/lock0lock	4.85086060	4.85086060
memory/innodb/ut0pool	4.00003052	4.00003052
memory/innodb/hash0hash	3.69776917	3.69776917
memory/innodb/os0file	2.60422516	3.61988068
memory/innodb/memory	1.23812866	1.42373657

10 rows in set (0,00 sec)

Использование схемы sys

В схеме sys есть представления, которые позволяют вам получать статистику памяти более эффективным способом. Они также поддерживают агрегирование по хосту, пользователю, потоку или глобально. Представление `memory_global_total` содержит единственное значение, отображающее общий объем инструментированной памяти:

```
mysql> SELECT * FROM sys.memory_global_total;
```

total_allocated
441.84 MiB

1 row in set (0,09 sec)

Представления агрегирования преобразуют байты в килобайты, мегабайты и гигабайты по мере необходимости. Представление `memory_by_thread_by_current_bytes` содержит столбец `user`, который может принимать одно из следующих значений:

- `NAME@HOST`. Учетная запись обычного пользователя, например `sveta@oreilly.com`;

- системные пользователи, такие как `sql/main` или `innodb/*`. Данные для таких имен пользователей берутся из таблицы потоков и удобны, когда вам нужно понять, что делает конкретный поток.

Строки в представлении `memory_by_thread_by_current_bytes` отсортированы по выделенной в данный момент памяти в порядке убывания, поэтому вы легко найдете, какой поток занимает бо́льшую часть памяти:

```
mysql> SELECT thread_id tid, user,
-> current_allocated ca, total_allocated
-> FROM sys.memory_by_thread_by_current_bytes LIMIT 9;
```

tid	user	ca	total_allocated
52	sveta@localhost	1.36 MiB	10.18 MiB
1	sql/main	1.02 MiB	4.95 MiB
33	innodb/clone_gtid_thread	525.36 KiB	24.04 MiB
44	sql/event_scheduler	145.72 KiB	4.23 MiB
43	sql/slave_sql	48.74 KiB	142.46 KiB
42	sql/slave_io	20.03 KiB	232.23 KiB
48	sql/compress_gtid_table	13.91 KiB	17.06 KiB
25	innodb/fts_optimize_thread	1.92 KiB	2.00 KiB
34	innodb/srv_purge_thread	1.56 KiB	1.64 KiB

```
9 rows in set (0,03 sec)
```

Результат предыдущего примера получен на ноутбуке, поэтому цифры не характеризуют рабочий сервер. По-прежнему ясно, что бо́льшую часть памяти задействует локальное соединение, за которым следует основной серверный процесс.

Инструментарий анализа использования памяти удобен, когда вам нужно найти пользовательский поток, который занимает больше всего памяти. В следующем примере пользовательское подключение выделило 36 Гбайт ОЗУ, что довольно много даже для современных систем с большим объемом памяти:

```
mysql> SELECT * FROM sys.memory_by_thread_by_current_bytes
-> ORDER BY current_allocated desc\G
***** 1. row *****
thread_id: 152
user: lj@127.0.0.1
current_count_used: 325
current_allocated: 36.00 GiB
current_avg_alloc: 113.43 MiB
current_max_alloc: 36.00 GiB
total_allocated: 37.95 GiB
...
```


Анализ переменных

Performance Schema выводит инструментарий переменных на новый уровень. Он предоставляет инструменты для анализа:

- серверных переменных:
 - глобальных;
 - сеансовых для всех открытых в данный момент соединений;
 - источников, из которых берутся все текущие значения переменных;
- переменных состояния:
 - глобальных;
 - сеансовых для всех открытых в данный момент соединений;
 - агрегирований:
 - по хосту;
 - имени пользователя;
 - учетной записи пользователя;
 - потоку;
- пользовательских переменных.



До версии 5.7 серверные переменные и переменные состояния были инструментированы в `information_schema`. Этот инструментарий был ограничен: он позволял отслеживать только глобальные и текущие значения сеанса. Информация о переменных и статусе в других сессиях, а также о пользовательских переменных была недоступна. Однако по соображениям обратной совместимости MySQL 5.7 применяет `information_schema` для отслеживания переменных. Чтобы включить поддержку `performance_schema` для переменных, необходимо установить для переменной конфигурации `show_compatibility_56` значение 0. Этого требования, а также таблицы переменных в `information_schema` не существует в версии 8.0.

Значения глобальных переменных хранятся в таблице `global_variables`. Переменные сеанса для текущего сеанса хранятся в таблице `session_variables`. Обе таблицы имеют только два столбца с понятными именами: `VARIABLE_NAME` и `VARIABLE_VALUE`.

В таблице `variable_by_thread` есть дополнительный столбец `THREAD_ID`, содержащий поток, к которому принадлежит переменная. Это позволяет вам находить потоки, которые устанавливают значения переменных сеанса, отличные от указанных в конфигурации по умолчанию.

В следующем примере поток с `THREAD_ID=84` устанавливает переменную `tx_isolation` в значение `SERIALIZABLE`, что может привести к ситуациям, когда транзакции получают больше блокировок, чем при использовании уровня по умолчанию:

```
mysql> SELECT * FROM variables_by_thread
-> WHERE VARIABLE_NAME='tx_isolation';
+-----+-----+-----+
| THREAD_ID | VARIABLE_NAME | VARIABLE_VALUE |
+-----+-----+-----+
|          71 | tx_isolation  | REPEATABLE-READ |
|          83 | tx_isolation  | REPEATABLE-READ |
|          84 | tx_isolation  | SERIALIZABLE     |
+-----+-----+-----+
3 rows in set, 3 warnings (0.00 sec)
```

В следующем примере найдены все потоки со значениями переменных сеанса, отличными от текущего активного сеанса:

```
mysql> SELECT vt2.THREAD_ID AS TID, vt2.VARIABLE_NAME,
-> vt1.VARIABLE_VALUE AS MY_VALUE,
-> vt2.VARIABLE_VALUE AS OTHER_VALUE
-> FROM performance_schema.variables_by_thread vt1
-> JOIN performance_schema.threads t USING(THREAD_ID)
-> JOIN performance_schema.variables_by_thread vt2
-> USING(VARIABLE_NAME)
-> WHERE vt1.VARIABLE_VALUE != vt2.VARIABLE_VALUE
-> AND t.PROCESSLIST_ID=@pseudo_thread_id;
+-----+-----+-----+-----+
| TID | VARIABLE_NAME | MY_VALUE | OTHER_VALUE |
+-----+-----+-----+-----+
| 42 | max_allowed_packet | 67108864 | 1073741824 |
| 42 | pseudo_thread_id | 22715 | 5 |
| 42 | timestamp | 1626650242.678049 | 1626567255.695062 |
| 43 | gtid_next | AUTOMATIC | NOT_YET_DETERMINED |
| 43 | pseudo_thread_id | 22715 | 6 |
| 43 | timestamp | 1626650242.678049 | 1626567255.707031 |
+-----+-----+-----+-----+
6 rows in set (0.01 sec)
```

Глобальные и текущие значения состояния сеанса хранятся в таблицах `global_status` и `session_status` соответственно. У них также есть всего два столбца: `VARIABLE_NAME` и `VARIABLE_VALUE`.

Переменные состояния могут быть агрегированы по учетной записи пользователя, хосту, имени пользователя и потоку. На мой взгляд, наиболее интересной является агрегация по потокам, поскольку она позволяет быстро определить, какое соединение создает наибольшую нагрузку на ресурсы сервера. Например,

следующий фрагмент ясно показывает, что соединение с `THREAD_ID=83` выполняет большую часть операций записи:

```
mysql> SELECT * FROM status_by_thread
      -> WHERE VARIABLE_NAME='Handler_write';
+-----+-----+-----+
| THREAD_ID | VARIABLE_NAME | VARIABLE_VALUE |
+-----+-----+-----+
|          71 | Handler_write | 94              |
|          83 | Handler_write | 4777777777      | -- Most writes
|          84 | Handler_write | 101             |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

Пользовательские переменные создаются оператором `SET @my_var = 'foo'` и отслеживаются в таблице `user_variables_by_thread`:

```
mysql> SELECT * FROM user_variables_by_thread;
+-----+-----+-----+
| THREAD_ID | VARIABLE_NAME | VARIABLE_VALUE |
+-----+-----+-----+
|          71 | baz           | boo            |
|          84 | foo           | bar            |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

Этот инструментарий полезен, когда вам нужно найти источники потребления памяти, потому что каждая переменная занимает байты для хранения своих значений. Вы также можете использовать эту информацию для решения сложных проблем с постоянными соединениями, задействуя пользовательские переменные. И последнее, но не менее важное: эта таблица — единственный способ узнать, какие переменные вы определили в своем сеансе.

Таблица `variable_info` не содержит значений переменных. Скорее всего, в ней содержится информация о том, откуда возникли переменные сервера и другая документация, такая как минимальные и максимальные значения переменных по умолчанию. Столбец `SET_TIME` содержит временную метку последнего изменения переменной. Столбцы `SET_HOST` и `SET_USER` идентифицируют учетную запись пользователя, которая установила переменную. Например, чтобы найти все переменные, которые были динамически изменены с момента запуска сервера, выполните:

```
mysql> SELECT * FROM performance_schema.variables_info
      -> WHERE VARIABLE_SOURCE = 'DYNAMIC'\G
***** 1. row *****
VARIABLE_NAME: foreign_key_checks
VARIABLE_SOURCE: DYNAMIC
VARIABLE_PATH:
```

```

MIN_VALUE: 0
MAX_VALUE: 0
SET_TIME: 2021-07-18 03:14:15.560745
SET_USER: NULL
SET_HOST: NULL
***** 2. row *****
VARIABLE_NAME: sort_buffer_size
VARIABLE_SOURCE: DYNAMIC
VARIABLE_PATH:
  MIN_VALUE: 32768
  MAX_VALUE: 18446744073709551615    SET_TIME: 2021-07-19 02:37:11.948190
  SET_USER: sveta
  SET_HOST: localhost
2 rows in set (0,00 sec)

```

Возможные значения `VARIABLE_SOURCE` включают:

- `COMMAND_LINE` — набор переменных в командной строке;
- `COMPILED` — скомпилированное значение по умолчанию;
- `PERSISTED` — задается из файла параметров `mysqld-auto.cnf` для конкретного сервера.

Существует также множество опций для переменных, установленных в разных файлах опций. Я не буду обсуждать их все: либо они носят самоописательный характер, либо информацию о них можно легко посмотреть в справочном руководстве пользователя. Количество тонкостей также увеличивается от версии к версии.

Анализ наиболее частых ошибок

В дополнение к конкретной информации об ошибках представление `performance_schema` предоставляет сводные таблицы, объединяющие ошибки по пользователям, хостам, учетным записям, потокам и глобально по номеру ошибки. Все таблицы агрегирования имеют структуру, аналогичную той, что применяется в таблице `events_errors_summary_global_by_error`:

```

mysql> USE performance_schema;
mysql> SHOW CREATE TABLE events_errors_summary_global_by_error\G
***** 1. row *****
      Table: events_errors_summary_global_by_error
Create Table: CREATE TABLE `events_errors_summary_global_by_error` (
  `ERROR_NUMBER` int DEFAULT NULL,
  `ERROR_NAME` varchar(64) DEFAULT NULL,
  `SQL_STATE` varchar(5) DEFAULT NULL,
  `SUM_ERROR_RAISED` bigint unsigned NOT NULL,
  `SUM_ERROR_HANDLED` bigint unsigned NOT NULL,

```

```
`FIRST_SEEN` timestamp NULL DEFAULT '0000-00-00 00:00:00',
`LAST_SEEN` timestamp NULL DEFAULT '0000-00-00 00:00:00',
UNIQUE KEY `ERROR_NUMBER` (`ERROR_NUMBER`)
) ENGINE=PERFORMANCE_SCHEMA DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0,00 sec)
```

Столбцы `ERROR_NUMBER`, `ERROR_NAME` и `SQL_STATE` идентифицируют ошибку. `SUM_ERROR_RAISED` показывает, сколько раз возникала ошибка, `SUM_ERROR_HANDLED` — сколько раз она была обработана. `FIRST_SEEN` и `LAST_SEEN` — это метки времени, когда ошибка была обнаружена в первый и последний раз.

Конкретные сводные таблицы имеют дополнительные столбцы. Таким образом, в таблице `events_errors_summary_by_thread_by_error` есть столбец `THREAD_ID`, который идентифицирует поток, вызвавший ошибку, в таблице `events_errors_summary_by_host_by_error` есть столбец `HOST` и т. д.

Например, чтобы найти все учетные записи, которые выполняли операторы, вызвавшие ошибки более десяти раз, выполните:

```
mysql> SELECT * FROM
-> performance_schema.events_errors_summary_by_account_by_error
-> WHERE SUM_ERROR_RAISED > 10 AND USER IS NOT NULL
-> ORDER BY SUM_ERROR_RAISED DESC\G
***** 1. row *****
USER: sveta
HOST: localhost
ERROR_NUMBER: 3554
ERROR_NAME: ER_NO_SYSTEM_TABLE_ACCESS
SQL_STATE: HY000
SUM_ERROR_RAISED: 60
SUM_ERROR_HANDLED: 0
FIRST_SEEN: 2021-07-18 03:14:59
LAST_SEEN: 2021-07-19 02:50:13
1 row in set (0,01 sec)
```

Таблицы дайджеста ошибок могут быть полезны для определения того, какие учетные записи пользователей, хосты, пользователи или потоки отправляют самые ошибочные запросы и выполняют какое-либо действие. Они также могут помочь разобраться с такими ошибками, как `ER_DEPRECATED_UTF8_ALIAS`, которые могут указывать на то, что некоторые из часто используемых запросов были написаны для предыдущих версий MySQL и их необходимо обновить.

Анализ самой Performance Schema

Вы можете проанализировать саму Performance Schema, задействуя те же инструменты и потребители, что и для ваших собственных схем. Просто обратите внимание на то, что по умолчанию, если в качестве базы данных по умолчанию

задана `performance_schema`, запросы к ней не отслеживаются. Если вам нужно проанализировать запросы к `performance_schema`, следует сначала обновить таблицу `setup_actors`.

Как только таблица `setup_actors` будет обновлена, можно использовать все инструменты. Например, чтобы найти в `performance_schema` десять потребителей, которые выделили большую часть памяти, выполните запрос:

```
mysql> SELECT SUBSTRING_INDEX(EVENT_NAME, '/', -1) AS EVENT,
-> CURRENT_NUMBER_OF_BYTES_USED/1024/1024 AS CURRENT_MB,
-> HIGH_NUMBER_OF_BYTES_USED/1024/1024 AS HIGH_MB
-> FROM performance_schema.memory_summary_global_by_event_name
-> WHERE EVENT_NAME LIKE 'memory/performance_schema/%'
-> ORDER BY CURRENT_NUMBER_OF_BYTES_USED DESC LIMIT 10;
```

EVENT	CURRENT_MB	HIGH_MB
events_statements_summary_by_digest	39.67285156	39.67285156
events_statements_history_long	13.88549805	13.88549805
events_errors_summary_by_thread_by...	11.81640625	11.81640625
events_statements_summary_by_thread...	9.79296875	9.79296875
events_statements_history_long.dige...	9.76562500	9.76562500
events_statements_summary_by_digest...	9.76562500	9.76562500
events_statements_history_long.sql...	9.76562500	9.76562500
memory_summary_by_thread_by_event_name	7.91015625	7.91015625
events_errors_summary_by_host_by_error	5.90820313	5.90820313
events_errors_summary_by_account_by...	5.90820313	5.90820313

10 rows in set (0,00 sec)

Или используйте схему `sys`:

```
mysql> SELECT SUBSTRING_INDEX(event_name, '/', -1), current_alloc
-> FROM sys.memory_global_by_current_bytes
-> WHERE event_name LIKE 'memory/performance_schema/%' LIMIT 10;
```

SUBSTRING_INDEX(event_name, '/', -1)	current_alloc
events_statements_summary_by_digest	39.67 MiB
events_statements_history_long	13.89 MiB
events_errors_summary_by_thread_by_error	11.82 MiB
events_statements_summary_by_thread_by_event_name	9.79 MiB
events_statements_history_long.digest_text	9.77 MiB
events_statements_summary_by_digest.digest_text	9.77 MiB
events_statements_history_long.sql_text	9.77 MiB
memory_summary_by_thread_by_event_name	7.91 MiB
events_errors_summary_by_host_by_error	5.91 MiB
events_errors_summary_by_account_by_error	5.91 MiB

10 rows in set (0,00 sec)

performance_schema также поддерживает оператор SHOW ENGINE PERFORMANCE_SCHEMA STATUS:

```
mysql> SHOW ENGINE PERFORMANCE_SCHEMA STATUS\G
***** 1. row *****
Type: performance_schema
Name: events_waits_current.size
Status: 176
***** 2. row *****
Type: performance_schema
Name: events_waits_current.count
Status: 1536
***** 3. row *****
Type: performance_schema
Name: events_waits_history.size
Status: 176
***** 4. row *****
Type: performance_schema
Name: events_waits_history.count
Status: 2560
...
***** 244. row *****
Type: performance_schema
Name: (pfs_buffer_scalable_container).count
Status: 17
***** 245. row *****
Type: performance_schema
Name: (pfs_buffer_scalable_container).memory
Status: 1904
***** 246. row *****
Type: performance_schema
Name: (max_global_server_errors).count
Status: 4890
***** 247. row *****
Type: performance_schema
Name: (max_session_server_errors).count
Status: 1512
***** 248. row *****
Type: performance_schema
Name: performance_schema.memory
Status: 218456400
248 rows in set (0,00 sec)
```

В его выходных данных вы найдете такие сведения, как количество конкретных событий, хранящихся в потребителях, или максимальные значения конкретных показателей. Последняя строка содержит количество байтов, которое в данный момент занимает Performance Schema.

Резюме

Performance Schema — это функция, которую часто критикуют. Более ранние версии MySQL имели далеко не оптимальные реализации, что приводило к высокому потреблению ресурсов. Поэтому был общий совет просто отключить ее.

Она также считалась трудной для понимания. Включение инструмента — это просто включение небольшого дополнительного кода на сервере, который записывает данные и отправляет их потребителям. Потребители — это просто таблицы, которые хранятся в памяти, и вам нужно использовать стандартный SQL, чтобы задать таблице правильные вопросы, для того чтобы что-то найти. Понимание того, как Performance Schema управляет собственной памятью, поможет вам понять, что в MySQL нет утечки памяти — он просто хранит потребительские данные в памяти и освобождает ее только при перезапуске.

Мой совет прост: вы должны оставлять Performance Schema включенной, динамически включая инструменты и потребители, помогающие решить любые проблемы, которые могут у вас возникнуть: производительность запросов, блокировки, дисковый ввод/вывод, ошибки и многое другое. Вы также должны использовать схему `sys`, как кратчайший путь для быстрого решения наиболее распространенных проблем. Это обеспечит общедоступный способ измерения производительности непосредственно из MySQL.

ГЛАВА 4

Оптимизация операционной системы и оборудования

Ваш сервер MySQL способен работать настолько хорошо, насколько хорошо работает его самое слабое звено, при этом операционная система и оборудование, на котором он действует, зачастую являются ограничивающими факторами. Емкость диска, доступные ресурсы памяти и процессора, сеть и компоненты, которые их связывают, — все это может ограничивать общую производительность системы. Таким образом, вам необходимо тщательно выбирать для себя оборудование и соответствующим образом настраивать его и операционную систему. Например, если узким местом в существующей рабочей нагрузке является ввод/вывод, один из приемов его расширения заключается в перепроектировании приложения так, чтобы минимизировать рабочую нагрузку на ввод/вывод MySQL. Однако зачастую разумнее обновить подсистему ввода/вывода, установить дополнительную память или переконфигурировать имеющиеся диски. Если вы работаете в облачной среде, информация, приведенная в этой главе, тоже может оказаться очень полезной, особенно для понимания ограничений файловой системы и планировщиков ввода/вывода Linux.

Что ограничивает производительность MySQL

На производительность MySQL могут влиять многие аппаратные компоненты, но наиболее часто узким местом оказывается перегрузка процессора. Это может произойти, когда MySQL пытается выполнить параллельно слишком много запросов или когда меньшее количество запросов выполняется на процессоре слишком долго.

Перегрузка ввода/вывода до сих пор может возникать, но гораздо реже, чем перегрузка процессора. Во многом это связано с переходом на использование твердотельных накопителей (SSD). Исторически так сложилось, что снижение

производительности из-за невозможности целиком поместить данные в оперативной памяти и необходимости считывания с жесткого диска (HDD) было весьма значительным. SSD обычно в 10–20 раз быстрее, чем SSH. Сейчас, если запросы требуют чтения данных с диска, вы все равно получите для них приемлемую производительность.

Исчерпание памяти все еще может произойти, но обычно только тогда, когда вы пытаетесь выделить слишком много памяти для MySQL. Мы поговорим об оптимальных параметрах конфигурации для предотвращения этого в разделе «Настройка использования памяти» в главе 5.

Как выбрать процессоры для MySQL

При модернизации существующего или покупке нового оборудования необходимо решить, ограничена ли ваша рабочая нагрузка возможностями процессора. Вы можете определить рабочую нагрузку, связанную с процессором, проверив загрузку процессора, но не смотрите только на то, сильно ли загружены ваши ЦП в целом, а попытайтесь изучить баланс использования процессора и подсистемы ввода/вывода для наиболее важных запросов и обратите особое внимание на равномерность загрузки процессоров.

Вообще говоря, для вашего сервера могут существовать две цели:

- *низкая задержка (быстрое время отклика)*. Для ее достижения потребуются быстрые процессоры, поскольку каждый запрос будет выполняться только на одном процессоре;
- *высокая пропускная способность*. Если одновременно выполняется много запросов, то их обслуживание можно ускорить, увеличив количество процессоров.

Если рабочая нагрузка не использует все ваши процессоры, MySQL все равно может задействовать дополнительные процессоры для фоновых задач, таких как очистка буферов InnoDB, сетевые операции и т. д. Однако эти задания обычно второстепенны по сравнению с выполнением запросов.

Балансировка памяти и дисковых ресурсов

Иметь много памяти важно прежде всего не для того, чтобы уместить в ней как можно больше данных, а для того, чтобы избежать дисковых операций ввода/вывода, которые на несколько порядков медленнее, чем доступ к данным в памяти. Хитрость заключается в том, чтобы найти правильный баланс между объемом

оперативной и дисковой памяти, быстродействием, затратами и другими характеристиками, которые обеспечили бы высокую производительность при данной рабочей нагрузке.

Кэширование, чтение и запись

Если у вас достаточно памяти, можете полностью изолировать диск от запросов на чтение. Если все данные помещаются в память, каждое чтение будет попаданием в кэш после того, как кэш сервера будет разогрет. По-прежнему станут выполняться логические операции чтения из памяти, но не будет физических операций чтения с диска. Однако операции записи — это совсем другое дело. Запись может выполняться в памяти так же, как и чтение, но рано или поздно она должна быть занесена на диск, чтобы стать постоянной. Другими словами, кэш может откладывать запись, но кэширование не может исключить операции записи, как происходит при чтении.

Фактически кэширование позволяет не только откладывать записи, но и группировать их вместе двумя важными способами.

- *Много операций записи, одна операция сброса.* Один фрагмент данных может быть многократно изменен в памяти без записи всех новых значений на диск. Когда данные в конечном итоге сбрасываются на диск, все изменения, произошедшие с момента последней физической записи, становятся постоянными. Например, многие операторы могут обновлять счетчик в памяти. Если он увеличивается 100 раз, а затем записывается на диск, 100 модификаций группируются в одну физическую запись.
- *Объединение операций ввода/вывода.* В памяти может быть изменено множество различных элементов данных, и эти модификации могут быть собраны вместе, поэтому физическую запись можно выполнять как одну дисковую операцию.

Именно поэтому во многих транзакционных системах применяется *упреждающая запись в журнал*. Ведение журнала с упреждающей записью позволяет вносить изменения на страницы, хранящиеся в оперативной памяти, не сбрасывая изменения на диск, так как это потребовало бы операций ввода/вывода с произвольным доступом, что выполняется очень медленно. Вместо этого протокол изменений записывают в последовательный файл журнала — данная операция выполняется гораздо быстрее. Впоследствии фоновый поток может записать модифицированные страницы на диск, причем попутно оптимизировать запись.

Запись значительно выигрывает от буферизации, поскольку она преобразует произвольный ввод/вывод в последовательный. Асинхронные (буферизованные) записи обычно группируются и обрабатываются операционной системой,

которая может производить пакетный сброс на диск наиболее оптимальным образом. Синхронные (небуферизованные) записи должны быть записаны на диск до их завершения. Именно поэтому такой выигрыш дает буферизация на уровне RAID-контроллера, оборудованного кэшем записи с резервным электропитанием (мы обсудим технологию RAID в дальнейшем).

Каково ваше рабочее множество

У каждого приложения есть рабочее множество данных, то есть данные, которые ему действительно нужны для работы. В большинстве баз данных имеется также большое количество данных, которые не входят в рабочее множество. Вы можете представить себе базу данных как стол с выдвижными ящиками для документов. Рабочее множество состоит из тех документов, которые необходимо иметь на рабочем столе для выполнения работы. В этой аналогии поверхность рабочего стола — это оперативная память, а ящики для документов — жесткие диски. Точно так же, как вам не нужно, чтобы все листы бумаги лежали на рабочем столе, чтобы выполнить работу, для достижения оптимальной производительности не требуется, чтобы вся база данных помещалась в памяти, — достаточно рабочего множества.

При работе с жесткими дисками рекомендуется попытаться найти эффективное соотношение объема памяти к объему диска. Во многом это связано с более низкой задержкой и малым количеством операций ввода/вывода в секунду (IOPS) для жестких дисков. При использовании твердотельных накопителей соотношение объема памяти и объема диска становится гораздо менее важным.

Твердотельные хранилища данных

Твердотельные (флеш-) хранилища являются стандартом для большинства систем баз данных, особенно систем оперативной обработки транзакций (OLTP). Жесткие диски обычно можно найти только в очень больших хранилищах данных или в устаревших системах. Это изменение произошло, когда цены на твердотельные накопители значительно упали, — примерно в 2015 году.

В твердотельных запоминающих устройствах вместо магнитных пластин используются микросхемы энергонезависимой флеш-памяти, состоящие из ячеек. Их также называют *энергонезависимой оперативной памятью (NVRAM)*. У них нет движущихся частей, поэтому они ведут себя совсем не так, как жесткие диски.

Вот краткий обзор характеристик флеш-памяти. Высококачественные флеш-устройства имеют:

- *гораздо лучшую производительность произвольного чтения и записи по сравнению с жесткими дисками.* Флеш-устройства обычно немного лучше справляются с чтением, чем с записью;
- *более высокую производительность последовательного чтения и записи по сравнению с жесткими дисками.* Однако это не такое значительное улучшение, как при произвольном вводе/выводе, поскольку жесткие диски работают гораздо медленнее при произвольном вводе/выводе, чем при последовательном;
- *гораздо лучшую поддержку параллелизма, чем у жестких дисков.* Флеш-устройства могут поддерживать намного больше одновременных операций и не достигают максимальной пропускной способности, пока у вас не будет обеспечен значительный параллелизм.

Самое важное здесь — улучшение произвольного ввода/вывода и параллелизма. Флеш-память обеспечивает очень хорошую производительность произвольного ввода/вывода при высокой степени параллелизма.

Обзор флеш-памяти

У жестких дисков с вращающимися пластинами и колеблющимися головками существовали ограничения, а их характеристики являлись следствием задействованной физики. То же самое относится и к твердотельному накопителю, который построен поверх флеш-памяти. Не думайте, что твердотельные накопители — это просто. В некотором смысле они на самом деле сложнее, чем жесткие диски. Ограничения флеш-памяти довольно серьезны, и их трудно преодолеть, поэтому типичное твердотельное устройство имеет сложную архитектуру с множеством абстракций, кэширования и фирменной магии.

Наиболее важной характеристикой флеш-памяти является то, что ее можно считывать много раз быстро и небольшими блоками, но запись — гораздо более сложная задача. Ячейка не может быть перезаписана без специальной операции стирания, а стирать можно только большими блоками, например 512 Кбайт. Цикл стирания протекает медленно и в итоге приводит к износу блока. Количество циклов стирания, которые может выдержать блок, зависит от базовой технологии, которую он использует (подробнее об этом позже).

Ограничения на запись являются причиной сложности твердотельных хранилищ. Вот почему одни устройства обеспечивают устойчивую и стабильную работу, а другие — нет. Вся магия заключается в фирменном микропрограммном обеспечении, драйверах и других компонентах, которые обеспечивают работу твердотельного устройства. Чтобы операции записи хорошо выполнялись и можно было избежать преждевременного износа блоков флеш-памяти, устрой-

ство должно иметь возможность перемещать страницы, выполнять сборку мусора и так называемое выравнивание износа. Термин «усиление записи» используется для описания дополнительных операций записи, вызванных перемещением данных с места на место, многократной записью данных и метаданных из-за частичной записи блоков.

Сборка мусора

Важно понимать, что такое сборка мусора. Чтобы сохранить некоторые блоки свежими и готовыми к новым записям, устройство восстанавливает блоки. Для этого требуется, чтобы на устройстве имелось свободное место. Либо на устройстве будет зарезервировано определенное внутреннее пространство, которое вы не сможете видеть, либо нужно зарезервировать пространство самостоятельно, не заполняя его целиком, — подход варьируется от устройства к устройству. В любом случае по мере заполнения устройства сборщику мусора придется прилагать больше усилий, чтобы поддерживать чистоту некоторых блоков, поэтому коэффициент усиления записи увеличивается.

В результате многие устройства становятся медленнее по мере заполнения. Насколько медленнее, зависит от конкретного производителя, модели и архитектуры устройства. Некоторые из них рассчитаны на высокую производительность, даже когда довольно плотно заполнены, но в целом файл размером 100 Гбайт будет работать иначе на твердотельном накопителе емкостью 160 Гбайт, чем на твердотельном накопителе емкостью 320 Гбайт. Замедление вызвано необходимостью ожидать завершения стирания, когда свободных блоков нет. Запись в свободный блок занимает пару сотен микросекунд, но стирание происходит намного медленнее — обычно несколько миллисекунд.

Оптимизация производительности с помощью RAID

Подсистемы хранения часто размещают все данные и индексы в отдельных больших файлах, что означает: RAID (redundant array of inexpensive disks — избыточный массив независимых дисков) обычно является наиболее приемлемым вариантом для хранения большого количества данных. RAID может решить проблемы резервирования, емкости, кэширования и быстродействия. Но как и в случае с другими оптимизациями, которые мы рассматривали, существует множество вариантов конфигураций RAID, и важно выбрать уровень, который соответствует именно вашим потребностям.

Мы не будем здесь рассматривать каждый уровень RAID или вдаваться в подробности того, как именно организовано хранение данных на разных уровнях. Вместо этого сосредоточимся на том, как различные конфигурации RAID удовлетворяют требованиям, предъявляемым серверами базы данных. Вот наиболее важные уровни RAID.

- **RAID 0.** Это самая малозатратная и наиболее производительная конфигурация RAID, по крайней мере при упрощенном подходе к оценке затрат и производительности (например, если рассматривать также восстановление данных, затраты начинают расти). Поскольку этот уровень не обеспечивает никакой избыточности, вряд ли RAID 0 когда-либо подойдет для производственной базы данных, но если вы действительно хотите сэкономить, он может стать выбором в средах разработки, где полный отказ сервера не является инцидентом.

Опять же обратите внимание: RAID 0 не обеспечивает никакой избыточности, даже несмотря на то, что резервирование — это буква R в аббревиатуре RAID. На самом деле вероятность отказа массива RAID 0 даже выше, чем вероятность отказа любого отдельного диска!

- **RAID 1.** Уровень RAID 1 обеспечивает неплохую производительность чтения во многих ситуациях, а так как данные дублируются, то имеется и избыточность. При операциях чтения RAID 1 чуть быстрее, чем RAID 0. Он хорошо подходит для серверов, которые обрабатывают ведение журналов и аналогичные рабочие нагрузки, потому что для 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 часто выбирают для недорогих серверов, которым нужна избыточность, но которые имеют только два жестких диска.

RAID 0 и RAID 1 очень просты, и их часто можно хорошо реализовать программно. Большинство операционных систем предоставляют простые средства для создания программных томов типа RAID 0 и RAID 1.

- **RAID 5.** Раньше использования RAID 5 для систем баз данных старались избегать, в основном из-за влияния на производительность. Поскольку твердотельные накопители становятся обычным явлением, теперь это вполне приемлемый вариант. RAID 5 распределяет данные по множеству дисков с распределенными блоками четности, так что в случае отказа любого диска данные могут быть восстановлены из блоков четности. Если два диска выйдут из строя, весь том выйдет из строя и его будет невозможно восстановить. С точки зрения затрат на единицу хранения это самая экономичная конфигурация с резервированием, поскольку во всем массиве вы теряете место для хранения только на одном диске.

Самая большая проблема с RAID 5 заключается в том, как работает массив в случае сбоя диска. Это связано с тем, что данные должны восстанавливаться путем чтения всех других дисков, что сильно влияло на производительность жесткого диска, поэтому прежде RAID 5 обычно не одобрялся. Проблема усугубляется, если используется много дисков. Если вы пытаетесь сохранить сервер в режиме онлайн во время перестроения, не ожидайте, что перестроение или производительность массива будут хорошими. Другие потери производительности включали ограниченную масштабируемость из-за блоков четности — RAID 5 плохо масштабируется более чем на десять дисков или около того — и проблемы с кэшированием. Хорошая производительность RAID 5 в значительной степени зависит от кэша RAID-контроллера, что может противоречить потребностям сервера баз данных. Как мы упоминали ранее, твердотельные накопители обеспечивают значительно лучшую производительность с точки зрения операций ввода/вывода и пропускной способности, также в них устранены проблемы с низкой производительностью произвольного чтения/записи.

Один из факторов, помогающих примириться с уровнем RAID 5, — это его популярность. Поэтому контроллеры RAID часто сильно оптимизированы для RAID 5, и, несмотря на теоретические ограничения, интеллектуальные контроллеры, которые хорошо используют кэши, иногда могут работать почти так же хорошо, как контроллеры RAID 10 для некоторых рабочих нагрузок. Правда, это может всего лишь свидетельствовать о том, что контроллеры RAID 10 менее оптимизированы, но независимо от причины мы с таким сталкивались.

- **RAID 6.** Самая большая проблема с RAID 5 заключалась в том, что потеря двух дисков была катастрофической. Чем больше дисков в вашем массиве, тем выше вероятность отказа диска. RAID 6 помогает снизить вероятность сбоя, добавляя второй диск четности. Это позволяет выдержать два сбоя диска и при этом восстановить массив. Недостатком является то, что вычисление дополнительной четности приведет к замедлению записи по сравнению с RAID 5.
- **RAID 10.** Это очень хороший выбор для хранения данных. Он состоит из массива зеркальных пар жестких дисков с данными, записанными с чередованием, поэтому хорошо масштабируется как при чтении, так и при записи. По сравнению с RAID 5 он работает и реконструируется очень быстро. Его также можно довольно хорошо реализовать программно.

Снижение производительности при выходе из строя одного жесткого диска все равно может быть значительным, поскольку узким местом становятся находящиеся на нем данные. 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. Это неоптимальное решение, так как в этом

случае отсутствует чередование: может оказаться, что данные, к которым сервер обращается чаще всего, размещены только на одной паре дисков, а не распределены по многим парам, поэтому вы получите низкую производительность.

- **RAID 50.** Состоит из чередующихся массивов RAID 5 и может стать хорошим компромиссом между экономичностью RAID 5 и высокой производительностью RAID 10, если у вас достаточное количество дисков. Этот метод наиболее полезен для очень больших наборов данных, таких как хранилища данных или чрезвычайно большие OLTP-системы.

В табл. 4.1 приведены сведения о различных конфигурациях RAID.

Таблица 4.1. Сравнение уровней RAID

Уровень	Характеристики	Избыточность	Требуется дисков	Быстрое чтение	Быстрая запись
RAID 0	Дешевый, быстрый, опасный	Нет	N	Да	Да
RAID 1	Быстрое чтение, простой, безопасный	Да	2 (обычно)	Да	Нет
RAID 5	Безопасный, быстрый, позволяющий достичь компромисса по затратам	Да	$N + 1$	Да	Зависит от реализации
RAID 6	Дорогой, быстрый, безопасный	Да	$N + 2$	Да	Зависит от реализации
RAID 10	Для очень больших хранилищ данных	Да	$2N$	Да	Да
RAID 50	Дешевый, быстрый, опасный	Да	N	Да	Да

Отказ, восстановление и мониторинг RAID

Все уровни RAID, кроме RAID 0, обеспечивают избыточность. Это, конечно, важно, но не стоит недооценивать вероятность одновременного отказа нескольких дисков.

Не стоит думать, что RAID дает полную гарантию сохранности данных.

RAID не отменяет и даже не уменьшает необходимость резервного копирования. В случае возникновения проблемы время восстановления будет зависеть от вашего контроллера, уровня RAID, размера массива, скорости диска и необходимости поддерживать сервер в рабочем состоянии во время перестроения массива.

Существует вероятность того, что диски выйдут из строя точно в одно и то же время. Например, скачок напряжения или перегрев могут легко привести к по-

вреждению двух или более дисков. Однако более распространены два сбоя дисков близко друг к другу. Многие такие проблемы могут остаться незамеченными. Распространенной причиной является повреждение физического носителя, содержащего данные, к которым редко обращаются. Это может оставаться незамеченным в течение нескольких месяцев, пока либо вы не попытаетесь прочитать данные, либо не выйдет из строя другой диск и RAID-контроллер не попытается использовать поврежденные данные для восстановления массива. Чем больше размер жесткого диска, тем больше вероятность этого.

Именно поэтому так важно вести мониторинг состояния RAID-массивов. Обычно вместе с большинством контроллеров поставляется программное обеспечение для формирования отчета о состоянии массива, и им надо пользоваться, поскольку в противном случае вы так и будете пребывать в неведении об отказе диска. Из-за этого можете упустить возможность восстановить данные и обнаружите проблему только тогда, когда выйдет из строя второй диск и будет слишком поздно. Следует настроить систему мониторинга так, чтобы она предупреждала, когда состояние диска или тома меняется на ухудшенное или неисправное.

Вы можете снизить риск скрытого повреждения, регулярно проверяя массивы на согласованность. 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, и может помочь предотвратить такие проблемы. Но как и в случае с восстановлением, проверка очень больших массивов может занимать длительное время, поэтому планируйте ее заблаговременно.

Можно также добавить горячий резервный диск, который не используется, а сконфигурирован как резервный, чтобы контроллер мог автоматически задействовать его для восстановления. Это хорошая идея, если для вас важна работоспособность каждого сервера. Такое решение дорогостоящее для серверов, которые имеют всего несколько жестких дисков, потому что стоимость простаивающего диска выше стоимости рабочих, но если у вас много дисков, просто глупо не иметь еще одного для горячей замены. Помните, что вероятность отказа диска быстро возрастает с увеличением количества дисков.

В дополнение к мониторингу дисков на предмет сбоев следует также контролировать блок резервного питания аккумулятора RAID-контроллера и политику кэширования записей. Если батарея выйдет из строя, по умолчанию большинство контроллеров отключат кэширование записи, изменив политику кэширования на Write-Through (запись в обход кэша) вместо Write-Back (отложенная запись в кэш). Это может привести к серьезному снижению производительности. Многие контроллеры также будут периодически заряжать батареи во время процесса обучения, в течение которого кэш-память также отключается. Утилита

управления вашего RAID-контроллера должна позволять просматривать и планировать цикл зарядки батарей, чтобы он не застал вас врасплох. Более новые RAID-контроллеры позволяют избежать этого, используя кэш с поддержкой флеш-памяти, который применяет NVRAM для хранения незафиксированных записей, вместо кэша с резервным питанием от батареи. Это избавит от боли, связанной с циклом обучения.

Возможно, вы также захотите протестировать свою систему с помощью политики кэширования, настроенной на сквозную запись, чтобы знать, чего ожидать. Предпочтительный подход заключается в том, чтобы запланировать циклы обучения батареи на периоды с низким трафиком, как правило ночь или выходные дни. Если производительность довольно сильно страдает из-за сквозной записи в любое время, можете переключиться на другой сервер до начала цикла обучения. В самом крайнем случае вы можете перенастроить свои серверы, изменив переменные настройки долговечности `innodb_flush_log_at_trx_commit` и `sync_binlog` на более низкие значения. Это уменьшит загрузку диска во время сквозной записи и может обеспечить приемлемую производительность, однако делать это следует в крайнем случае. Снижение долговечности серьезно влияет на объем данных, которые можно потерять во время сбоя базы данных, и на вашу способность их восстановить.

Конфигурация RAID и кэширование

Обычно для конфигурирования RAID-контроллера нужно войти в утилиту его настройки во время загрузки машины или запустить ее из командной строки. Хотя большинство контроллеров предлагают множество опций, мы сосредоточимся на двух: размере фрагмента для массивов с чередованием и *кэше контроллера* (его еще называют *RAID-кэшем*, 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 произвольной выборкой лучше подходит большой размер фрагмента, потому что это означает, что с одного диска может быть выполнено больше операций чтения.

Чтобы понять, почему так происходит, рассмотрим типичную операцию произвольного ввода/вывода для имеющейся рабочей нагрузки. Если размер фрагмента не меньше длины этой операции и данные не выходят за границы между фрагментами, то в чтении может участвовать только один диск. Но если размер фрагмента меньше, чем длины считываемых данных, нет способа избежать вовлечения в чтение более одного диска.

Но достаточно теории. На практике многие RAID-контроллеры плохо работают с большими фрагментами. Например, контроллер может использовать размер фрагмента в качестве единицы кэширования в своем кэше, что окажется расточительностью. Контроллер также может сопоставить размер фрагмента, размер кэша и размер единицы считывания количеству данных, которые он считывает за одну операцию. Если единица считывания оказывается слишком велика, его кэш может быть менее эффективным и в итоге он может считывать намного больше данных, чем ему действительно нужно, даже для крошечных запросов.

Кроме того, на практике сложно понять, будет ли тот или иной фрагмент данных расположен на одном или нескольких дисках. Даже если размер фрагмента составляет 16 Кбайт, что соответствует размеру страницы InnoDB, мы не можем быть уверены, что все операции считывания будут выровнены по границам 16 Кбайт. Файловая система может фрагментировать файл и обычно выравнивает его фрагменты по границам блока файловой системы, который чаще всего составляет 4 Кбайт. Некоторые файловые системы могут быть разумнее, но не стоит на это рассчитывать.

RAID-кэш

RAID-кэш— это довольно небольшая область памяти, которая физически находится на плате RAID-контроллера. Его можно применять для буферизации данных при их перемещении между дисками и хост-системой. Вот несколько из причин, по которым RAID-контроллер может воспользоваться кэшем.

- *Кэширование результатов чтения.* После того как контроллер прочитает некоторые данные с дисков и отправит их в хост-систему, он может сохранить их, что позволит ему удовлетворять будущие запросы на те же данные, не обращаясь к диску повторно.

Как правило, это неудачное применение кэша RAID. Почему? Потому что операционная система и сервер баз данных имеют свои кэши гораздо большего размера. При попадании в один из этих кэшей данные в кэше RAID использоваться не будут. И наоборот, если не было попадания ни в один из кэшей более высокого уровня, вероятность попадания в RAID-кэш исчезающе мала. Поскольку RAID-кэш намного меньше, он почти наверняка будет очищен и заполнен другими данными. В общем, как ни крути, кэширование операций чтения в RAID-кэше — пустая трата памяти.

- *Упреждающее кэширование данных при чтении.* Если RAID-контроллер обнаруживает запросы на чтение последовательных данных, он может прибегнуть к упреждающему чтению, то есть заранее выбрать данные, которые, по его прогнозам, скоро потребуются. Однако у него должно быть место для хранения данных, пока они не будут запрошены. Для этой цели можно

использовать RAID-кэш. Влияние такой тактики на производительность может варьироваться в широких пределах, поэтому следует убедиться, дала ли она что-нибудь в вашей ситуации. Операции упреждающего чтения могут не помочь, если сервер базы данных реализует собственный алгоритм интеллектуального упреждающего чтения, как это делает InnoDB. К тому же оно может воспрепятствовать гораздо более важному механизму буферизации синхронных операций записи.

- *Кэширование операций записи.* RAID-контроллер может буферизовать операции записи в своем кэше и планировать их выполнение на более позднее время. У такой методики есть два достоинства: во-первых, контроллер может быстрее вернуть хост-системе признак успешного завершения, чем если бы ему приходилось фактически выполнять запись на физические диски, и во-вторых, он может накопить записи и выполнить их более эффективно.
- *Внутренние операции.* Некоторые операции RAID-контроллера очень сложны, особенно записи в случае RAID 5: они должны вычислять биты четности, которые можно использовать для восстановления данных в случае сбоя. Контроллер должен задействовать некоторую память для этого типа внутренней операции. В частности, по этой причине уровень RAID 5 на некоторых контроллерах может работать медленно: для обеспечения высокой производительности контроллеру необходимо считывать много данных в кэш. 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, который нужно стараться использовать с умом. Применение его для кэширования результатов — обычно откровенное расточительство, но для кэширования операций записи — важный способ ощутимо повысить производительность ввода/вывода. Многие контроллеры позволяют вам выбирать, как распределять память. Например, можно выбрать, какую часть использовать для кэширования операций записи, а какую — для результатов чтения. В случае RAID 0, RAID 1 и RAID 10 вам, вероятно, лучше всего выделить 100 % памяти контроллера для кэширования операций записи. В случае RAID 5 вы должны зарезервировать часть памяти контроллера для его внутренних операций. В общем случае это хорошая рекомендация, но применима она не всегда: разные RAID-контроллеры конфигурируются по-разному.

При использовании RAID-кэша для кэширования записи многие контроллеры позволяют настроить допустимую задержку записи (1 с, 5 с и т. д.). Более длительная задержка означает, что больше записей может быть сгруппировано вместе и оптимально сброшено на диски. Недостаток же заключается в том, что запись оказывается более «пульсирующей». В этом нет ничего страшного, если только ваше приложение не делает кучу запросов на запись как раз в тот момент,

когда кэш контроллера заполнен и должен быть сброшен на диск. Если для запросов записи вашего приложения недостаточно места, ему придется подождать. 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. (Здесь мы несколько упрощаем — в контроллерах часто реализуются сложные патентованные алгоритмы балансировки, но мы пытаемся объяснить основополагающие принципы.)

Кэш записи очень полезен для синхронной записи, например для системных вызовов `fsync()` при записи в журнал транзакций и создания двоичных журналов в режиме `sync_binlog`, но его не следует активизировать, если ваш контроллер не оборудован блоком аварийного электропитания (battery backup unit, BBU) или другим энергонезависимым запоминающим устройством. Кэширование записей без BBU в случае отключения питания может привести к повреждению базы данных и даже транзакционной файловой системы. Однако, если у вас есть BBU, включение кэша записи может повысить производительность в 20 и более раз для рабочих нагрузок, если рабочая нагрузка подразумевает большое количество сбросов в журнал, например, в момент фиксации транзакций.

В завершение следует отметить, что многие жесткие диски имеют собственные кэши записи, которые могут «фальсифицировать» операции `fsync()`, обманывая контроллер сообщением, что данные были записаны на физический носитель. Жесткие диски, подключенные напрямую (в отличие от подключенных к RAID-контроллеру), иногда могут позволить операционной системе управлять своими кэшами, но это тоже не всегда работает. Такие кэши обычно сбрасываются при вызове `fsync()` и обходятся при синхронном вводе/выводе, но опять же жесткий диск может лгать. Необходимо либо убедиться, что кэш действительно сбрасывается в момент вызова `fsync()`, либо отключить их совсем, потому что аварийное питание от батареи для него не предусмотрено. Жесткие диски, которые не управляются должным образом операционной системой или микропрограммой RAID, во многих случаях приводили к потере данных.

По этой и другим причинам всегда рекомендуется проводить настоящее краш-тестирование (буквально выдергивая вилку из розетки) при установке нового оборудования. Зачастую это единственный способ найти трудноуловимые ошибки в конфигурации или скрытое поведение жесткого диска. Удобный скрипт для этого можно найти в Интернете.

Чтобы проверить, действительно ли вы можете положиться на BBU вашего RAID-контроллера, убедитесь, что шнур питания не подключен к сети в течение длительного времени. Некоторые батарейные источники поддерживают питание не так долго, как должны. В этом случае одно слабое звено может сделать бесполезной всю цепочку компонентов хранения данных.

Конфигурация сети

Задержка и пропускная способность являются ограничивающими факторами как для жесткого диска, так и для сетевого подключения. Самой большой проблемой для большинства приложений является задержка: типичное приложение выполняет много небольших сетевых передач, и небольшие задержки для каждой передачи складываются вместе.

Сеть, работающая плохо, является основным узким местом для производительности. Потеря пакетов — тоже распространенная проблема. Даже 1 % потерянных пакетов достаточно, чтобы вызвать значительное снижение производительности, поскольку различные уровни в стеке протоколов будут пытаться исправить проблемы, для чего ненадолго перейдут в режим ожидания, а затем начнут отправлять пакет повторно, затрачивая на все это лишнее время. Другая распространенная проблема — неправильно сконфигурированный или медленно работающий DNS-сервер.

DNS — это ахиллесова пята, поэтому на промышленных серверах целесообразно включать режим `skip_name_resolve`. Неработающий или медленно работающий DNS-сервер является проблемой для многих приложений, но для MySQL она особенно серьезна. Когда MySQL получает запрос на соединение, он выполняет как прямой, так и обратный поиск в DNS. По самым разным причинам поиск может завершиться неудачно. Сбой поиска в DNS приведет к отказу в соединении, замедлению процесса подключения к серверу и, как правило, к полному хаосу вплоть до атак типа «отказ в обслуживании» (DoS-атаки). Если вы включите опцию `skip_name_resolve`, то MySQL вообще не будет выполнять поиск DNS. Однако это также означает, что во всех учетных записях пользователей столбец `host` может содержать только IP-адрес (возможно, с метасимволами) или строку `localhost`. Пользователь, в учетной записи которого указано доменное имя, не сможет войти в систему.

Однако чаще более важно настроить параметры таким образом, чтобы они эффективно справлялись с большим количеством соединений и небольшими запросами. Одной из наиболее распространенных настроек является изменение диапазона локальных портов. Системы Linux имеют ряд локальных портов, которые можно использовать. При соединении с вызывающим абонентом задействуется локальный порт. Если у вас много соединений одновременно, то локальные порты могут закончиться. Вот как система конфигурируется по умолчанию:

```
$ cat /proc/sys/net/ipv4/ip_local_port_range
32768 61000
```

Иногда вам может потребоваться изменить эти значения на больший диапазон, например:

```
$ echo 1024 65535 > /proc/sys/net/ipv4/ip_local_port_range
```

Протокол TCP позволяет системе ставить входящие соединения в очередь. В чем-то это похоже на ведро: если «ведро» заполнится, клиенты не смогут подключиться. Вы можете позволить большему количеству подключений становиться в очередь следующим образом:

```
$ echo 4096 > /proc/sys/net/ipv4/tcp_max_syn_backlog
```

Для серверов баз данных, которые используются только локально, можно уменьшить величину тайм-аута после закрытия сокета в случае, если одноранговый узел не работает и не закрывает свою сторону соединения. По умолчанию в большинстве систем применяется 1 мин, что довольно долго:

```
$ echo <value> > /proc/sys/net/ipv4/tcp_fin_timeout
```

В большинстве случаев эти настройки можно оставить в качестве настроек по умолчанию. Изменять их имеет смысл только тогда, когда происходит что-то необычное, например, очень низка производительность сети или очень много подключений. Поиск в Интернете по запросу *TCP variables* («Параметры TCP») позволит найти много полезной информации об этих и многих других переменных.

Выбор файловой системы

Выбор файловой системы в значительной степени зависит от выбора операционной системы. Во многих системах, таких как Windows, действительно есть только один или два варианта и лишь один из них (NTFS) заслуживает внимания. В то же время GNU/Linux поддерживает множество файловых систем.

Многие хотят знать, какие файловые системы обеспечат наилучшую производительность для MySQL на GNU/Linux или, более конкретно, какой из вариантов лучше всего подходит для InnoDB. Эталонные тесты показывают, что большинство из них очень близки во многих отношениях и полагаться на файловую систему в вопросе производительности не стоит. Производительность файловой системы сильно зависит от рабочей нагрузки, и никакая файловая система не является панацеей. Как правило, каждая конкретная файловая система не будет работать значительно лучше или хуже, чем любая другая. Исключение составляет лишь случай, когда вы сталкиваетесь с некоторыми ограничениями файловой системы, например высокой конкурентностью, работой со многими файлами, ростом фрагментации и т. д.

В целом лучше всего использовать файловые системы с журналированием, такие как ext4, XFS или ZFS. В противном случае проверка файловой системы после сбоя может занять много времени.

Если вы задействуете ext3 или ее преемницу ext4, существует три варианта журналирования данных, выбрать которые вы можете в параметрах монтирования `/etc/fstab`.

- **data=writeback**. Этот параметр означает, что в журнал заносятся только записи метаданных. Запись метаданных не синхронизирована с записью данных. Это самая быстрая конфигурация, и *обычно* ее безопасно использовать, работая с InnoDB, поскольку у нее есть собственный журнал транзакций. Есть, правда, одно исключение: сбой в неподходящий момент может привести к повреждению файла `.frm` в версии MySQL до 8.0.

Приведем пример ситуации, когда эта конфигурация может вызвать проблемы. Предположим, программа решает расширить файл, чтобы увеличить его размер. Метаданные (размер файла) будут зарегистрированы и записаны в журнал до того, как данные будут фактически записаны в файл, который теперь стал больше. В результате в конце файла — в добавленной области — окажется мусор.

- **data=ordered**. Этот параметр задает режим, при котором в журнал также записываются только метаданные, но обеспечивается некоторая согласованность за счет того, что данные записываются раньше метаданных. Производительность лишь немного уступает производительности режима `writeback`, но в случае сбоя система ведет себя намного надежнее. Если в этом режиме, предположим, программа хочет расширить файл, метаданные файла не будут отражать его новый размер до тех пор, пока в новой расширенной области не будут записаны данные.
- **data=journal**. При этом значении параметра обеспечивается атомарное журналирование: перед записью данных в окончательное место назначения производится их атомарная запись в журнал. Обычно это излишняя процедура, и она имеет гораздо более высокие накладные расходы, чем два других варианта. Однако в некоторых случаях производительность даже увеличивается, поскольку ведение журнала позволяет файловой системе откладывать запись в конечное место хранения данных в файле.

Независимо от файловой системы существует ряд параметров, которые лучше отключить, потому что они не дают никаких преимуществ, но могут немного увеличить накладные расходы. Самый известный — это запись времени доступа, потому что операция записи выполняется даже тогда, когда вы просто читаете файл или каталог. Чтобы отключить эту опцию, добавьте в файл `/etc/fstab` параметры монтирования `noatime` и `nodirtime`. Иногда это может повысить

производительность на 5–10 % в зависимости от рабочей нагрузки и файловой системы (но может и не иметь большого значения). Приведем пример строки файла `/etc/fstab` для упомянутых нами параметров для файловой системы `ext3`:

```
/dev/sda2 /usr/lib/mysql ext3 noatime,nodiratime,data=writeback 0 1
```

Можно также настроить в файловой системе поведение при упреждающем чтении, потому что иногда оно может быть избыточным. Например, InnoDB предсказывает упреждающее чтение. Отключение или ограничение упреждающего чтения особенно полезно для Solaris UFS. Использование флага `innodb_flush_method=O_DIRECT` автоматически отключает упреждающее чтение.

Некоторые файловые системы могут не поддерживать необходимые возможности. Например, поддержка прямого ввода/вывода может быть важна, если вы используете метод сброса `O_DIRECT` для InnoDB. Кроме того, некоторые файловые системы лучше других обрабатывают большое количество базовых дисков. Например, XFS в этом отношении зачастую намного лучше, чем `ext3`. Наконец, если вы планируете использовать мгновенные снимки менеджера логических томов (LVM) для инициализации подчиненных серверов или создания резервных копий, следует убедиться, что выбранная вами файловая система и версия LVM хорошо работают вместе.

В табл. 4.2 приведены характеристики некоторых распространенных файловых систем.

Таблица 4.2. Общие характеристики файловых систем

Файловая система	Операционная система	Журналирование	Большие каталоги
ext3	GNU/Linux	По желанию	По желанию/частично
ext4	GNU/Linux	Есть	Есть
JFS (журналируемая файловая система)	GNU/Linux	Есть	Нет
NTFS	Windows	Есть	Есть
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	GNU/Linux, Solaris, FreeBSD	Есть	Есть

Обычно мы рекомендуем применять файловую систему XFS. Просто файловая система ext3 имеет слишком много серьезных ограничений, таких как единый мьютекс на каждый индексный дескриптор, а также использование функции `fsync()` для сброса всех «грязных» блоков данных всей файловой системы вместо «грязных» блоков только одного файла. Файловая система ext4 — это приемлемый выбор, хотя в некоторых версиях ядра имеются узкие места с производительностью, которые следует изучить, прежде чем переходить на нее.

При рассмотрении любой файловой системы для базы данных полезно учитывать, давно ли она применяется, насколько она зрелая и проверена ли в производственных средах. Данные файловой системы — это самый нижний уровень целостности данных в базе данных.

Выбор планировщика дисковых очередей

В GNU/Linux планировщик очередей определяет порядок, в котором запросы на устройство блочного ввода/вывода фактически отправляются на базовое устройство. По умолчанию применяется полноценная организация очереди, или `cfq` (completely fair queueing). Это нормально для временного использования на ноутбуках и настольных компьютерах, где помогает предотвратить нехватку ресурсов ввода/вывода, но ужасно для серверов. Из-за этого увеличивается время отклика при типичных рабочих нагрузках, которые генерирует MySQL, потому что она без необходимости останавливает некоторые запросы в очереди.

Вы можете посмотреть, какие планировщики доступны и какой из них активен, с помощью следующей команды:

```
$ cat /sys/block/sda/queue/scheduler
noop deadline [cfq]
```

Замените `sda` именем интересующего вас диска. В примере квадратные скобки указывают, какой планировщик используется для этого устройства. Два других варианта подходят для аппаратного обеспечения серверного класса и в большинстве случаев работают одинаково хорошо. Планировщик `noop` подходит для устройств, которые выполняют собственное планирование «за кулисами», таких как аппаратные RAID-контроллеры и сети хранения данных (SAN), а `deadline` подходит как для RAID-контроллеров, так и для подключаемых дисков. Эталонные тесты показывают очень небольшую разницу между этими двумя планировщиками. Основная рекомендация такова: планировщик может быть любым, но только не `cfq`, который способен вызвать серьезные проблемы с производительностью.

Память и подкачка

MySQL лучше всего работает при большом объеме выделенной памяти. Как мы узнали в главе 1, InnoDB использует память в качестве кэша, чтобы избежать доступа к диску. Это означает, что производительность системы памяти может напрямую влиять на скорость обслуживания запросов. Даже сегодня одним из лучших способов обеспечить более быстрый доступ к памяти является замена встроенного распределителя памяти (`glibc`) на внешний, такой как `tcmalloc` или `jemalloc`. Многочисленные тесты¹ показали, что оба они обеспечивают повышенную производительность и уменьшенную фрагментацию памяти по сравнению с `glibc`.

Подкачка происходит тогда, когда операционная система записывает некоторую область виртуальной памяти на диск, потому что у нее недостаточно физической памяти для ее хранения. Подкачка прозрачна для процессов, работающих в операционной системе. Только операционная система знает, где находится конкретный адрес виртуальной памяти, в физической памяти или на диске.

При использовании твердотельных накопителей снижение производительности уже не такое резкое, как при использовании жестких дисков. Вы по-прежнему должны активно сторониться подкачки — даже просто для того, чтобы избежать ненужных операций записи, которые могут сократить общий срок службы диска. Необходимо также рассмотреть возможность применения подхода без подкачки, который полностью исключает ее возможность, но создает риск того, что нехватка памяти может привести к завершению процесса.

В ОС GNU/Linux за подкачкой можно следить с помощью утилиты `vmstat` (приведем несколько примеров в следующем разделе). Активность ввода/вывода подкачки отображается в столбцах `si` и `so`, а не в столбце `swpd`, в котором показан объем использованного пространства в файле подкачки. Значение в столбце `swpd` может проинформировать о процессах, которые были загружены в память, но сейчас не работают, что на самом деле не проблематично. Желательно, чтобы значения столбцов `si` и `so` были равны 0, и определенно они должны быть меньше 10 блоков в секунду.

В редких случаях выделение слишком большого объема памяти может привести к тому, что операционной системе не хватит места в файле подкачки. Если это произойдет, возникающая в результате нехватка виртуальной памяти может вызвать аварийное завершение MySQL. Но даже если место в файле подкачки не исчерпано, очень активная подкачка может привести к тому, что вся операционная система перестанет отвечать на запросы, так что вы даже не сможете

¹ Сравнения см. в статьях [Impact of memory allocators on MySQL performance \(https://oreil.ly/AAJHX\)](https://oreil.ly/AAJHX) и [MySQL \(or percona\) memory usage tests \(https://oreil.ly/slp7v\)](https://oreil.ly/slp7v).

войти в систему и принудительно завершить процесс MySQL. Иногда ядро Linux может даже полностью зависнуть, когда ему не хватает места в файле подкачки. Мы рекомендуем запускать базы данных без использования пространства подкачки. Диск по-прежнему на порядок медленнее ОЗУ, и это позволяет избежать всех упомянутых здесь проблем.

Еще одно явление, которое часто возникает при экстремальной нагрузке на виртуальную память, заключается в том, что включается убийца процессов «отсутствие памяти» (out-of-memory, OOM) и прерывает другие процессы. Часто им является MySQL, но это может быть и другой процесс, например SSH. В этом случае вы останетесь без сетевого доступа к системе. Можно предотвратить эту проблему, установив значение `oom_adj` или `oom_score_adj` процесса SSH. Мы настоятельно рекомендуем: работая с выделенными серверами баз данных, идентифицируйте все ключевые процессы, такие как MySQL и SSH, и заблаговременно настраивайте показатель убийцы процессов OOM так, чтобы они не были выбраны первыми для принудительного завершения.

Большинство проблем, связанных с подкачкой, можно решить, правильно настроив буферы MySQL, но иногда система виртуальной памяти операционной системы все равно решает выгрузить соответствующий процесс. Иногда это связано с тем, как работает неоднородный доступ к памяти (NUMA)¹ в Linux. Обычно это происходит, когда операционная система видит, что MySQL выдает слишком много запросов на ввод/вывод, поэтому она пытается увеличить файловый кэш, чтобы он вмещал больше данных. Если памяти недостаточно, что-то нужно выгружать на диск, и этим чем-то вполне может оказаться сама MySQL. Некоторые более старые версии ядра Linux тоже имели контрпродуктивные приоритеты, из-за которых выгружалось то, что выгружать не нужно было, но в более поздних ядрах это исправлено.

Операционные системы обычно предоставляют какие-то средства контроля над виртуальной памятью и подсистемой ввода/вывода. Упомянем лишь некоторые инструменты, имеющиеся в GNU/Linux. Самый простой способ — изменить значение параметра `/proc/sys/vm/swappiness` на низкое значение, например 0 или 1. Это означает, что ядро не должно выполнять подкачку до тех пор, пока потребность в виртуальной памяти не станет критической. Например, вот как можно узнать текущее значение:

```
$ cat /proc/sys/vm/swappiness
60
```

Значение 60 — это параметр подкачки по умолчанию (значение может изменяться от 0 до 100). Это очень плохое значение по умолчанию для серверов.

¹ Дополнительную информацию можно найти в статье по адресу <https://oreil.ly/VGW65>.

Оно подходит только для ноутбуков. Для серверов должно быть установлено значение 0:

```
$ echo 0 > /proc/sys/vm/swappiness
```

Другой вариант заключается в том, чтобы изменить порядок чтения и записи данных подсистемой хранения. Например, установка режима `innodb_flush_method=O_DIRECT` снижает нагрузку на ввод/вывод. Прямой ввод/вывод не кэшируется, поэтому операционная система не считает его поводом для увеличения размера кэша. Этот параметр работает только для InnoDB.

Еще один способ — воспользоваться конфигурационным параметром `memlock` MySQL, который фиксирует MySQL в памяти. Это позволит избежать подкачки, но может быть опасным: если невыгружаемая память кончится, то MySQL может аварийно завершиться при попытке получить дополнительную память. Проблемы могут быть вызваны и тем, что заблокировано слишком много памяти, так что ее недостаточно для операционной системы.

Есть немало приемов, специфичных для конкретной версии ядра, поэтому будьте осторожны, особенно при обновлении версии ядра. При некоторых рабочих нагрузках бывает трудно заставить операционную систему вести себя разумно, и тогда единственным выходом может быть уменьшение размеров буфера до условно оптимальных значений.

Состояние операционной системы

Ваша операционная система, скорее всего, предоставляет инструменты, которые позволяют выяснить, что делают операционная система и оборудование. В этом разделе мы приведем примеры использования двух широко распространенных инструментов, `iostat` и `vmstat`. Если в вашей системе нет какого-либо из них, скорее всего, в ней существует что-то подобное. Таким образом, наша цель не в том, чтобы сделать вас экспертом в применении `iostat` или `vmstat`, мы хотим просто показать, на что нужно обращать внимание, пытаясь диагностировать проблемы с помощью подобных инструментов.

В дополнение к этим инструментам в вашей операционной системе могут быть и другие, такие как `mpstat` или `sar`. Если вас интересуют другие части системы, например сеть, можете вместо этого использовать утилиты `ifconfig` (которая среди прочего показывает, сколько сетевых ошибок произошло) или `netstat`.

По умолчанию `vmstat` и `iostat` выдают лишь один отчет, показывающий средние значения различных счетчиков с момента запуска сервера, — это не слишком полезная информация. Однако вы можете указать для обеих программ в качестве аргумента интервал времени. В этом случае генерируются инкрементные отчеты,

показывающие, что сервер делает в настоящий момент, что гораздо полезнее для настройки. (Первая строка показывает статистику с момента запуска системы, можете игнорировать ее.)

Как интерпретировать выдачу `vmstat`

Сначала рассмотрим пример работы `vmstat`. Чтобы выводить новый отчет каждые 5 с с указанием размеров в мегабайтах, выполните следующую команду:

```
$ vmstat -SM 5
procs -----memory----- -swap- -----io----- ---system--- -----cpu-----
 r  b swpd free buff cache  si so      bi    bo      in      cs us sy id wa st
11  0    0 2410   4 57223   0  0  9902 35594 122585 150834 10  3 85  1  0
10  2    0 2361   4 57273   0  0 23998 35391 124187 149530 11  3 84  2  0
```

Чтобы остановить `vmstat`, нажмите клавиши `Ctrl+C`. Вид отчета зависит от операционной системы, поэтому для получения точной информации потребуется прочитать страницу руководства.

Как указывалось ранее, несмотря на то что мы запрашивали инкрементный вывод, в первой строке значений показаны средние значения с момента загрузки сервера. Значения во второй строке отражают текущие показатели, а последующие строки показывают, что происходит с пятисекундными интервалами. Столбцы объединены в следующие группы:

- *procs*. В столбце **r** показано, сколько процессов ожидают выделения процессорного времени. А в столбце **b** — сколько из них находятся в непрерываемом режиме ожидания, что обычно означает, что они ожидают завершения операций ввода/вывода (дискового, сетевого, ввода информации пользователем и т. д.);
- *memory*. В столбце **swpd** показано, сколько блоков выгружено на диск (**paged**). Остальные три столбца показывают, сколько блоков свободно (**unused**), сколько используется для буферов (**buff**) и сколько — для кэша операционной системы;
- *swap*. В столбцах из этой группы показана активность подкачки: сколько блоков в секунду операционная система подкачивает с диска и выгружает на диск. Эту информацию гораздо важнее отслеживать, чем столбец **swpd**. Желательно, чтобы значения в столбцах **si** и **so** оставались равными 0 большую часть времени, и определенно они не должны превышать более 10 блоков в секунду. Кратковременные всплески активности тоже не означают ничего хорошего;
- *io*. Значения в этих столбцах показывают, сколько блоков в секунду считывается с блочных устройств (**bi**) и записывается на них (**bo**). Обычно значения в этих столбцах отражают дисковый ввод/вывод;

- *system*. В столбцах из этой группы показаны количество прерываний в секунду (*in*) и количество переключений контекста в секунду (*cs*);
- *cpu*. Значения в этих столбцах показывают, какую часть времени (в процентах) процессор затратил на выполнение пользовательского кода (вне ядра), выполнение системного кода (в ядре), простой и ожидание ввода/вывода. Если применяется виртуализация, может существовать и пятый столбец (*st*), показывающий процент «украденного» у виртуальной машины времени. Речь идет о том времени, в течение которого на виртуальной машине имелся готовый к исполнению процесс, но вместо этого гипервизор предпочел запустить что-то другое. Если у виртуальной машины не было ничего готового к запуску и при этом гипервизор запускает что-то еще, это время не считается «украденным».

Формат отчета `vmstat` зависит от системы, поэтому следует прочитать справочную страницу `vmstat(8)` в своей системе, если ваш вывод отличается от приведенного здесь примера.

Как интерпретировать выдачу `iostat`

Теперь перейдем к утилите `iostat`. По умолчанию она показывает ту же информацию об использовании ЦП, что и `vmstat`. Однако обычно нас интересует лишь статистика ввода/вывода, поэтому мы применяем следующую команду для отображения только расширенной статистики устройств:

```
$ iostat -dxk 5
Device: rrqm/s wrqm/s r/s w/s rkB/s wkB/s
sda 0.00 0.00 1060.40 3915.00 8483.20 42395.20

avgrrq-sz avgwrq-sz await r_await w_await svctm %util
20.45 3.68 0.74 0.57 0.78 0.20 98.22
```

Как и в случае с `vmstat`, в первой строке отчета отображаются средние значения с момента загрузки сервера (обычно для экономии места их опускают), а в последующих строках — инкрементные средние значения. В каждой строке выводятся данные об одном устройстве.

Существуют различные параметры, позволяющие показать или скрыть отдельные столбцы отчета. Официальная документация немного запутанна, и нам пришлось покопаться в исходном коде, чтобы понять, что на самом деле отображается. Вот что показывается в следующих столбцах:

- *rrqm/s* и *wrqm/s* — количество сгруппированных запросов на чтение и запись, поставленных в очередь в секунду. Определение «сгруппированный» означает, что операционная система объединила несколько логических запросов из очереди и сгруппировала их в один физический запрос к устройству;

- `r/s` и `w/s` — количество запросов на чтение и запись, отправленных на устройство в секунду;
- `rkB/s` и `wkB/s` — количество килобайтов, прочитанных и записанных в секунду;
- `avgrq-sz` — размер запроса в секторах;
- `avgqu-sz` — количество запросов, стоящих в очереди к устройству;
- `Await` — количество миллисекунд, потраченных на время ожидания в дисковой очереди;
- `r_await` и `w_await` — среднее время в миллисекундах для запросов на чтение, отправленных на обслуживаемое устройство, как для чтения, так и для записи соответственно. Сюда включается время, затраченное на запросы в очереди, и время на их обслуживание;
- `Svctm` — количество миллисекунд, затраченных на обслуживание запросов, за исключением времени ожидания в очереди;
- `%util`¹ — процент времени, в течение которого был активен хотя бы один запрос. Показатель назван очень неудачно. Это не использование устройства, если вы знакомы со стандартным определением использования в теории массового обслуживания. Устройство с более чем одним жестким диском, например RAID-контроллер, должно поддерживать более высокий уровень конкурентности, чем 1, но значение показателя `%util` никогда не превысит 100 %, если только при его вычислении не возникнет ошибка округления. Так что этот показатель не является хорошей характеристикой насыщенности устройства, вопреки тому что говорится в документации, за исключением особого случая, когда вы задействуете только один физический жесткий диск.

Сгенерированный отчет содержит выходные данные, которые можно использовать для вывода некоторых фактов о подсистеме ввода/вывода машины. Один из важных показателей — количество одновременно обслуживаемых запросов. Поскольку количество операций чтения и записи приведено в расчете на секунду, а единицей измерения времени обслуживания является тысячная доля секунды, можете применить формулу Литтла для вычисления количества запросов, одновременно обслуживаемых устройством:

$$\text{concurrency} = (r/s + w/s) * (svctm/1000)$$

Подставив эти полученные ранее значения выборки в формулу конкурентности, получим коэффициент конкурентности, равный примерно 0,995. Это означает, что в среднем устройство обслуживало менее одного запроса за раз в течение интервала выборки.

¹ Программный RAID, как и MD/RAID, может не показывать использование самого массива RAID.

Другие полезные инструменты

Мы показали утилиты `vmstat` и `iostat`, потому что они широко доступны, а `vmstat` обычно устанавливается по умолчанию во многих Unix-подобных операционных системах. Однако у каждого из этих инструментов есть свои ограничения, такие как путаница в единицах измерения, опросы с интервалами, которые не соответствуют тому, когда операционная система обновляет статистику, и невозможность увидеть все показатели сразу. Если эти инструменты не удовлетворяют ваши потребности, вас могут заинтересовать утилиты `dstat` или `collectl`.

Мы также предпочитаем задействовать для просмотра статистики процессора `mpstat` — она позволяет получить гораздо лучшее представление о том, как ведут себя процессоры по отдельности, вместо того чтобы группировать их все вместе. Иногда это очень важно при диагностике проблемы. Возможно, утилита `blktrace` тоже может оказаться полезной для анализа использования дискового ввода/вывода.

Компания Percona написала собственную замену `iostat` — инструмент под названием `pt-diskstats`. Он является частью пакета Percona Toolkit. В нем устранены некоторые недостатки `iostat`, например представление данных по операциям чтения и записи в совокупности, а также отсутствие наглядности в области конкурентности. Инструмент интерактивный и управляется с помощью клавиш, поэтому вы можете увеличивать и уменьшать масштаб, изменять агрегацию, отфильтровывать устройства, показывать и скрывать столбцы. Он дает отличный способ детально проанализировать выборку статистики диска, которую можно получить с помощью простого сценария командной строки, даже если у вас не установлен этот инструмент. Можете зафиксировать образцы активности диска и отправить их по электронной почте или сохранить для последующего анализа.

Наконец, профилировщик Linux `perf` — это бесценный инструмент для проверки того, что происходит на уровне операционной системы. Вы можете использовать `perf` для проверки общей информации об операционной системе, например, почему ядро так сильно задействует процессор. Можете также проверить идентификаторы определенных процессов, что позволит увидеть, как MySQL взаимодействует с операционной системой. Проверка производительности системы требует очень глубокого погружения, поэтому рекомендуем в качестве отличного дополнительного материала второе издание книги Брендана Грегга «Производительность систем»¹ (*Gregg Brendan. Systems Performance*, 2-nd ed. — Pearson).

¹ Грегг Б. Производительность систем. — Питер, 2023.

Резюме

Выбор и настройка оборудования для MySQL, а также конфигурация MySQL для аппаратного обеспечения — это не тайное искусство. В целом вам нужны те же навыки и знания, что и для достижения большинства других целей. Тем не менее существуют некоторые особенности конфигурации, специфические для MySQL, которые вы должны знать.

Обычно мы советуем найти хороший баланс между производительностью и стоимостью. По многим причинам нам нравится использовать стандартные серверы. Например, если у вас возникли проблемы с сервером и нужно вывести его из эксплуатации, пока вы пытаетесь его диагностировать, или если вы просто в качестве формы диагностики хотите попробовать заменить его другим сервером, это намного проще сделать с сервером за 5000 долларов, чем с тем, который стоит 50 000 долларов или больше. MySQL также обычно лучше подходит — с точки зрения как самого программного обеспечения, так и типичных рабочих нагрузок, с которыми она имеет дело, — для стандартного оборудования.

Четыре основных ресурса, необходимых MySQL, — это ресурсы процессора, памяти, диска и сетевые ресурсы. Сеть, как правило, не очень часто оказывается серьезным узким местом, но процессоры, память и диски, безусловно, являются таковыми. Баланс быстродействия и количества действительно зависит от рабочей нагрузки, и вы должны стремиться к балансу между быстродействием и количеством, насколько позволяет ваш бюджет. Чем большей конкурентности вы ожидаете, тем больше должны полагаться на большее количество процессоров для удовлетворения вашей рабочей нагрузки.

Взаимосвязь между процессорами, памятью и дисками сложна, и часто проблемы, возникшие в одной области, проявляются совсем в другом месте. Прежде чем бросать ресурсы на решение проблемы, спросите себя, не следует ли вместо этого направить их в другую область. Если диски сильно нагружены, вам нужно больше мощностей для ввода/вывода или просто больше памяти? Ответ зависит от размера рабочего множества, которое представляет собой набор данных, требующихся чаще всего в течение заданного периода времени.

Твердотельные устройства отлично подходят для повышения производительности сервера в целом и, как правило, в настоящее время должны стать стандартом для баз данных, особенно для рабочих нагрузок OLTP. Продолжать использовать жесткие диски стоит лишь в системах с крайне ограниченным бюджетом или в тех, где требуется ошеломляюще большой объем дискового пространства — порядка петабайт в ситуации с хранилищем данных.

Что касается операционной системы, есть всего несколько важных вещей, которые вам нужно сделать правильно, в основном они связаны с хранением, сетевым подключением и управлением виртуальной памятью. Если вы применяете GNU/Linux, как делает большинство пользователей MySQL, рекомендуем задействовать файловую систему XFS и установить для подкачки и планировщика очередей дисков значения параметров, подходящие для сервера. Есть сетевые параметры, которые вам, возможно, потребуется изменить, и вы можете настроить что-то еще (например, отключить SELinux), но эти изменения зависят лишь от ваших предпочтений.

ГЛАВА 5

Оптимизация настроек сервера

В этой главе мы объясним, как создать хороший файл конфигурации для вашего сервера MySQL. Этот процесс можно сравнить с круизом, включающим осмотр большого количества достопримечательностей и периодическими отклонениями от маршрута для посещения живописных мест. Эти отклонения от прямого пути необходимы, поскольку поиск кратчайшего пути к хорошей конфигурации начинается не с изучения параметров конфигурации и выяснения того, какие из них вы должны установить или как их изменить. Он также не начинается с изучения поведения сервера и выяснения того, могут ли какие-либо параметры конфигурации улучшить его. Вначале стоит разобраться с внутренним устройством и поведением MySQL. В дальнейшем можете использовать эти знания в качестве руководства по настройке MySQL. Наконец, вы можете сравнить желаемую конфигурацию с существующей и исправить любые существенные и важные для вас различия.

Нам часто задают вопросы примерно такого плана: «Каков оптимальный конфигурационный файл для моего сервера с 32 Гбайт оперативной памяти и 12-ядерным процессором?» К сожалению, на этот вопрос нет однозначного ответа. Вы должны настроить сервер в соответствии с рабочей нагрузкой, требованиями к данным и используемым приложениям, а не только к оборудованию. В MySQL есть множество настроек, которые можно изменить, но делать этого не стоит. В основном, лучше правильно настроить базовые параметры (а в большинстве случаев важны лишь некоторые из них) и потратить больше времени на оптимизацию схемы, индексы и проектирование запросов. Если вы правильно настроили основные параметры конфигурации MySQL, потенциальная выгода от дальнейших изменений обычно невелика.

В то же время манипулирование настройками может привести к огромным проблемам. Значения MySQL по умолчанию существуют по важной причине. Их изменение без понимания последствий может привести к сбоям, постоянным зависаниям или снижению производительности. Вы никогда не должны слепо доверять тому, что кто-то сообщает об оптимальной конфигурации на популяр-

ных справочных сайтах, таких как форумы MySQL или Stack Overflow¹. Всегда проверяйте любые изменения, прочитав соответствующий раздел руководства и тщательно протестировав.

Так что же вам следует сделать? Удостоверьтесь в том, что значения основных параметров, таких как размер буферного пула InnoDB и файла журнала, соответствуют требованиям для вашей системы. Затем следует установить несколько параметров безопасности, если хотите обеспечить хорошую работу (но обратите внимание на то, что они обычно не улучшают производительность, а лишь предотвращают проблемы). Остальные настройки оставьте в покое. Если у вас возникла проблема, начните с ее тщательной диагностики. Если она обусловлена компонентом сервера, поведение которого можно исправить с помощью параметра конфигурации, может потребоваться изменить его.

Кроме того, иногда вам может потребоваться установить определенные параметры конфигурации, которые в особых случаях способны существенно повлиять на производительность. Однако учтите, что они не должны быть частью базового файла конфигурации сервера. Вы должны устанавливать их только в том случае, если обнаружите конкретные проблемы с производительностью, которые они могут решить. Вот почему мы рекомендуем не искать специально параметры конфигурации, которые можно улучшить, и не изменять их. Если какая-то проблема требует решения, она должна сначала проявиться во времени отклика на запрос. Лучше всего заниматься улучшениями, начав с запросов и времени ответа на них, а не с параметров конфигурации. Это поможет сэкономить много времени и предотвратит множество проблем.

Еще один хороший способ сэкономить время и усилия — использовать значения по умолчанию, если только вы точно не уверены в том, что их стоит изменить. Работа с настройками по умолчанию повышает безопасность системы, поскольку многие пользователи работают с ними. В итоге эти настройки можно считать наиболее тщательно протестированными. Когда же вы без необходимости что-то меняете, могут возникнуть неожиданные ошибки.

Основы конфигурации MySQL

Мы начнем с объяснения работы механизма конфигурирования MySQL, а затем рассмотрим, что нужно настраивать в MySQL. Как правило, MySQL довольно снисходительно относится к ошибкам конфигурации, но, следуя нашим рекомендациям, вы сэкономите много времени и сил.

¹ Например, MySQL может работать невероятно быстро, если вы отключите настройки устойчивости, однако из-за этого во время сбоя можете потерять свои данные.

Первое, что нужно запомнить, — это то, откуда MySQL получает информацию о конфигурации: из аргументов командной строки и параметров в файле конфигурации. В Unix-подобных системах файл конфигурации обычно находится в каталоге `/etc/my.cnf` или `/etc/mysql/my.cnf`. Если вы используете скрипты запуска, входящие в состав операционной системы, то обычно только в этом файле и надо определять параметры конфигурации. Если же запускаете MySQL вручную, например, при тестовой установке, то можете указать параметры также в командной строке. Сервер фактически считывает содержимое файла конфигурации, удаляет из него все строки комментариев и символы разрыва строки, а затем обрабатывает этот файл вместе с параметрами командной строки.

ЗАМЕЧАНИЯ ПО ТЕРМИНОЛОГИИ

Поскольку многие параметры командной строки MySQL соответствуют переменным сервера, мы иногда используем термины «параметр» и «переменная» как синонимы. Большинство конфигурационных переменных имеют те же имена, что и соответствующие им параметры командной строки, но есть несколько исключений. Например, `--memlock` устанавливает переменную `locked_in_memory`.



Любые настройки, которые вы хотите использовать постоянно, должны войти в глобальный файл конфигурации, а не указываться в командной строке. В противном случае вы рискуете случайно запустить сервер без них. Рекомендуем также хранить все ваши файлы конфигурации в одном месте, чтобы вы могли легко их проверять.

Убедитесь, что знаете, где находится файл конфигурации вашего сервера! Нам случалось встречать людей, безуспешно пытавшихся настроить сервер с файлом, который этот сервер и не собирался читать, например `/etc/my.cnf` на серверах Debian, где сервер ищет свой файл конфигурации `/etc/mysql/my.cnf`. Иногда такие файлы находятся в нескольких местах, возможно, из-за того, что предыдущий системный администратор тоже запутался. Если вы не знаете, какие файлы читает ваш сервер, то лучше 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
```

Конфигурационный файл имеет стандартный формат INI и разбит на разделы (секции), каждый из которых начинается со строки, содержащей название секции в квадратных скобках. Программа в составе дистрибутива MySQL обычно

читает раздел с тем же именем, что и эта программа, а многие клиентские программы читают также раздел `client`, в который можно поместить общие для всех клиентов параметры. Сервер обычно считывает раздел `mysqld`. Убедитесь, что вы разместили свои параметры в правильном разделе файла, иначе они не будут иметь никакого эффекта.

Синтаксис, область видимости и динамичность

Конфигурационные параметры записываются строчными буквами, слова разделяются символами подчеркивания или дефисами. Следующие варианты эквивалентны, и вы можете увидеть обе формы в командных строках или конфигурационных файлах:

```
/usr/sbin/mysqld --auto-increment-offset=5  
/usr/sbin/mysqld --auto_increment_offset=5
```

Мы рекомендуем выбрать один из этих стилей и использовать его последовательно. Это упрощает поиск конкретного параметра в ваших файлах.

Конфигурационный параметр может иметь несколько областей действия. Некоторые параметры действуют на уровне всего сервера (глобальная область видимости), другие могут задаваться по-своему для каждого соединения (сеансовая область видимости), а третьи относятся к конкретным объектам. Многие параметры сеансовой области видимости имеют глобальные эквиваленты, которые можно рассматривать как значения по умолчанию. Если вы измените переменную области сеанса, это повлияет только на то соединение, из которого вы ее изменили, и изменения будут утрачены при закрытии соединения. Вот несколько примеров разнообразного поведения, о которых вам следует знать.

- Переменная `max_connections` имеет глобальную область видимости.
- Переменная `sort_buffer_size` имеет глобальное значение по умолчанию, но также вы можете установить ее для каждого сеанса.
- Переменная `join_buffer_size` имеет глобальное значение по умолчанию и может задаваться для каждого сеанса, кроме того, для каждого запроса, в котором задействует несколько таблиц, может выделять один буфер для каждой *операции соединения*, поэтому для одного запроса могут существовать несколько буферов соединения.

Помимо установки переменных в конфигурационных файлах, вы можете изменять многие из них (но не все) во время работы сервера. В MySQL они называются *динамическими* переменными конфигурации. Следующие операторы

показывают различные способы динамического изменения значений 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>;
```

Если вы устанавливаете переменные динамически, имейте в виду, что эти настройки будут утрачены при завершении работы MySQL. Если хотите сохранить измененные параметры, запишите их в конфигурационный файл.



Если вы устанавливаете значение глобальной переменной во время работы сервера, значения для текущего сеанса и любых других существующих сеансов не затрагиваются. Имейте это в виду, если ваши клиенты используют постоянные соединения с базой данных. Это связано с тем, что значения сеансовых переменных инициализируются из глобального значения при создании соединений. После каждого изменения выполняйте команду `SHOW GLOBAL VARIABLES`, чтобы удостовериться, что изменение произвело желаемый эффект.

Существует также специальное значение `DEFAULT`, которое вы можете присвоить переменным с помощью команды `SET`. Присвоение данного значения сеансовой переменной устанавливает ее в соответствующее значение переменной глобальной области. Это полезно для сброса переменных области сеанса обратно к значениям, которые они имели при открытии соединения. Мы рекомендуем не использовать это значение по отношению к глобальным переменным: результат может отличаться от того, чего вы хотите. То есть вы не получите значения, которые были при запуске сервера, или даже значение, указанное в файле конфигурации, — переменная будет установлена в скомпилированное значение по умолчанию.

Сохраняемые системные переменные

Если уж на то пошло, работа с областью действия переменных и с конфигурацией не была слишком сложной и вы должны были узнать, что, если MySQL будет перезапущена, она вернется к тем параметрам, которые были в вашем файле конфигурации, даже если вы использовали `SET GLOBAL` для изменения глобальной переменной. Это означало, что, администрируя конфигурационный файл и конфигурацию среды выполнения MySQL, необходимо обеспечивать их синхронизацию друг с другом. Если вы хотели увеличить для своих серверов параметр `max_connections`, требовалось выполнить команду `SET GLOBAL`

`max_connections` для каждого запущенного экземпляра, а затем отредактировать файл конфигурации, чтобы отразить новую конфигурацию.

В MySQL 8.0 представлена новая функция, называемая сохраняемыми системными переменными (`persisted system variables`), которая помогает сделать это немного проще. Новая команда `SET PERSIST` теперь позволяет установить значение один раз во время выполнения, и MySQL запишет этот параметр на диск, чтобы его можно было использовать при следующем перезапуске.

Побочные эффекты установки переменных

Динамическая установка переменных может иметь неожиданные побочные эффекты, такие как сброс на диск изменившихся блоков из буферов. Будьте осторожны с настройками, которые вы меняете онлайн, потому что это может привести к значительной нагрузке на сервер.

Иногда назначение переменной можно понять по ее имени. Например, `max_heap_table_size` делает именно то, что подразумевает ее наименование: она указывает максимальный размер, до которого могут увеличиваться в памяти временные таблицы. Однако соглашения об наименованиях не всегда последовательны, поэтому вы далеко не всегда сможете догадаться о назначении переменной по ее названию.

Рассмотрим некоторые часто используемые переменные и последствия их динамического изменения.

- `table_open_cache`. Установка этой переменной не дает немедленного эффекта — действие откладывается до следующей попытки потока открыть таблицу. Когда это происходит, MySQL проверяет значение данного параметра. Если значение больше, чем текущее количество таблиц в кэше, поток может поместить вновь открытую таблицу в кэш. Если значение меньше, чем количество таблиц в кэше, MySQL удалит неиспользуемые таблицы из кэша.
- `thread_cache_size`. Установка этой переменной не дает немедленного эффекта — действие откладывается до момента следующего закрытия соединения. В это время MySQL проверяет, есть ли в кэше место для хранения потока. Если это так, то поток кэшируется для повторного применения в будущем другим соединением. Если нет, поток уничтожается вместо кэширования. В этом случае количество потоков в кэше и, следовательно, объем памяти, который использует кэш потоков, сразу не уменьшается — это происходит только тогда, когда новое соединение удаляет поток из кэша, чтобы задействовать его. (MySQL добавляет потоки в кэш только при закрытии соединений и удаляет их из кэша только при создании новых соединений.)

- `read_buffer_size`. MySQL не выделяет память для этого буфера до тех пор, пока он не понадобится запросу. Когда же необходимость возникает, MySQL немедленно выделяет весь блок запрошенного размера.
- `read_rnd_buffer_size`. MySQL не выделяет память для этого буфера до тех пор, пока он не понадобится запросу. Когда же необходимость возникает, MySQL немедленно выделяет весь блок запрошенного размера. (Название `max_read_rnd_buffer_size` более точно описывало бы эту переменную.)

Официальная документация MySQL подробно объясняет назначение этих переменных. И это не исчерпывающий список. Наша цель — просто показать, какого поведения следует ожидать при изменении нескольких важных переменных.

Не следует глобально увеличивать параметр, относящийся к соединению, если не уверены, что это правильно. Некоторые буферы выделяются сразу одним куском, даже если они не нужны, и глобальное изменение параметра может привести к растрачиванию памяти впустую. Вместо этого вы можете увеличить значение, когда это необходимо для конкретного запроса.

Планирование изменений ваших переменных

Будьте осторожны при установке переменных. Больше — не всегда лучше, и если вы установите слишком большие значения, то можете легко вызвать проблемы — у вас может не хватить памяти или сервер начнет выгружать ее в файл подкачки.

Как говорилось в главе 2, отслеживайте свои SLO, чтобы убедиться, что сделанные вами изменения не влияют на качество обслуживания клиентов. Эталонного тестирования недостаточно, поскольку оно не характеризует реальное использование сервера. Если не измерять фактическую производительность сервера, то можно ухудшить производительность, даже не подозревая об этом. Мы видели много случаев, когда кто-то менял конфигурацию сервера и думал, что это повысило производительность, хотя на самом деле производительность сервера в целом ухудшалась, поскольку в разное время дня или в разные дни недели рабочая нагрузка варьировалась.

В идеале необходимо использовать систему контроля версий для отслеживания изменений в файлах конфигурации. Эта стратегия может быть очень эффективной при сопоставлении изменения производительности или нарушения SLO с конкретным изменением конфигурации. Просто имейте в виду, что изменение файла конфигурации ничего не делает по умолчанию — нужно также изменить настройку среды выполнения.

Прежде чем приступить к изменению конфигурационных параметров, следует оптимизировать запросы и схему, обратившись хотя бы к таким очевидным

вещам, как добавление индексов. Если вы слишком углубитесь в настройку конфигурации, а затем измените свои запросы или схему, вам, возможно, придется начинать настройку заново. Имейте в виду, что, если ваше оборудование, рабочая нагрузка и данные не являются абсолютно неизменными, скорее всего, придется вернуться к конфигурированию позже. На самом деле большинство серверов не имеют постоянной рабочей нагрузки в течение дня, а это означает, что идеальная конфигурация для середины утра не подходит для полудня! Очевидно, что гнаться за мифической идеальной конфигурацией совершенно нецелесообразно. Таким образом, вам не нужно выжимать из своего сервера всю производительность до последней капли. Ведь в реальности отдача от таких затрат времени, вероятно, будет очень небольшой. Мы рекомендуем сосредоточиться на оптимизации пиковой рабочей нагрузки, а затем остановиться на приемлемом результате, если у вас нет оснований полагать, что можно добиться значительного повышения производительности.

Чего делать не следует

Прежде чем приступить к настройке сервера, мы хотим посоветовать вам избегать нескольких распространенных практик, которые, по нашему мнению, являются рискованными или практически не стоящими затраченных усилий. Предупреждение: громкие слова впереди!

Возможно, от вас ожидают (или полагают, что от вас ожидают) создания набора эталонных тестов и настройки сервера путем итеративного изменения его конфигурации в поисках оптимальных параметров. Обычно мы не советуем это делать. Это требует так много работы и исследований, а потенциальная отдача в большинстве случаев настолько мала, что много времени будет потрачено впустую. Вероятно, лучше потратить его на другие действия, такие как проверка резервных копий, мониторинг изменений в планах запросов и т. д.

Вы не должны производить тонкую настройку по коэффициенту. Классический пример коэффициента тонкой настройки — это эмпирическое правило, согласно которому коэффициент попаданий в кэш ключей InnoDB должен быть выше некоторого процента и следует увеличить размер кэша, если коэффициент попаданий слишком низкий. Это совершенно неправильный совет. Независимо от того, что вам говорят, *коэффициент попаданий в кэш не имеет никакого отношения к тому, слишком большой кэш или нет*. Начнем с того, что коэффициент совпадений зависит от рабочей нагрузки — некоторые рабочие нагрузки просто не кэшируются независимо от размера кэша, к тому же попадания в кэш бессмысленны по причинам, которые мы объясним позже. Иногда бывает, что, когда кэш слишком мал, коэффициент попадания низок, а увеличение размера

кэша увеличивает коэффициент попаданий. Однако это случайная корреляция, и ничто не говорит о производительности или правильном определении размера кэша.

Проблема с корреляциями, которые иногда кажутся верными, заключается в том, что люди начинают думать, что те всегда будут верны. Администраторы баз данных Oracle отказались от настройки на основе коэффициентов много лет назад, и мы хотим, чтобы администраторы баз данных MySQL последовали их примеру¹. Еще больше мы желаем, чтобы люди не писали «скрипты настройки», которые кодифицируют эти опасные практики и обучают им тысячи людей. Из этого вытекает следующая рекомендация: не используйте скрипты настройки! Несколько очень популярных вы можете найти в Интернете. Вероятно, лучше всего их игнорировать.

Мы также предлагаем вам избегать словосочетания «тонкая настройка», которое часто употребляли в нескольких последних разделах. Вместо него стоит применять термины «конфигурация» или «оптимизация» (если это то, что вы на самом деле делаете). Когда мы видим словосочетание «тонкая настройка», воображение рисует образ недисциплинированного новичка, который настраивает сервер и смотрит, что происходит. В предыдущем разделе мы рекомендовали оставить эту практику тем, кто исследует внутреннее устройство сервера. «Тонкая настройка» вашего сервера может оказаться пустой тратой времени.

Что касается смежной темы, то поиск в Интернете рекомендаций по разработке конфигурации тоже не всегда является хорошей идеей. Вы можете найти много плохих советов в блогах, на форумах и т. д. Хотя многие эксперты делятся своими знаниями в Интернете, не всегда легко определить, кто является квалифицированным специалистом. Конечно, мы не можем дать объективных и беспристрастных рекомендаций о том, где найти настоящих специалистов. Но можем сказать, что заслуживающие доверия и авторитетные поставщики услуг MySQL в целом являются более безопасным выбором, чем результаты простого поиска в Интернете, потому что люди, у которых есть довольные клиенты, вероятно, делают что-то правильно. Однако даже их советы могут быть опасны, если их применять без проверки и понимания, потому что они могут относиться к ситуации, отличающейся от вашей, и неясным для вас деталям.

¹ Если вы не уверены, что тонкая настройка по коэффициенту — это плохо, прочтите книгу «Oracle. Оптимизация производительности» Кэри Миллсэп и Джеффа Холта (*Millsap Cary, Holt Jeff. Optimizing Oracle Performance. — O'Reilly*). Они даже посвятили этой теме приложение с инструментом, который может искусственно генерировать любое соотношение попаданий в кэш, какое вы пожелаете, независимо от того, насколько плохо работает ваша система! Конечно, все это для того, чтобы проиллюстрировать бесполезность такого соотношения.

Наконец, не верьте популярной формуле потребления памяти — да, той самой, которую выводит сама MySQL при сбое. (Мы не будем повторять ее здесь.) Она сохранилась с древнейших времен и не является надежным или хотя бы полезным способом понять, сколько памяти MySQL может использовать в худшем случае. Кроме того, вы можете найти некоторые варианты этой формулы в Интернете. Они также ошибочны, даже если в них добавлены факторы, которых нет в исходной формуле. По правде говоря, вы не можете установить верхнюю границу потребления памяти MySQL. Она не регулируется жестко сервером базы данных, который управляет распределением памяти.

Создание конфигурационного файла MySQL

Как мы предупреждали в самом начале этой главы, у нас нет универсального «наилучшего конфигурационного файла», скажем, для сервера с четырьмя процессорами, 16 Гбайт оперативной памяти и 12 жесткими дисками, который годился бы на все случаи жизни. Вам действительно придется разработать собственные конфигурации, потому что даже хорошая отправная точка будет сильно различаться в зависимости от того, как вы используете конкретное оборудование.

Минимальная конфигурация

Для этой книги мы создали минимальный пример файла конфигурации, который вы можете применять в качестве хорошей отправной точки для создания собственных серверов¹. Вы должны выбрать значения для некоторых параметров, мы объясним их позже в этой главе. Наш базовый файл, созданный для MySQL 8.0, выглядит следующим образом:

```
[mysqld]
# GENERAL
datadir                = /var/lib/mysql
socket                 = /var/lib/mysql/mysql.sock
pid_file               = /var/lib/mysql/mysql.pid
user                   = mysql
port                   = 3306
# INNODB
innodb_buffer_pool_size = <value>
innodb_log_file_size   = <value>
innodb_file_per_table   = 1
innodb_flush_method    = O_DIRECT
# LOGGING
```

¹ Обратите внимание на то, что в новых версиях MySQL некоторые параметры удаляются, изменяются и устаревают. Проверьте детали в документации.

```

log_error                = /var/lib/mysql/mysql-error.log
log_slow_queries         = /var/lib/mysql/mysql-slow.log
# OTHER
tmp_table_size           = 32M
max_heap_table_size      = 32M
max_connections          = <value>
thread_cache_size        = <value>
table_open_cache         = <value>
open_files_limit         = 65535
[client]
socket                   = /var/lib/mysql/mysql.sock
port                     = 3306

```

Этот файл может показаться *слишком* минималистичным по сравнению с тем, что вы привыкли видеть, но на самом деле он включает в себя даже больше, чем нужно многим людям. Существует несколько других типов параметров конфигурации, которые вы, вероятно, также будете использовать, например ведение двоичного журнала. Мы рассмотрим их позже в этой и других главах.

Первое, что мы сконфигурировали, — это расположение данных. Для этого выбрали `/var/lib/mysql`, потому что это популярное место для большинства вариантов Unix. Но нет ничего плохого в том, чтобы выбрать другое место, — вам решать. Мы поместили файл `.pid` в то же место, однако многие операционные системы захотят поместить его в `/var/run`. Это тоже хорошо. Нам просто требовалось установить что-то для этих параметров. Кстати, не позволяйте сокету и файлу `.pid` располагаться в соответствии с установленными на сервере значениями по умолчанию — в некоторых версиях MySQL есть ряд ошибок, которые из-за этого могут вызвать проблемы. Лучше всего указать эти местоположения явно. (Мы не советуем выбирать разные местоположения, а просто рекомендуем убедиться, что эти местоположения явно упомянуты в файле `my.cnf`, чтобы они не изменились и не нарушились при обновлении сервера.) Мы также указали, что `mysqld` должен работать как учетная запись пользователя `mysql` в операционной системе. Вам потребуется убедиться, что эта учетная запись существует и этот аккаунт является владельцем каталога данных и всех файлов внутри него. По умолчанию для порта установлено значение 3306, но иногда может понадобиться его изменить.

В MySQL 8.0 был введен новый параметр конфигурации `innodb_dedicated_server`. Он проверяет доступную память на сервере и соответствующим образом настраивает четыре дополнительные переменные (`innodb_buffer_pool_size`, `innodb_log_file_size`, `innodb_log_files_in_group` и `innodb_flush_method`) для выделенного сервера базы данных, что упрощает вычисление и изменение этих значений. Это может быть особенно полезно в облачной среде, где вы можете запустить виртуальную машину со 128 Гбайт ОЗУ, а затем перезагрузить ее

с целью масштабирования до 256 Гбайт ОЗУ. MySQL здесь будет самонастраивающейся, и вам не нужно управлять изменением значений в файле конфигурации. Часто это наилучший способ управлять этими четырьмя параметрами.

Большинство других параметров в нашем примере файла конфигурации говорят сами за себя, и многие из них являются предметом обсуждения. Мы рассмотрим некоторые из параметров в оставшейся части этой главы. А еще позже обсудим настройки безопасности, которые могут быть очень полезны для повышения надежности вашего сервера и предотвращения получения неверных данных и других проблем. Тут мы эти настройки не показали.

Здесь нужно объяснить один параметр — `open_files_limit`. Мы установили для него значение, максимально возможное для типичной системы Linux. Дескрипторы открытых файлов очень дешевы в современных операционных системах. Если этот параметр недостаточно велик, вы увидите ошибку `24 Too many open files` (Слишком много открытых файлов).

Пропустив все до конца, перейдем к последнему разделу в файле конфигурации. Он предназначен для клиентских программ, таких как `mysql` и `mysqladmin`, и просто сообщает им, как подключиться к серверу. Вы должны установить значения для клиентских программ так, чтобы они соответствовали тем, которые выбраны для сервера.

Проверка переменных состояния сервера MySQL

Иногда для того, чтобы лучше настроить параметры сервера, учитывающие фактическую рабочую нагрузку, в качестве исходных данных можно использовать вывод команды `SHOW GLOBAL STATUS`. Для достижения наилучших результатов обращайте внимание как на абсолютные значения, так и на то, как значения изменяются с течением времени, желательно с несколькими моментальными снимками в пиковое и непиковое время. Чтобы увидеть инкрементальные изменения переменных состояния каждые 60 с, можете применить следующую команду:

```
$ mysqladmin extended-status -ri60
```

Объясняя различные настройки конфигурации, мы будем часто ссылаться на изменения переменных состояния с течением времени. Обычно мы ожидаем, что вы будете изучать вывод команды, подобной той, которую мы только что показали. Другими полезными инструментами, способными обеспечить компактное отображение изменений счетчика состояния, являются `pt-mext` или `pt-mysqlsummary` из пакета Percona Toolkit.

Теперь, закончив с описанием предварительных сведений, проведем для вас экскурсию по некоторым внутренним устройствам сервера, перемежая ее советами по настройке. Таким образом вы получите справочную информацию, необходимую для выбора соответствующих значений параметров конфигурации, когда мы позже вернемся к примеру файла конфигурации.

Настройка использования памяти

При использовании `innodb_dedicated_server` обычно задействуется 50–75 % вашей оперативной памяти. Это оставляет вам как минимум 25 % для выделения памяти каждому соединению, накладных расходов операционной системы и других параметров памяти. Мы рассмотрим каждый из них в следующих разделах, а затем более подробно поговорим о требованиях к различным кэшам MySQL.

Сколько памяти нужно для соединения

MySQL требуется небольшой объем памяти только для того, чтобы поддерживать соединение открытым (обычно со связанным выделенным потоком). Кроме того, определенное количество памяти требуется для выполнения любого запроса. Вам нужно будет выделить достаточно памяти для MySQL, чтобы выполнять запросы в периоды пиковой нагрузки. В противном случае запросам будет не хватать памяти и они станут выполняться очень медленно или вообще завершаться с ошибкой.

Вообще полезно знать, сколько памяти MySQL будет потреблять во время пиковой нагрузки, но в некоторых ситуациях потребление памяти неожиданно и резко возрастает, что делает любые прогнозы ненадежными. Подготовленные операторы являются одним из таких примеров, потому что вы можете открыть одновременно многие из них. Другим примером является словарь данных InnoDB (подробнее об этом позже).

Вам не нужно предполагать наихудший сценарий при попытке предсказать пиковое потребление памяти. Например, если MySQL сконфигурирована так, чтобы разрешать максимум 100 одновременных соединений, то теоретически возможно одновременное выполнение очень тяжелых запросов по всем 100 соединениям, но в действительности этого, скорее всего, не произойдет. Запросы, в которых используется множество больших временных таблиц или сложных хранимых процедур, являются наиболее вероятными причинами высокого потребления памяти для каждого соединения.

Резервирование памяти для операционной системы

Для работы операционной системы, как и для выполнения запросов, вам необходимо зарезервировать достаточно памяти. Обычно ОС использует память, чтобы запускать любое локальное программное обеспечение для мониторинга, инструментов управления конфигурацией, запланированных заданий и т. д. Лучшим свидетельством того, что в операционной системе достаточно памяти, является то, что она не выполняет активную подкачку виртуальной памяти на диск.

Буферный пул InnoDB

Для буферного пула InnoDB требуется больше памяти, чем для чего-то еще, поскольку это, как правило, самая важная переменная для производительности. В буферном пуле InnoDB кэшируются не только индексы — там хранятся также сами данные, адаптивный хеш-индекс, буфер вставок, блокировки и другие внутренние структуры. В InnoDB буферный пул используется также для реализации отложенных операций записи и позволяет объединить несколько таких процедур, чтобы затем выполнить их последовательно. Короче говоря, InnoDB *очень сильно* зависит от буферного пула, и вы должны быть уверены, что выделили для него достаточно памяти. Для мониторинга использования памяти вашего буферного пула InnoDB можно проанализировать переменные, выводимые командами `SHOW` или инструментов типа `innotop`.

Если у вас не так уж много данных и вы знаете, что их количество не будет расти быстро, не нужно выделять слишком много памяти для буферного пула. На самом деле не стоит делать его намного больше, чем размер таблиц и индексов, которые будут в нем храниться. Конечно, нет ничего плохого в том, чтобы заранее планировать быстрорастущую базу данных, но иногда мы видим огромные буферные пулы с крошечным объемом данных. Это совсем не обязательно.

Большие буферные пулы сопряжены с некоторыми проблемами, такими как длительное время отключения и «прогрева». Если в буферном пуле много «грязных» (модифицированных) страниц, InnoDB может потребоваться много времени для завершения работы, потому что она записывает «грязные» страницы в файлы данных при завершении работы. Вы можете принудительно выполнить процедуру быстрого останова, но тогда InnoDB просто нужно будет больше времени на восстановление при перезапуске, поэтому вы не можете фактически уменьшить суммарное время цикла выключения и перезапуска. Если вы заранее знаете, что придется останавливать сервер, то можете изменить переменную `innodb_max_dirty_pages_pct` во время выполнения, задав более низкое значение, дождаться, когда поток сброса очистит буферный пул, а затем завершить работу,

как только количество «грязных» страниц станет небольшим. Вы можете отслеживать количество «грязных» страниц, наблюдая за переменной состояния сервера `innodb_buffer_pool_pages_dirty` или используя утилиту `innotop` для мониторинга вывода команды `SHOW INNODB STATUS`. Можете также взять переменную `innodb_fast_shutdown`, чтобы настроить процесс выключения MySQL.

Уменьшение значения переменной `innodb_max_dirty_pages_pct` на самом деле не гарантирует, что InnoDB будет хранить в буферном пуле 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 с целью повышения эффективности группирует операции записи и выполняет их последовательно. Такое поведение называется *ленивым*, поскольку оно позволяет InnoDB откладывать сброс «грязных» страниц в буферный пул, если ему не нужно использовать пространство для каких-то других данных. Когда процент «грязных» страниц превышает заданное пороговое значение, InnoDB начинает сбрасывать страницы на диск с максимальной возможной скоростью, стремясь уменьшить их количество. Операции очистки страниц были значительно оптимизированы по сравнению с поведением в предыдущих версиях, включая возможность настройки нескольких потоков для выполнения очистки.

Когда MySQL снова запускается, кэш буферного пула, также называемый холодным кэшем, пуст. Все преимущества наличия строк и страниц в памяти исчезли. К счастью, по умолчанию параметры конфигурации `innodb_buffer_pool_dump_at_shutdown` и `innodb_buffer_pool_load_at_startup` работают вместе, чтобы «прогреть» сервер при запуске. Загрузка кэша при запуске требует времени, но она может повысить производительность сервера намного быстрее, чем ожидание его непосредственного заполнения.

Кэш потоков

Кэш потоков содержит потоки, которые в данный момент не связаны ни с одним соединением, но готовы обслуживать новые соединения. Если в кэше есть поток и поступает запрос на создание нового соединения, MySQL удаляет поток из кэша и передает его создаваемому соединению. Когда соединение закрывается, MySQL помещает поток обратно в кэш, если там есть место. Если места нет, она уничтожает поток. Пока в кэше есть свободные потоки, MySQL может очень быстро реагировать на запросы об открытии соединения, потому что ей не нужно создавать новый поток для обслуживания каждого нового соединения.

Переменная `thread_cache_size` определяет максимальное количество потоков, которые MySQL может хранить в этом кэше. Вам, вероятно, не нужно будет

изменять значение по умолчанию `-1` или автоматический размер, если только ваш сервер не получает много запросов на подключение. Чтобы проверить, достаточно ли велик кэш потока, посмотрите на переменную состояния `Threads_created`. Обычно мы стараемся поддерживать кэш потока достаточно большим, чтобы в каждую секунду создавалось не более десяти новых потоков, но часто без труда удастся снизить этот показатель, так что в секунду будет создаваться менее одного потока.

Правильный подход состоит в том, чтобы, понаблюдав за переменной `Threads_connected`, попытаться установить значение `thread_cache_size` достаточно большим, чтобы справиться с типичными колебаниями вашей рабочей нагрузки. Например, если `Threads_connected` обычно изменяется между 100 и 120, можно установить размер кэша равным 20. Если она изменяется между 500 и 700, то кэша размером 200 будет достаточно. Мы рассуждаем следующим образом: при 700 соединениях в кэше, вероятно, нет потоков, при 500 соединениях имеется 200 кэшированных потоков, готовых к использованию, если нагрузка снова увеличится до 700.

Создавать очень большой кэш потоков чаще всего не обязательно, однако и уменьшение его сверх меры экономит немного памяти, так что в этом мало пользы. Каждый поток, находящийся в кэше потоков или спящий, обычно использует около 256 Кбайт памяти. Это совсем немного по сравнению с объемом памяти, который поток может задействовать, когда соединение активно обрабатывает запрос. В общем, вы должны иметь достаточно большой кэш потоков, чтобы `Thread_created` не увеличивался слишком часто. Однако, если при таком условии объем кэша становится слишком велик (например, порядка нескольких тысяч потоков), можете уменьшить его, потому что некоторые операционные системы плохо справляются с очень большим количеством потоков, даже когда большинство из них находятся в спящем режиме.

Настройка ввода/вывода в MySQL

Несколько конфигурационных параметров управляют тем, как MySQL синхронизирует данные в памяти и на диске и выполняет восстановление. Эти параметры могут существенно повлиять на производительность, поскольку относятся к весьма затратным операциям ввода/вывода. Кроме того, они определяют компромисс между производительностью и безопасностью данных. Как правило, стремление обеспечить немедленную и непротиворечивую запись ваших данных на диск очень затратно. Если вы готовы пойти на риск, обусловленный тем, что запись на диск еще не обеспечивает постоянного хранения, то можете увеличить

степень конкурентности и/или уменьшить время ожидания ввода/вывода. Но вам придется самостоятельно определять допустимый уровень риска.

InnoDB позволяет вам управлять не только способом восстановления, но и тем, как открываются файлы и сбрасываются на диск данные кэшей, что сильно влияет на восстановление и общую производительность. Процесс восстановления в InnoDB полностью автоматический и всегда выполняется в момент запуска InnoDB, хотя вы можете влиять на то, какие действия при этом предпринимаются. Даже если оставить в стороне вопрос о восстановлении и предполагать, что ничего никогда не выйдет из строя или вообще все нормально, у вас в любом случае имеется множество возможностей для конфигурирования InnoDB. Эта подсистема имеет сложную цепочку буферов и файлов, спроектированных так, чтобы добиться высокой производительности и гарантировать свойства ACID, и при этом каждый элемент цепочки можно настраивать. На рис. 5.1 показаны эти файлы и буферы.

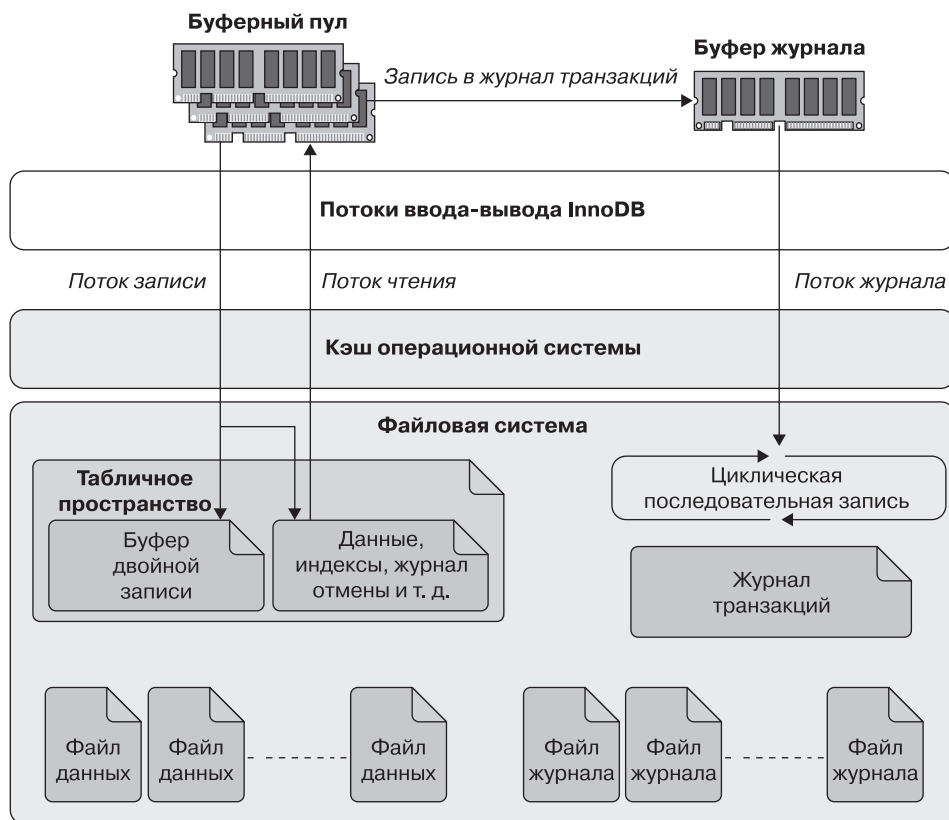


Рис. 5.1. Буферы и файлы InnoDB

Наиболее важные вещи, которые имеет смысл изменить для нормального использования, — это размер файла журнала InnoDB, способ сброса журнального буфера на диск и то, как InnoDB выполняет ввод/вывод.

Журнал транзакций InnoDB

В InnoDB журнал применяется для уменьшения затрат на коммит транзакций. Вместо того чтобы сбрасывать пул буферов на диск после коммита каждой транзакции, InnoDB регистрирует транзакции в журнале. Изменения в данных и индексах, произведенные внутри транзакций, часто относятся к различным местам в табличном пространстве, поэтому для сброса этих изменений на диск потребуются операции произвольного ввода/вывода. InnoDB предполагает использование обычных дисков, где произвольный ввод/вывод намного затратнее, чем последовательный, из-за времени, необходимого для поиска нужного места на диске и ожидания, пока оно не окажется под головкой.

InnoDB использует журнал для преобразования произвольного дискового ввода/вывода в последовательный. Как только произведена запись в журнал на диске, транзакции становятся долговечными, хотя изменения еще не были записаны в файлы данных. Если произойдет какая-то авария (например, сбой питания), InnoDB сможет воспроизвести журнал и восстановить закоммиченные транзакции.

Конечно, InnoDB в конечном итоге должна записывать изменения в файлы данных, потому что журнал имеет фиксированный размер. Запись в журнал выполняется циклически: при достижении конца журнала происходит переход к его началу. InnoDB не может затереть запись журнала, если содержащиеся в ней изменения не были занесены в файлы данных, потому что при этом была бы уничтожена единственная долговечная запись о закоммиченной транзакции.

InnoDB использует фоновый поток для рационального сброса изменений в файлы данных. Этот поток умеет группировать операции записи так, чтобы они выполнялись последовательно, для повышения эффективности. По существу, журнал транзакций преобразует произвольный ввод/вывод файла данных в преимущественно последовательный ввод/вывод файла журнала и файла данных. Выполнение сброса изменений в фоновом режиме ускоряет выполнение запросов и помогает защитить систему ввода/вывода от всплесков нагрузки запросов.

Общий размер файла журнала, задаваемый параметрами `innodb_log_file_size` и `innodb_log_files_in_group`, очень важен с точки зрения производительности записи. Если вы воспользовались нашей предыдущей рекомендацией и задействовали параметр `innodb_dedicated_server`, эти параметры будут регулироваться за вас в зависимости от того, сколько памяти имеет ваша система.

Буфер журнала

При модификации любых данных InnoDB помещает запись об изменении в свой *буфер журнала*, который хранится в памяти. InnoDB сбрасывает буфер в файлы журнала на диске в следующих случаях: когда буфер заполняется, когда производится коммит транзакции или раз в секунду в зависимости от того, что произойдет раньше. Увеличение размера буфера, который по умолчанию составляет 1 Мбайт, может помочь сократить количество операций ввода/вывода в случае больших транзакций. Управляющая размером буфера переменная называется `innodb_log_buffer_size`.

Обычно нет необходимости делать буфер очень большим. Рекомендуемый размер варьируется в диапазоне 1–8 Мбайт, и обычно этого достаточно, если только вы не вставляете много записей с огромными BLOB. Записи журнала очень компактны по сравнению с обычными данными InnoDB. Они не используют страницы как единицу хранения, поэтому не расходуют место на запись целых страниц каждый раз. InnoDB также стремится максимально сократить записи журнала. Иногда даже в них хранятся лишь несколько целых чисел — номер зарегистрированной операции и параметры, необходимые для ее выполнения!

Как InnoDB сбрасывает буфер журнала

Когда InnoDB сбрасывает буфер журнала в файлы журнала на диске, он блокирует доступ к буферу с помощью мьютекса, переписывает данные на диск вплоть до нужной точки, а затем перемещает все оставшиеся записи в начало буфера. Может случиться так, что к моменту освобождения мьютекса скопится несколько транзакций, готовых к записи в журнал. InnoDB использует механизм группового коммита, позволяющий записать их все в журнал за одну операцию ввода/вывода.

Буфер журнала *должен* быть сброшен на устройство постоянного хранения, чтобы обеспечить долговечность зафиксированных транзакций. Если производительность вам важнее, чем долговечность, можете изменить `innodb_flush_log_at_trx_commit`, чтобы контролировать, куда и как часто сбрасывается буфер журнала.

Возможны следующие значения:

- 0 — писать буфер в файл журнала и сбрасывать журнал на устройство постоянного хранения каждую секунду, но ничего не делать в момент коммита транзакции;
- 1 — писать буфер в файл журнала и сбрасывать его в долговременное хранилище каждый раз, когда выполняется коммит транзакции. Это настройка по умолчанию (и самая безопасная), она гарантирует, что вы не потеряете

ни одной зафиксированной транзакции, если диск или операционная система не делают операцию сброса фиктивной;

- 2 — писать буфер в файл журнала при каждом коммите, но не сбрасывать его. InnoDB планирует выполнение сброса один раз в секунду. Самое важное отличие от параметра 0 заключается в том, что при значении 2 транзакции не будут потеряны в случае аварийного завершения процесса MySQL. Однако, если весь сервер выйдет из строя или будет нарушено его питание, вы все равно можете потерять транзакции.

Важно понимать разницу между *записью* буфера в файл журнала и *сбросом* журнала на устройство долговременного хранения. В большинстве операционных систем запись буфера в журнал сводится к простому копированию данных из буфера памяти InnoDB в кэш операционной системы, который также находится в памяти. Никакой записи на реальное устройство при этом не происходит. Таким образом, настройки 0 и 2 *обычно* приводят к потере данных не более чем на 1 с в случае сбоя или отключения питания, поскольку в течение этого времени информация, возможно, существует только в кэше операционной системы. Мы говорим «обычно», потому что InnoDB старается сбросить файл журнала на диск примерно раз в секунду при любых обстоятельствах, но в некоторых ситуациях возможна потеря транзакций более чем на секунду, например, когда поток сброса останавливается.

Иногда контроллер жесткого диска или операционная система имитирует сброс, помещая данные в *еще один* кэш, например в собственный кэш жесткого диска. Этот прием более быстрый, но очень опасный, потому что данные могут быть потеряны, если у диска внезапно исчезнет питание. Такая ситуация даже хуже, чем установка параметра `innodb_flush_log_at_trx_commit` в любое значение, отличное от 1, потому что может привести к повреждению данных, а не просто к потере транзакций.

Установка для параметра `innodb_flush_log_at_trx_commit` значения, отличного от 1, может привести к потере транзакций. Однако если долговечность (буква D (durability) в аббревиатуре ACID) для вас не очень важна, то, возможно, вы сочтете полезными другие настройки. Может быть, вам просто нужны иные возможности InnoDB, такие как кластерные индексы, устойчивость к повреждению данных и блокировка на уровне строк.

Наилучшая конфигурация для высокопроизводительных транзакционных приложений получается, если оставить значение `innodb_flush_log_at_trx_commit` равным 1 и поместить файлы журналов в RAID-том с кэш-памятью записи с батарейным питанием и твердотельными накопителями. Это и безопасно, и очень быстро. На самом деле мы осмеливаемся сказать, что любой производственный сервер базы данных, который должен обрабатывать серьезную рабочую нагрузку, должен иметь такое оборудование.

Как InnoDB открывает и сбрасывает файлы журналов и данных

Параметр `innodb_flush_method` позволяет вам указать, как InnoDB фактически взаимодействует с файловой системой. Несмотря на свое название, он влияет также на то, как InnoDB читает данные, а не только на то, как он их записывает.



Изменение режима выполнения операций ввода/вывода в InnoDB может существенно повлиять на производительность в целом, поэтому убедитесь, что вы понимаете, что делаете, прежде чем что-либо менять!

Этот параметр может слегка сбить с толку, поскольку он влияет как на файлы журналов, так и на файлы данных и иногда выполняет разные операции для разных типов файлов. Было бы неплохо иметь один параметр конфигурации для журналов, а другой для файлов данных, но они объединены.

Если вы используете Unix-подобную операционную систему и ваш RAID-контроллер имеет кэш записи с резервным питанием от батареи, рекомендуем вам применять `O_DIRECT`. Если нет, то лучшим выбором, вероятно, будет либо значение по умолчанию, либо `O_DIRECT` в зависимости от вашего приложения. Как было сказано ранее, этот параметр устанавливается автоматически, если вы решили использовать параметр `innodb_dedicated_server`.

Табличное пространство InnoDB

InnoDB хранит свои данные в *табличном пространстве*, которое, по существу, представляет собой виртуальную файловую систему, охватывающую один или несколько файлов на диске. InnoDB использует табличное пространство не только для хранения таблиц и индексов, но и для иных целей. В нем же находятся журнал отмены (информация, необходимая для воссоздания старых версий строк), буфер изменений, буфер двойной записи и другие внутренние структуры табличного пространства.

Конфигурирование табличного пространства

Файлы, помещаемые в табличное пространство, перечисляются в конфигурационном параметре `innodb_data_file_path`. Все файлы будут находиться в каталоге, заданном 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:

```
innodb_data_file_path = /disk1/ibdata1:1G:/disk2/ibdata2:1G;...
```

Хотя при этом файлы действительно помещаются в разные каталоги, которые в данном случае находятся на разных дисках, но InnoDB объединяет файлы друг за другом. Таким образом, обычно вы не получаете реального выигрыша. InnoDB сначала будет заполнять первый файл, затем второй, когда первый будет заполнен, и т. д. — нагрузка на самом деле не распределяется так, как вам хотелось бы для достижения более высокой производительности. RAID-контроллер — это более разумный способ распределения нагрузки.

Чтобы позволить табличному пространству увеличиваться, если данным не хватает места, вы можете сделать так, чтобы последний файл автоматически расширялся следующим образом:

```
...ibdata3:1G:autoextend
```

По умолчанию создается один автоматически расширяемый файл размером 10 Мбайт. Если вы решите сделать файл автоматически расширяемым, хорошей идеей будет установить верхний предел размера табличного пространства, чтобы предотвратить очень большой его рост, потому что, достигнув определенного размера, он уже не сжимается. Например, в следующем примере размер автоматически расширяемого файла ограничен 2 Гбайт:

```
...ibdata3:1G:autoextend:max:2G
```

Управление одним табличным пространством может вызвать сложности, особенно если оно автоматически расширяется и вы хотите освободить пространство (по этой причине рекомендуем отключить функцию автоматического расширения или по крайней мере установить разумный предел пространства). В данном случае единственный способ освободить место — это 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 удалить файлы или изменить их размеры. InnoDB откажется запускаться, если обнаружит, что табличное пространство повреждено. Так же трепетно InnoDB относится к своим файлам журналов. Если вы привыкли без особых раздумий перемещать файлы, как вы могли бы делать с MyISAM, будьте осторожны!

Параметр `innodb_file_per_table` позволяет сконфигурировать InnoDB для использования одного файла на таблицу. Данные хранятся в каталоге базы данных

в виде файлов с именами вида `tablename.ibd`. Это упрощает освобождение места при удалении таблицы. Однако размещение данных в нескольких файлах может на самом деле привести к увеличению неиспользуемого пространства в целом, поскольку внутренняя фрагментация в одном табличном пространстве InnoDB заменяется неиспользуемым пространством в файлах `.ibd`.

Даже если вы включите режим `innodb_file_per_table`, вам все равно понадобится главное табличное пространство для журналов отмены и других системных данных. Если в нем не будут храниться все данные, оно будет меньше размером.

Некоторым нравится использовать режим `innodb_file_per_table` просто потому, что он предоставляет дополнительные средства управления и делает структуру базы более наглядной. Например, гораздо быстрее определить размер таблицы, изучив один файл, чем выполнять команду `SHOW TABLE STATUS`, которая должна делать более сложную работу, чтобы определить, сколько страниц выделено для таблицы.



Однако у `innodb_file_per_table` всегда был и недостаток — медленное выполнение команды `DROP TABLE`. Она может выполняться настолько медленно, что способна даже заметно затормозить весь сервер. Это может произойти по двум причинам.

Удаление таблицы разрывает (удаляет) связи файла на уровне файловой системы, что может быть очень медленным в некоторых файловых системах (`ext3`, мы смотрим на тебя). Вы можете сократить продолжительность этой операции с помощью трюков с файловой системой: свяжите файл `.ibd` с файлом нулевого размера, затем удалите файл вручную, не дожидаясь, пока это сделает MySQL.

Когда вы включаете этот параметр, каждая таблица получает собственное табличное пространство внутри InnoDB. Оказывается, что удаление табличного пространства фактически требует, чтобы InnoDB блокировала и сканировала буферный пул, пока она ищет страницы, принадлежащие этому табличному пространству, что очень медленно выполняется на сервере с большим буферным пулом. Ситуацию можно улучшить, если разбить буферный пул на множество частей, используя `innodb_buffer_pool_instances`.

Несколько исправлений были внесены в различных версиях MySQL. Начиная с версии 8.0.23, эта проблема решена.

Какова же окончательная рекомендация? Мы предлагаем вам использовать `innodb_file_per_table` и ограничить размер общего табличного пространства, чтобы облегчить себе жизнь. Если вы столкнетесь с какими-либо обстоятельствами, которые делают это болезненным, как отмечалось ранее, рассмотрите одно из предложенных нами исправлений.

Старые версии строк и табличное пространство

В среде с большим количеством операций записи табличное пространство InnoDB может стать очень большим. Если транзакции остаются открытыми в течение длительного времени (даже если они не делают ничего полезного) и используют уровень изоляции транзакций `REPEATABLE READ` по умолчанию, InnoDB не сможет удалить старые версии строк, поскольку они должны быть видны незакоммиченным транзакциям. InnoDB хранит старые версии в табличном пространстве, поэтому оно продолжает расти по мере обновления данных. Процесс очистки является многопоточным, но может потребоваться его настройка для ваших рабочих нагрузок, если возникают проблемы с задержкой очистки (`innodb_purge_threads` и `innodb_purge_batch_size`).

В любом случае команда `SHOW INNODB STATUS` может помочь вам идентифицировать причину проблемы. Посмотрите на длину списка историй в разделе `TRANSACTIONS` — она покажет размер журнала отмены в страницах:

```
-----  
TRANSACTIONS  
-----  
Trx id counter 1081043769321  
Purge done for trx's n:o < 1081041974531 undo n:o < 0 state: running but idle  
History list length 697068
```

Если у вас есть большой журнал отмен и по этой причине табличное пространство растет, то можно принудительно замедлить работу MySQL настолько, чтобы поток очистки InnoDB справлялся с нагрузкой. Это может показаться не слишком привлекательным, но альтернативы нет. В противном случае InnoDB будет продолжать записывать данные и заполнять диск до тех пор, пока на нем не закончится место или табличное пространство не достигнет установленных вами пределов.

Чтобы притормозить запись, установите для переменной `innodb_max_purge_lag` значение, отличное от 0. Оно указывает максимальное количество транзакций, которые могут ожидать очистки, прежде чем InnoDB начнет задерживать выполнение дальнейших запросов, обновляющих данные. Чтобы выбрать подходящее значение, нужно знать конкретную рабочую нагрузку. Например, если типичная транзакция изменяет в среднем 1 Кбайт данных и вы можете допустить 100 Мбайт невычищенных строк в своем табличном пространстве, можете установить значение `100000`.

Имейте в виду, что наличие невычищенных версий строк отражается на всех запросах, поскольку из-за них фактически увеличивается размер ваших таблиц и индексов. Если поток очистки просто не справляется с нагрузкой, производительность может заметно снизиться. Установка параметра `innodb_max_purge_lag` также снизит скорость выполнения запросов, но это меньшее из двух зол!

Прочие параметры настройки ввода/вывода

Параметр `sync_binlog` управляет тем, как MySQL сбрасывает двоичный журнал на диск. Его значение по умолчанию равно 1, что означает: MySQL выполнит очистку и сохранит двоичные журналы надежными и безопасными. Это рекомендуемая настройка, и мы предостерегаем вас от установки любого другого значения.

Если вы не зададите для `sync_binlog` значение 1, вполне вероятно, что сбой приведет к тому, что двоичный журнал не будет синхронизирован с транзакционными данными. Это может легко нарушить репликацию и сделать восстановление невозможным, особенно если ваши базы данных используют глобальные идентификаторы транзакций (подробнее об этом — в главе 9). Безопасность, обеспечиваемая при значении 1, намного перевешивает потери производительности ввода/вывода.

Мы более подробно рассмотрели RAID в главе 4, но здесь стоит повторить, 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 обеспечивать надежное хранение данных. Данные записываются в быстрый кэш с батареей, поэтому его содержимое сохраняется даже при отключении питания. Когда питание восстановится, RAID-контроллер переписывает данные из кэша на диск, прежде чем сделать его доступным для использования. Таким образом, хороший RAID-контроллер с достаточно большим кэшем записи и питанием от батареи может значительно повысить производительность, а поэтому становится отличным капиталовложением. Конечно же, и твердотельное хранилище является рекомендуемым решением на этом этапе и значительно повышает производительность ввода/вывода.

Настройка конкурентного доступа в MySQL

Когда вы запускаете MySQL на рабочей нагрузке с высокой степенью конкурентного доступа, вы можете столкнуться с узкими местами, которые не встречаются при других условиях. В этом разделе объясняется, как распознать подобные проблемы, как они возникают и как добиться максимально возможной производительности при таких рабочих нагрузках.

Если вы сталкиваетесь с проблемами, вызванными конкурентным доступом в InnoDB, и не используете по крайней мере MySQL 5.7, решение обычно заключается в обновлении сервера. В более ранних версиях было много проблем с масштабируемостью конкурентного доступа. Все процессы выстраивались

в очереди к глобальным мьютексам, например к мьютексу буферного пула, и сервер практически останавливался. Если вы переходите на одну из более новых версий MySQL, в большинстве случаев ограничивать конкурентный доступ не нужно.

Если вы обнаружите, что столкнулись с этим узким местом, лучший вариант — раздробить данные. Если сегментирование — нежизнеспособный вариант, вам может потребоваться ограничить конкурентный доступ. В InnoDB имеется собственный планировщик потоков, который управляет тем, как потоки входят в ядро для доступа к данным, и тем, что они могут делать, когда находятся внутри ядра. Самый простой способ ограничить степень параллелизма — использовать переменную `innodb_thread_concurrency`, ограничивающую количество потоков, которые могут одновременно находиться в ядре. Значение 0 показывает, что количество потоков не ограничено. При возникновении проблем с конкурентным доступом в InnoDB в старых версиях MySQL на эту переменную следует обратить внимание в первую очередь.

Онлайн-документация предоставляет хорошее руководство по настройке MySQL. Вам придется поэкспериментировать, чтобы найти наилучшее значение для своей системы, но мы рекомендуем вначале установить для `innodb_thread_concurrency` то количество ядер ЦП, которое у вас имеется, а затем начать настройку по мере необходимости.

Если в ядре уже находится больше допустимого количества потоков, то никакой другой поток не может войти в него. InnoDB использует двухфазный процесс, смысл которого заключается в том, чтобы сделать вход в ядро максимально эффективным. Двухфазный процесс снижает накладные затраты на переключение контекста, свойственное планировщику операционной системы. Поток сначала засыпает на `innodb_thread_sleep_delay` микросекунд, а затем пытается снова войти в ядро. Если он по-прежнему не может войти, он переходит в очередь ожидающих потоков и уступает управление операционной системе.

По умолчанию время ожидания на первом шаге составляет 10 000 мкс. Изменение этого значения может помочь в средах с высокой степенью конкуренции, когда процессор не задействуется полностью, потому что большое количество потоков находится в состоянии ожидания перед входом в очередь. К тому же принимаемое по умолчанию значение может быть слишком большим, если у вас много небольших запросов, потому что это увеличивает задержку запроса.

У потока, оказавшегося внутри ядра, есть определенное количество «билетов», позволяющих ему «бесплатно» вернуться в ядро без каких-либо проверок условий конкуренции. Тем самым налагаются ограничения на объем работы, которую поток может выполнить, прежде чем ему придется вернуться в очередь наравне с другими ожидающими потоками. Параметр `innodb_concurrency_tickets`

управляет количеством «билетов». Его редко нужно изменять — разве что в случае, когда имеется большое количество чрезвычайно длительных запросов. «Билеты» выдаются за запрос, а не за транзакцию. После завершения запроса все его неиспользованные «билеты» аннулируются.

В дополнение к узким местам, возникающим из-за буферного пула и других структур, существует еще одно узкое место конкуренции на этапе коммита транзакции, которое в значительной степени связано с вводом/выводом и вызвано операциями сброса. Переменная `innodb_commit_concurrency` определяет количество потоков, которые могут одновременно выполнять коммит. Настройка этого параметра может помочь, если из-за слишком низкого значения переменной `innodb_thread_concurrency` потоки начинают пробуксовывать.

Настройки безопасности

После того как вы установили основные параметры конфигурации, можете включить несколько параметров, которые сделают сервер более безопасным и надежным. Некоторые из них влияют на производительность, поскольку обеспечение безопасности и надежности, как правило, довольно затратно. Однако некоторые из них просто разумны: они предотвращают глупые ошибки, такие как вставка бессмысленных данных на сервер. При этом некоторые не влияют на повседневную работу, но защищают от серьезных проблем в экстремальных случаях.

Сначала рассмотрим набор полезных параметров конфигурации общего поведения сервера.

- **max_connect_errors.** Если в какой-то момент во время работы что-то пойдет не так с вашей сетью, могут возникнуть различные проблемы. Это могут быть ошибки приложения или конфигурации. Может появиться и проблема, которая препятствует успешному завершению соединений в течение короткого времени: клиенты могут быть заблокированы и не смогут снова подключиться, пока вы не очистите кэш хоста. Значение по умолчанию данного параметра (100) настолько мало, что это может произойти очень легко. Вы можете увеличить его, а если уверены, что сервер нельзя взломать простым перебором, то есть знаете, что сервер достаточно хорошо защищен от атак грубой силы, можете просто сделать его очень большим, чтобы эффективно решить проблему блокировки хостов из-за ошибок подключения. Однако если параметр `skip_name_resolve` включен, параметр `max_connect_errors` не даст никакого эффекта, поскольку его поведение зависит от кэша хоста, который отключается параметром `skip_name_resolve`.
- **max_connections.** Этот параметр действует как аварийный тормоз, предохраняя ваш сервер от перегрузки из-за переполнения потоком подключений из

приложения. Если последнее работает неправильно или сервер сталкивается с такой проблемой, как зависание, может быть открыто много новых подключений. Но открытие соединения бесполезно, если оно не может выполнять запросы, поэтому отказ с ошибкой «слишком много соединений» — это способ быстро и просто выйти из строя.

Установите `max_connections` достаточно высоким, чтобы выдержать обычную нагрузку, которую, по вашим оценкам, вы будете испытывать, а также увеличьте запас прочности, связанный с дополнительной нагрузкой при входе в систему и администрировании сервера. Например, если вы думаете, что при обычной работе у вас будет около 300 подключений, то можете установить это значение на 500 или около того. Даже если не знаете, сколько в среднем будет соединений, 500 в любом случае не является необоснованной отправной точкой. Значение по умолчанию — 151, но этого недостаточно для многих приложений.

Остерегайтесь сюрпризов, которые могут заставить вас превысить лимит подключений. Например, если вы перезапустите сервер приложений, он может не закрыть свои подключения корректно, а MySQL может не понять, что они были закрыты. Когда сервер приложений перезагрузится в исходное состояние и попытается открыть соединения с базой данных, ему может быть отказано из-за мертвых соединений, время ожидания которых еще не истекло. Это может иметь значение и в случае, если вы не используете постоянные соединения и ваше приложение не отключается корректно. Сервер будет поддерживать соединение до тех пор, пока не будет достигнут TCP-таймаут, или в худшем случае до тех пор, пока не истечет количество секунд, установленное с помощью параметра `wait_timeout`.

Постоянно наблюдайте за переменной состояния `max_used_connections` с течением времени. Это наивысшая отметка, которая показывает, был ли в какой-то момент у сервера пик подключений. Если значение этой переменной достигает `max_connections`, высока вероятность того, что клиенту было отказано в доступе хотя бы один раз.

- `skip_name_resolve`. Этот параметр нейтрализует другую ловушку, связанную с сетью и аутентификацией, — поиск DNS. DNS является одним из слабейших звеньев в процессе подключения MySQL. Когда вы подключаетесь к серверу, по умолчанию он пытается определить имя хоста, с которого вы это делаете, и задействует его как часть учетных данных аутентификации (то есть ваши учетные данные — это не только имя пользователя и пароль, но и имя хоста). Но, чтобы проверить ваше имя хоста, сервер должен выполнить обратный и прямой поиск DNS. Все это нормально до тех пор, пока у DNS не начнутся проблемы, что в какой-то момент вполне может произойти. Когда это случается, все накапливается и в конце концов время соединения заканчивается. Чтобы предотвратить это, мы настоятельно рекомендуем установить данный

параметр, который отключает поиск DNS во время аутентификации. Однако, если вы сделаете это, потребуется преобразовать разрешения, основанные на имени хоста, на использование IP-адресов джокерных символов или специального имени хоста `localhost`, поскольку учетные записи на основе имени хоста будут отключены.

- `sql_mode`. Этот параметр может принимать различные значения, изменяющие поведение сервера. Мы не рекомендуем менять их просто ради любопытства, в большинстве случаев лучше всего позволить MySQL быть самой собой и не пытаться заставить ее вести себя как другие серверы баз данных. (Например, многие клиентские инструменты и инструменты с графическим интерфейсом ожидают, что MySQL будет иметь собственный диалект SQL, поэтому, если вы измените ее, сделав более ANSI-совместимой, что-то может нарушиться.) Однако некоторые из этих настроек весьма полезны, а некоторые, возможно, стоит рассмотреть в особых случаях. В прошлом MySQL, как правило, очень легко относилась к `sql_mode`, но в более поздних версиях стала более строгой.

Однако имейте в виду, что изменять эти настройки для существующих приложений может быть не очень удачным шагом, поскольку это может сделать сервер несовместимым с ожиданиями приложения. Например, довольно часто люди невольно пишут запросы, в том числе с использованием агрегатных функций, которые ссылаются на столбцы, не входящие в предложение `GROUP BY`. Поэтому, если вы хотите включить параметр `ONLY_FULL_GROUP_BY`, рекомендуется сделать это сначала на сервере, применяемом для разработки, или на вспомогательном сервере, убедиться, что все работает, и только потом разворачивать приложение в рабочей среде.

Кроме того, обязательно проверяйте наличие изменений в `sql_mode` по умолчанию, когда планируете обновление своих баз данных. Изменения в этой переменной могут быть несовместимы с существующим приложением, и вам необходимо заранее протестировать это. Подробнее об обновлении мы поговорим в приложении А.

- `sysdate_is_now`. Это еще один параметр, который может быть несовместим с ожиданиями приложений. Но если вы явно не хотите, чтобы функция `SYSDATE()` имела недетерминированное поведение, которое в определенный момент времени может нарушить репликацию и сделать ненадежным восстановление из резервных копий, то можете включить этот параметр и сделать поведение функции детерминированным.
- `read_only` и `super_read_only`. Параметр `read_only` запрещает непривилегированным пользователям вносить изменения в подчиненные серверы (реплики), которые должны получать изменения только через репликацию, а не из приложения. Мы настоятельно рекомендуем настроить реплики в режиме «только для чтения».

Существует более строгий параметр только для чтения, `super_read_only`, который не позволяет даже пользователям с привилегией `SUPER` записывать данные. Если этот параметр включен, единственное, что может записывать изменения в вашу базу данных, — это репликация. Мы настоятельно рекомендуем включить и `super_read_only`. Это предотвратит случайное применение учетной записи администратора для записи данных в реплику, доступную только для чтения, что приведет к ее рассинхронизации.

Дополнительные настройки InnoDB

Некоторые из рассмотренных ранее параметров настройки InnoDB очень важны для производительности сервера, а несколько других параметров обеспечивают безопасность.

- `innodb_autoinc_lock_mode`. Этот параметр управляет тем, как InnoDB генерирует автоматически увеличивающиеся значения первичного ключа, что в некоторых случаях может стать узким местом, например при вставках с высокой степенью конкурентности. Если у вас много транзакций, ожидающих блокировки автоинкремента (это можно увидеть в выводе команды `SHOW ENGINE INNODB STATUS`), вам следует изучить этот параметр. Мы не будем повторять объяснение этого параметра и его функций, приведенное в руководстве пользователя.
- `innodb_buffer_pool_instances`. В MySQL 5.5 и более новых версиях этот параметр делит буферный пул на несколько сегментов и, вероятно, является одним из наиболее важных способов повышения масштабируемости MySQL на многоядерных машинах с высококонкурентной рабочей нагрузкой. Несколько буферных пулов разделяют рабочую нагрузку, поэтому некоторые глобальные мьютексы не становятся горячими точками конкуренции.
- `innodb_io_capacity`. Ранее InnoDB разрабатывался в предположении, что он работает на одном жестком диске, способном выполнять 100 операций ввода/вывода в секунду. Это была неудачная настройка по умолчанию. Теперь вы можете сообщить InnoDB производительность доступной системы ввода/вывода. Иногда необходимо установить большое значение этого параметра (десятки тысяч на чрезвычайно быстром хранилище, таком как флеш-устройства PCIe) для устойчивого сброса «грязных» страниц по причинам, которые довольно сложно объяснить¹.

¹ В качестве дополнительного чтения смотрите сообщения в блоге Percona: Give Love to Your SSDs — Reduce `innodb_io_capacity_max`, InnoDB Flushing in Action for Percona Server for MySQL и MySQL/InnoDB Flushing for a Write-Intensive Workload (<https://oreil.ly/CdzsQ>).

- `innodb_read_io_threads` и `innodb_write_io_threads`. Эти параметры определяют количество фоновых потоков, доступных для операций ввода/вывода. По умолчанию в последних версиях MySQL имеется четыре потока чтения и четыре потока записи, которых вполне достаточно для большинства серверов, особенно при реализации естественного асинхронного ввода/вывода, доступного, начиная с MySQL 5.5. Если у вас много жестких дисков, рабочая нагрузка обладает высокой степенью конкурентности и вы видите, что потоки с трудом справляются с ней, можете увеличить количество потоков или просто установить его равным числу физических шпинделей, которые используются для ввода/вывода (даже если они находятся за RAID-контроллером).
- `innodb_strict_mode`. Этот параметр заставляет InnoDB в некоторых условиях выдавать ошибки вместо предупреждений, особенно при недопустимых или даже потенциально опасных параметрах `CREATE TABLE`. Если вы включите этот параметр, обязательно проверьте все команды `CREATE TABLE`, потому что это может не позволить вам создавать таблицы, которые раньше отлично создавались. Иногда это немного огорчает и становится слишком большим ограничением. Вы вряд ли захотите узнать об этом лишь тогда, когда будете пытаться восстановить данные из резервной копии.
- `innodb_old_blocks_time`. В InnoDB есть состоящий из двух частей список недавно использованного буферного пула (least recently used, LRU), предназначенный для предотвращения нежелательных запросов от вытесняемых страниц, которые применялись много раз в течение длительного времени. Разовый запрос, подобный тем, которые выполняет утилита `mysqldump`, как правило, заносит страницу в список LRU буферного пула, считывает из него строки и переходит к следующей странице. Теоретически состоящий из двух частей список LRU не позволит этой странице вытеснить страницы, которые будут нужны в течение длительного времени, помещая ее в «молодой» подсписок и перемещая в «старый» подсписок только после многократного обращения к ней. Но InnoDB не настроен по умолчанию на такое поведение, поскольку страница содержит несколько строк, и, следовательно, множественный доступ для чтения этих строк со страницы приведет к ее немедленному перемещению в «старый» подсписок и вытеснению страниц, которые требуются в течение длительного времени. Эта переменная указывает количество миллисекунд, которое должно пройти, прежде чем страница сможет перейти из «молодой» части списка LRU в «старую». Ее значение по умолчанию — 0, и установка небольшого значения, например 1000 (1 с), оказалась очень эффективной при эталонном тестировании.

Резюме

Проработав эту главу, вы сможете сконфигурировать свой сервер намного лучше, чем предусмотрено настройками по умолчанию. Сервер должен быть быстрым и стабильным, и у вас не должно возникать необходимости подправлять его конфигурацию, если только вы не столкнетесь с чрезвычайными обстоятельствами.

Для ознакомления предлагаем начать с приведенного здесь образца файла конфигурации, установить основные параметры своего сервера и рабочей нагрузки, а также добавить желаемые параметры. Это действительно все, что вам нужно сделать.

Если вы задействуете выделенный сервер базы данных, то лучшим вариантом установки является `innodb_dedicated_server`, который управляет 90 % конфигурации производительности. Если вы не можете использовать этот параметр, то наиболее важными станут следующие два:

- `innodb_buffer_pool_size`;
- `innodb_log_file_size`.

Поздравляем — вы только что решили подавляющее большинство реальных проблем конфигурации, с которыми мы встречались на практике!

Мы также сделали много предостережений о том, чего не следует делать. Наиболее важные из них следующие:

- не делайте «тонкую настройку» сервера;
- не используйте коэффициенты, формулы или настроечные скрипты в качестве основы для установки переменных конфигурации.

ГЛАВА 6

Разработка схемы и управление

Хорошее логическое и физическое проектирование схемы является краеугольным камнем высокой производительности. Следует разрабатывать свою схему для конкретных запросов, которые вы будете выполнять. При этом зачастую приходится идти на компромиссы. Например, денормализованная схема может ускорить выполнение некоторых типов запросов, но замедлить выполнение других. Добавление таблиц счетчиков и сводных таблиц — отличный способ оптимизировать запросы, но их поддержка может оказаться дорогостоящей. Конкретные функции MySQL и детали реализации незначительно влияют на это.

Ваша схема также будет развиваться со временем — когда вы еще больше узнаете о том, как храните данные и получаете к ним доступ, а также как со временем меняются бизнес-требования. Это означает, что вы должны планировать изменения схемы как частое событие. Далее в этой главе мы поможем вам понять, как не допустить, чтобы эти операции не стали узким местом в работе организации.

В этой и следующей главах, посвященных индексированию, рассматриваются специфические для MySQL аспекты проектирования схемы. Мы предполагаем, что вы знаете, как проектировать базы данных, так что это не вводная и даже не продвинутая глава по проектированию баз данных. Глава, посвященная проектированию баз данных с помощью MySQL, рассматривает этот процесс в сравнении с использованием других СУБД. Если вам нужно изучить основы проектирования баз данных, предлагаем книгу Клэр Черчер *Beginning Database Design* (Apress).

Данная глава подготовит вас к прочтению двух последующих. В этих трех главах мы рассмотрим взаимодействие логического и физического проектирования и выполнения запросов. Для этого потребуется как взгляд на картину в целом, так и внимание к деталям. Вам нужно разобраться во всей системе, чтобы понять, как каждая часть повлияет на другие. Возможно, полезно будет перечитать эту главу после прочтения главы 7 об индексировании и главы 8 об оптимизации запросов. Большинство из обсуждаемых тем нельзя рассматривать изолированно.

Выбор оптимальных типов данных

MySQL поддерживает большое разнообразие типов данных, и выбор правильного типа для хранения ваших данных имеет решающее значение для обеспечения хорошей производительности. Следующие простые рекомендации помогут сделать правильный выбор независимо от типа данных, которые вы храните.

- *Меньше обычно лучше.* В целом старайтесь использовать типы данных минимального размера, который может правильно хранить и представлять ваши данные. Меньшие по размеру типы данных обычно работают быстрее, потому что занимают меньше места на диске, в памяти и кэше процессора. Кроме того, для их обработки обычно требуется меньше процессорного времени.

Однако убедитесь, что правильно представляете диапазон возможных значений данных, которые необходимо хранить, поскольку увеличение размерности типа данных в нескольких местах схемы может быть болезненной и трудоемкой операцией. Если вы сомневаетесь в том, какой тип данных лучше использовать, выберите самый короткий при условии, что его размера будет достаточно. (Если система не очень загружена или не хранит много данных или если вы находитесь на ранней стадии проектирования, позже можно легко изменить решение.)

- *Просто — значит хорошо.* Для выполнения операций с более простыми типами данных обычно требуется меньше процессорного времени. Например, сравнение целых чисел менее затратно, чем сравнение символов, потому что различные кодировки и схемы упорядочения (правила сортировки) усложняют сравнение символов. Вот два примера: вы должны хранить значения даты и времени во встроенных типах MySQL, а не в виде строк. Для IP-адресов следует использовать целочисленные типы данных. Мы обсудим эти темы позже.
- *По возможности избегайте значений NULL.* Очень часто в таблицах встречаются поля, допускающие хранение NULL (отсутствие значения), хотя приложению это совершенно не нужно просто потому, что такой режим установлен по умолчанию. Обычно лучше объявить столбец как NOT NULL, если только вы не собираетесь хранить в них NULL. MySQL сложнее оптимизировать запросы, которые ссылаются на столбцы, допускающие значение NULL, потому что из-за них усложняются индексы, статистика индексов и сравнение значений. Столбец, допускающий значение NULL, использует больше места для хранения на диске и требует специальной обработки внутри MySQL. Повышение производительности от изменения столбцов NULL на NOT NULL обычно невелико, поэтому не делайте приоритетом их поиск и изменение в существующей схеме, если только вы не уверены, что именно они вызывают проблемы.

Первым шагом в принятии решения о том, какой тип данных использовать для данного столбца, является определение подходящего общего класса

типов: числовые, строковые, временные и т. д. Обычно это довольно просто, но бывают особые случаи, когда выбор интуитивно неочевиден.

Следующим шагом является выбор конкретного типа. Многие типы данных MySQL позволяют хранить данные одного и того же типа, но различаются диапазоном значений, которые они могут хранить, точностью, которую допускают, или требуют разного физического пространства на диске и в памяти. Кроме того, некоторые типы данных выделяются особым поведением или свойствами.

Например, в столбцах `DATETIME` и `TIMESTAMP` можно хранить один и тот же тип данных: дату и время с точностью до 1 с. Однако `TIMESTAMP` использует вдвое меньше места для хранения, позволяет работать с часовыми поясами и обладает специальными возможностями автоматического обновления. В то же время у него гораздо меньший диапазон допустимых значений, и иногда его особые возможности могут стать помехой.

Здесь мы обсудим базовые типы данных. В целях совместимости MySQL поддерживает множество псевдонимов, таких как `INTEGER` (соотносится с `INT`), `BOOL` (соотносится с `TINYINT`) и `NUMERIC` (соотносится с `DECIMAL`). Все это именно псевдонимы. Данный факт может сбить с толку, но не влияет на производительность. Если вы создадите таблицу с псевдонимом типа данных, а затем просмотрите `SHOW CREATE TABLE`, то увидите, что MySQL сообщает базовый тип, а не использованный псевдоним.

Целые числа

Есть два типа чисел: целые и вещественные (числа с дробной частью). Для хранения целых чисел используйте один из целочисленных типов: `TINYINT`, `SMALLINT`, `MEDIUMINT`, `INT` или `BIGINT`. Для хранения они требуют 8, 16, 24, 32 и 64 бита дискового пространства соответственно. Они позволяют хранить значения в диапазоне от $-2(N-1)$ до $2(N-1)-1$, где N — количество битов пространства для хранения, которое они используют.

Целочисленные типы могут иметь необязательный атрибут `UNSIGNED`, который запрещает отрицательные значения и приблизительно удваивает верхний предел положительных значений, которые вы можете хранить. Например, `TINYINT UNSIGNED` позволяет хранить значения в диапазоне от 0 до 255, а не от -128 до 127.

Знаковые и беззнаковые типы требуют одинакового пространства и обладают одинаковой производительностью, поэтому используйте тот тип, который лучше всего подходит для вашего диапазона данных.

Ваш выбор определяет то, как MySQL хранит данные в памяти и на диске. Однако при целочисленных вычислениях обычно применяются 64-битные целые числа типа **BIGINT**. (Исключением являются определенные агрегатные функции, которые для выполнения вычислений используют тип **DECIMAL** или **DOUBLE**.) MySQL позволяет указать размер для целочисленных типов, например **INT(11)**. Это неважно для большинства приложений, так как не ограничивает разрешенный диапазон значений, а просто указывает количество символов, которое интерактивным инструментам MySQL, таким как клиент командной строки, необходимо зарезервировать для вывода числа. С точки зрения хранения и вычислений **INT(1)** идентичен **INT(20)**.

Вещественные числа

Вещественные числа — это числа, имеющие дробную часть. Однако они предназначены не только для дробных чисел — можно использовать **DECIMAL** также для хранения целых чисел, которые настолько велики, что не помещаются в **BIGINT**. MySQL поддерживает как «точные», так и «неточные» типы.

Типы **FLOAT** и **DOUBLE** поддерживают приблизительные вычисления со стандартной математикой с плавающей точкой. Если вам нужно точно знать, как рассчитываются результаты с плавающей точкой, изучите, как эти вычисления реализованы на имеющейся у вас платформе. Вы можете указать желаемую точность столбца с плавающей точкой несколькими способами, из-за чего MySQL может автоматически выбрать другой тип данных или округлить значения при их сохранении. Эти спецификаторы точности нестандартны, поэтому рекомендуем задавать нужный тип, но не точность.

Типы с плавающей точкой обычно требуют меньше места для хранения одного и того же диапазона значений, чем тип **DECIMAL**. Столбец типа **FLOAT** использует 4 байта памяти. Тип **DOUBLE** занимает 8 байт и имеет большую точность и больший диапазон значений, чем **FLOAT**. Как и в случае работы с целыми числами, вы выбираете только тип для хранения: MySQL использует **DOUBLE** для внутренних вычислений с типами с плавающей точкой.

Из-за дополнительных требований к пространству и затрат на вычисления вы должны задействовать **DECIMAL** только тогда, когда нужны точные результаты при вычислениях с дробными числами, например при хранении финансовых данных. Но при некоторых операциях с большими числами целесообразнее использовать тип **BIGINT** вместо **DECIMAL** и хранить данные как некоторое число, кратное наименьшей доле валюты, которую нужно обрабатывать. Предположим, вам требуется хранить финансовые данные с точностью до десяти тысячной доли

цента. Вы можете умножить все суммы в долларах на миллион и сохранить результат в `BIGINT`, тем самым избегая как неточности хранения типов с плавающей точкой, так и высоких затрат на точные расчеты типа `DECIMAL`.

Строковые типы

MySQL поддерживает довольно много строковых типов данных, каждый из которых имеет множество вариаций. Каждый строковый столбец может иметь собственную кодировку и соответствующую схему упорядочения для этого набора символов.

Типы `VARCHAR` и `CHAR`

Двумя основными типами строк являются `VARCHAR` и `CHAR`, в которых хранятся символьные значения. К сожалению, трудно точно объяснить, как именно эти значения хранятся на диске и в памяти, потому что реализация зависит от выбранной подсистемы хранения. Мы предполагаем, что вы работаете с InnoDB. В противном случае вам следует прочитать документацию по используемой подсистеме хранения.

Посмотрим, как обычно значения `VARCHAR` и `CHAR` хранятся на диске. Имейте в виду, что подсистема хранения может хранить значение `CHAR` или `VARCHAR` в памяти не так, как она хранит это значение на диске, и сервер может преобразовывать значения в другой формат, когда извлекает их из подсистемы хранения. Приведем общее сравнение двух этих типов.

- Тип `VARCHAR` хранит строки символов переменной длины и является наиболее широко используемым строковым типом данных. Для хранения строк этого типа может потребоваться меньше места, чем для типов строк с фиксированной длиной, поскольку задействуется ровно столько места, сколько нужно (то есть меньше места используется для хранения более коротких значений).

В типе `VARCHAR` используются 1 или 2 дополнительных байта для хранения длины значения: 1 байт, если максимальная длина столбца не превышает 255 байт, и 2 байта — для более длинных строк. Если применяется кодировка `latin1`, тип `VARCHAR(10)` будет использовать до 11 байт дискового пространства. Тип `VARCHAR(1000)` может задействовать до 1002 байт, поскольку для хранения информации о длине требуется 2 байта.

Тип `VARCHAR` повышает производительность за счет экономии места. Но поскольку строки имеют переменную длину, они могут увеличиваться при

обновлении, что может потребовать дополнительной работы. Если строка становится длиннее и больше не помещается в отведенное для нее место, то дальнейшее поведение системы зависит от подсистемы хранения. Возможно, InnoDB придется выполнить разбиение страницы, чтобы поместить в нее строку. Другие подсистемы хранения могут никогда не обновлять данные в месте их хранения.

- Тип **CHAR** имеет фиксированную длину: MySQL всегда выделяет место для указанного количества символов. При сохранении значения **CHAR** MySQL удаляет все пробелы в конце строки. Для операций сравнения значения дополняются пробелами при необходимости.

Тип **CHAR** полезен, когда требуется хранить очень короткие строки или все значения имеют приблизительно одинаковую длину. Например, **CHAR** — хороший выбор для хранения MD5-сверток паролей пользователей, которые всегда имеют одинаковую длину. Также тип **CHAR** лучше, чем **VARCHAR**, для данных, которые часто изменяются, потому что строка фиксированной длины не фрагментируется. Если в столбцах хранятся очень короткие строки, тип **CHAR** также более эффективен, чем **VARCHAR**: если тип **CHAR(1)** предназначен для хранения только значений **Y** и **N**, то он использует только 1 байт в однобайтовой кодировке¹, а **VARCHAR(1)** — 2 байта из-за наличия дополнительного байта длины строки.

Такое поведение может несколько сбивать с толку, поэтому проиллюстрируем его на примере. Вначале создаем таблицу с одним столбцом **CHAR(10)** и сохраняем в ней некоторые значения:

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

Когда мы извлекаем значения, конечные пробелы удаляются:

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

```
+-----+
| CONCAT("'", char_col, "'") |
+-----+
| 'string1'                  |
| ' string2'                  |
| 'string3'                  |
+-----+
```

¹ Помните, что длина указывается в символах, а не байтах. При многобайтовой кодировке может потребоваться более 1 байта для хранения каждого символа.

Если мы сохраним те же значения в столбце `VARCHAR(10)`, то после извлечения получим результат, где конечный пробел в строке 3 не удален:

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

Родственными типами для `CHAR` и `VARCHAR` являются `BINARY` и `VARBINARY`, которые хранят двоичные строки. Двоичные строки очень похожи на обычные, но вместо символов в них хранятся байты. Метод заполнения двоичных строк также отличается: MySQL добавляет в строки типа `BINARY` значение `\0` (нулевой байт) вместо пробелов и не удаляет дополненные байты при извлечении¹. Эти типы полезны, когда вам нужно хранить двоичные данные и вы хотите, чтобы MySQL сравнивала значения как байты, а не как символы. Преимущество побайтового сравнения заключается не только в нечувствительности к регистру. MySQL сравнивает строки `BINARY` побайтно в соответствии с числовым значением каждого байта. В результате двоичное сравнение может оказаться намного проще, чем сравнение символов, и, как следствие, выполняется быстрее.

ЩЕДРОСТЬ НЕ ВСЕГДА РАЗУМНА

Для хранения значения `'hello'` требуется одинаковое место в столбцах `VARCHAR(5)` и `VARCHAR(200)`. Есть ли преимущество в использовании более короткого столбца?

Как оказалось, преимущество есть, и довольно значительное. Для большего столбца может потребоваться гораздо больше памяти, потому что MySQL часто выделяет для внутреннего хранения значений блоки памяти фиксированного размера. Это особенно плохо для сортировки или операций, использующих временные таблицы в памяти. То же самое происходит при файловой сортировке, задействующей временные таблицы на диске. Наилучшей стратегией будет выделять ровно столько места, сколько вам действительно нужно.

Типы BLOB и TEXT

`BLOB` и `TEXT` — это строковые типы данных, предназначенные для хранения больших объемов данных в виде двоичных или символьных строк соответственно.

На самом деле каждое из них представляет собой семейство типов данных: типы символов `TINYTEXT`, `SMALL TEXT`, `TEXT`, `MEDIUMTEXT` и `LONGTEXT`, а также двоичные

¹ Будьте осторожны с типом `BINARY`, если значение после извлечения должно оставаться неизменным. MySQL дополнит его до необходимой длины с помощью `\0s`.

типы `TINYBLOB`, `SMALL BLOB`, `BLOB`, `MEDIUMBLOB` и `LONGBLOB`. `BLOB` — это синоним `SMALLBLOB`, а `TEXT` — синоним `SMALLTEXT`.

В отличие от остальных типов данных MySQL обрабатывает каждое значение `BLOB` и `TEXT` как отдельный объект с собственным идентификатором. Подсистемы хранения часто хранят их особым образом: InnoDB может использовать для них отдельную внешнюю область хранения, если они большого размера. Для каждого значения требуется от 1 до 4 байт дискового пространства в самой строке и достаточно места во внешнем хранилище для фактического хранения значения.

Единственная разница между семействами `BLOB` и `TEXT` заключается в том, что типы `BLOB` хранят двоичные данные без учета схемы упорядочения и кодировки, а типы `TEXT` используют схемы упорядочения и кодировку.

MySQL сортирует столбцы `BLOB` и `TEXT` не так, как столбцы других типов: вместо сортировки строки по всей длине она сортирует только по первым `max_sort_length` байтам каждого столбца. Если требуется отсортировать только по первым нескольким символам, можете уменьшить значение серверной переменной `max_sort_length`.

MySQL не может индексировать данные этих типов по полной длине и не может использовать индексы для сортировки.

ИЗОБРАЖЕНИЯ В БАЗЕ ДАННЫХ?

В прошлом некоторые приложения нередко принимали загруженные изображения и сохраняли их как данные `BLOB` в базе данных MySQL. Этот метод был удобен для хранения данных для приложения вместе, однако по мере увеличения размера данных такие операции, как изменение схемы, становились все медленнее и медленнее из-за размера этих данных `BLOB`.

Не храните данные, такие как изображения, в базе данных, если можете избежать этого. Лучше запишите их в отдельное хранилище данных объектов и используйте таблицу для отслеживания местоположения или имени файла изображения.

Применение ENUM вместо строкового типа

Иногда можно использовать столбец типа `ENUM` вместо обычных строковых типов. Столбец `ENUM` может хранить predetermined набор различных строковых значений. MySQL хранит их очень компактно, упаковывая в 1 или 2 байта в зависимости от количества значений в списке. Она хранит каждое значение внутри как целое число, представляющее его позицию в списке определений полей. Приведем пример:

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

Три строки на самом деле хранят целые числа, а не строки. Вы можете увидеть двойственную природу значений, извлекая их в числовом контексте:

```
mysql> SELECT e + 0 FROM enum_test;
```

```
+-----+
| e + 0 |
+-----+
|      1 |
|      3 |
|      2 |
+-----+
```

Эта двойственность может вызвать путаницу, если вы определите числа в качестве констант ENUM, например ENUM('1', '2', '3'). Мы не рекомендуем так делать.

Еще одним сюрпризом является то, что поле ENUM сортируется по внутренним целочисленным значениям, а не по самим строкам:

```
mysql> SELECT e FROM enum_test ORDER BY e;
```

```
+-----+
| e      |
+-----+
| fish   |
| apple  |
| dog    |
+-----+
```

Проблему можно решить, определив значения для столбца ENUM в желаемом порядке сортировки. Вы также можете использовать FIELD() для явного указания порядка сортировки в своих запросах, но это не позволит MySQL применять индекс для сортировки:

```
mysql> SELECT e FROM enum_test ORDER BY FIELD(e, 'apple', 'dog', 'fish');
```

```
+-----+
| e      |
+-----+
| apple  |
| dog    |
| fish   |
+-----+
```

Если бы мы определили значения в алфавитном порядке, нам не нужно было бы это делать.

Поскольку MySQL хранит каждое значение как целое число и вынуждена просматривать таблицы соответствий для преобразования их в строковое представление, то со столбцами типа ENUM связаны некоторые накладные расходы. Обычно это компенсируется их малым размером, но так происходит не всегда.

В частности, соединение столбца `CHAR` или `VARCHAR` со столбцом типа `ENUM` может оказаться медленнее, чем соединение с другим столбцом типа `CHAR` или `VARCHAR`.

Чтобы проиллюстрировать данное утверждение, мы провели эталонное тестирование того, как быстро MySQL выполняет такое соединение с таблицей в одном из наших приложений. В таблице определили довольно большой 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 занимает всего около 10 Мбайт, поэтому полностью помещается в памяти. Столбец `service` содержит 5 различных значений со средней длиной 4 символа, а столбец `method` содержит 71 значение со средней длиной 20 символов.

Мы сделали копию этой таблицы и преобразовали столбцы `service` и `method` в `ENUM` следующим образом:

```
CREATE TABLE webservicecalls_enum (  
  ... omitted ...  
  service ENUM(...values omitted...) NOT NULL,  
  method ENUM(...values omitted...) NOT NULL,  
  ... omitted ...  
) ENGINE=InnoDB;
```

Затем измерили производительность соединения таблиц по столбцам первичного ключа. Вот использованный запрос:

```
mysql> SELECT SQL_NO_CACHE COUNT(*)  
  -> FROM webservicecalls  
  -> JOIN webservicecalls USING(day, account, service, method);
```

Мы варьировали его, соединяя столбцы типа `VARCHAR` и `ENUM` в разных комбинациях. В табл. 6.1 показаны результаты¹.

¹ Время указано для относительного сравнения, так как скорость процессоров, памяти и другого оборудования постепенно меняется.

Таблица 6.1. 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 выполняется быстрее после преобразования столбцов к типу ENUM, но соединение столбцов типа ENUM со столбцами типа VARCHAR происходит медленнее. В данном случае кажется хорошей идеей преобразовать эти столбцы, если не планируется их соединение со столбцами типа VARCHAR. Распространенная практика проектирования состоит в использовании справочных таблиц с целочисленными первичными ключами, чтобы избежать применения соединения по символьным значениям.

Однако у преобразования столбцов есть еще одно преимущество: 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 быть полезно, даже если столбцы ENUM должны быть объединены со столбцами VARCHAR. Кроме того, после преобразования размер самого первичного ключа уменьшается примерно наполовину. Поскольку это таблица InnoDB, если в ней есть какие-либо другие индексы, уменьшение размера первичного ключа сделает и их намного меньше.



Хотя типы ENUM очень эффективны в части хранения значений данных, изменения допустимых значений в ENUM всегда требуют изменения схемы. Необходимость частого изменения значений в ENUM может доставлять серьезные неудобства, если у вас еще нет надежной системы автоматизации изменения схемы. Такая система описывается далее в этой главе. Позже мы рассмотрим также антипаттерн «Слишком много ENUM» в дизайне схемы.

Типы Date и Time

MySQL имеет много типов для различных значений даты и времени, таких как `YEAR` и `DATE`. Минимальная единица времени, которую MySQL может хранить, — 1 мкс. У большинства временных типов нет альтернатив, поэтому вопрос о том, какой из них лучше, не возникает. Нужно лишь решить, что делать, когда нужно хранить и дату, и время. MySQL предлагает для этой цели два очень похожих

типа данных: `DATETIME` и `TIMESTAMP`. Для большинства приложений подойдет любой из них, но в некоторых случаях один работает лучше, чем другой. Рассмотрим их подробнее.

- **DATETIME.** Этот тип дает возможность хранить большой диапазон значений с 1000 по 9999 год с точностью до 1 мкс. Он хранит дату и время, упакованные в целое число в формате `YYYYMMDDHHMMSS`, независимо от часового пояса. Этот тип данных использует 8 байт дискового пространства.

По умолчанию MySQL отображает значения типа `DATETIME` в точно определенном допускающем сортировку формате `2008-01-16 22:37:08`. Это стандартный способ представления даты и времени ANSI.

- **TIMESTAMP.** Как следует из названия, в типе `TIMESTAMP` хранится количество секунд, прошедших после полуночи 1 января 1970 года по Гринвичу (GMT), — как во временной метке Unix. Для хранения данных типа `TIMESTAMP` используются только 4 байта памяти, поэтому он позволяет представить значительно меньший диапазон дат, чем тип `DATETIME`: с 1970 года по 19 января 2038 года. В MySQL имеются функции `FROM_UNIXTIME()` и `UNIX_TIMESTAMP()`, преобразующие временную метку Unix в дату и наоборот.

Таким образом, если в поле типа `TIMESTAMP` хранится значение 0, то для часового пояса Eastern Standard Time (EST), отличающегося от гринвичского времени на 5 часов, будет выведена строка `1969-12-31 19:00:00`. Стоит подчеркнуть эту разницу: если вы храните данные, относящиеся к нескольким часовым поясам, или получаете к ним доступ, поведение `TIMESTAMP` и `DATETIME` будет сильно различаться. Первый сохраняет значения относительно используемого часового пояса, а второй — текстовое представление даты и времени.

Кроме того, тип `TIMESTAMP` имеет специальные свойства, которых нет у типа `DATETIME`. По умолчанию, если вы не указали значение для столбца, MySQL устанавливает для первого столбца `TIMESTAMP` текущее время¹. Кроме того, по умолчанию MySQL изменяет значение первого столбца типа `TIMESTAMP`, когда вы обновляете строку, если ему явно не присвоено значение в операторе `UPDATE`. Можно настроить поведение при вставке и обновлении для любого столбца `TIMESTAMP`. Наконец, столбцы `TIMESTAMP` по умолчанию создаются в режиме `NOT NULL`, в отличие от остальных типов данных.

¹ Правила поведения типа `TIMESTAMP` сложны и неодинаковы в различных версиях MySQL, поэтому следует убедиться, что он ведет себя так, как хочется вам. Обычно рекомендуется проверить вывод команды `SHOW CREATE TABLE` после внесения изменений в столбцы `TIMESTAMP`.

ХРАНЕНИЕ ДАТЫ И ВРЕМЕНИ В ВИДЕ ЦЕЛОГО ЧИСЛА

Типы данных `DATE` и `TIME` заставляют вас иметь дело с часовыми поясами на сервере и клиенте, и хотя тип `TIMESTAMP` более эффективен с точки зрения занимаемого на диске места, чем `DATE` (4 байта против 8 байт и без поддержки дробных долей секунд), он тоже страдает от проблемы 2038 года.

В конечном счете хранение даты и времени сводится к нескольким вещам.

- Насколько большой интервал необходимо поддерживать для даты и времени?
- Какое значение имеет место для хранения этих данных?
- Нужна ли вам поддержка дробных долей секунды?
- Вы хотите переместить обработку даты, времени и часового пояса в MySQL или сделать это в программном коде?

В настоящее время все более популярным становится подход, предлагающий избегать сложностей обработки MySQL и сохранять данные типа даты и времени как эпоху Unix или количество секунд с 1 января 1970 года в универсальном скоординированном времени (UTC). Используя 32-битный `INT` со знаком, вы можете хранить даты до 2038 года. С 32-битным `INT` без знака можно хранить даты до 2106 года. С помощью 64-битного `INT` можно расширить интервал дат.

Подобно другим популярным дискуссиям об операционных системах, редакторах, знаках табуляции и проблемах, предлагаемые варианты хранения конкретных наборов данных — это скорее мнение, чем передовая практика. Оцените, насколько предлагаемый подход приемлем для вашего сценария использования.

Битовые типы данных

MySQL имеет несколько типов хранения, которые применяют для компактного хранения значений данных отдельные биты. Все эти типы с технической точки зрения являются строковыми независимо от основного формата хранения и обработки.

- **BIT.** Вы можете использовать столбец типа `BIT` для хранения одного или нескольких значений `true/false` в одном столбце. `BIT(1)` определяет поле, содержащее 1 бит, `BIT(2)` хранит 2 бита и т. д., максимальная длина столбца типа `BIT` составляет 64 бита. InnoDB хранит каждый столбец как наименьший целочисленный тип, достаточно большой для размещения всех битов, поэтому сэкономить пространство не получится.

MySQL рассматривает `BIT` как строковый, а не числовой тип. Когда вы извлекаете значение `BIT(1)`, результатом является строка, но ее содержимое представляет собой двоичное значение 0 или 1, а не значение 0 или 1 в кодировке ASCII. Но если извлекаете значение в числовом виде, результатом будет число, в которое преобразуется битовая строка. Учитывайте это, если нужно сравнить результат с другим значением. Например, если вы сохраните

значение `b'00111001'` (это двоичный эквивалент 57) в столбце `BIT(8)` и извлечете его, то получите строку, содержащую символ с кодом 57. Это ASCII-код символа 9. Но в числовом контексте получите значение 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, поэтому рекомендуем использовать тип `BIT` с осторожностью. Мы считаем, что для большинства приложений лучше его избегать.

Если вы хотите сохранять значение `true/false` в одном бите памяти, другим вариантом является создание столбца типа `CHAR(0)`, допускающего значение `NULL`. Такой столбец может хранить либо отсутствие значения (`NULL`), либо значение нулевой длины (пустая строка). Это работает, но может быть непонятно для людей, использующих данные в базе данных, и затруднить написание запросов. Если экономия места для вас не приоритетна, стоит применять `TINYINT`.

- **SET.** Если вам нужно хранить много значений `true/false`, рассмотрите возможность объединения нескольких столбцов в один MySQL с типом данных `SET`. В MySQL его внутренним представлением является упакованный битовый вектор. Он эффективно использует пространство, а в MySQL есть такие функции, как `FIND_IN_SET()` и `FIELD()`, которые упрощают применение данных типа `SET` в запросах.

Побитовые операции над целочисленными столбцами. Альтернативой типу `SET` является использование целого числа в качестве упакованного набора битов. Например, можно упаковать 8 бит в `TINYINT` и выполнять с ним побитовые операции. Вы можете упростить эту задачу, определив именованные константы для каждого бита в коде своего приложения.

Главное преимущество такого подхода, по сравнению с использованием типа `SET`, заключается в том, что вы можете изменить «перечисление», которое представляет поле, без команды `ALTER TABLE`. Недостатком является то, что ваши запросы становятся труднее для написания и понимания (что значит, если установлен бит 5?). Некоторым людям удобно работать с побитовыми манипуляциями, а некоторым — нет, поэтому захотите ли вы попробовать описанный метод, во многом зависит от вашего вкуса.

Примером применения упакованных битов является список управления доступом (access control list, ACL), в котором хранятся разрешения. Каждый

бит или элемент SET представляет значение, такое как CAN_READ, CAN_WRITE или CAN_DELETE. Используя столбец типа SET, вы позволите MySQL сохранить сопоставление битов и значений в определении столбца, а если применяете целочисленный столбец, то такое соответствие устанавливается в коде приложения. Приведем примеры запросов 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 |
+-----+
```

Если бы вы использовали целочисленный столбец, то могли бы написать этот пример 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 использовали переменные для определения значений, но в своем коде вместо этого можете задействовать константы.

JSON-данные

Применение JSON в качестве формата для обмена данными между системами становится все более распространенным. MySQL имеет нативный тип данных JSON, который упрощает работу с частями структуры JSON непосредственно в таблице. Сторонники чистого кода могут заявить, что хранение необработанного JSON в базе данных — это антипаттерн, потому что в идеале схемы базы данных — это представление полей в JSON. Новички, глядя на тип данных JSON, решат использовать этот простейший подход, позволяющий избежать создания независимых полей и управления ими. Какой метод лучше, во многом субъективно, но мы для объективности представим пример применения этих подходов и сравним скорость выполнения запроса и размер данных.

Наш пример данных представляет собой список из 202 обнаруженных околоземных астероидов и комет, любезно предоставленный NASA. Тесты проводились на четырехъядерной виртуальной машине с 16 Гбайт ОЗУ с MySQL 8.0.22. Вот пример использованных данных:

```
[
  {
    "designation": "419880 (2011 AH37)",
    "discovery_date": "2011-01-07T00:00:00.000",
    "h_mag": "19.7",
    "moid_au": "0.035",
    "q_au_1": "0.84",
    "q_au_2": "4.26",
    "period_yr": "4.06",
    "i_deg": "9.65",
    "pha": "Y",
    "orbit_class": "Apollo"
  }
]
```

Эти данные представляют собой название, дату обнаружения и сведения об объекте, включая числовые и текстовые поля.

Сначала мы взяли набор данных в формате JSON и преобразовали его в одну строку для каждой записи. В результате получилась относительно простая схема:

```
mysql> DESC asteroids_json;
```

Field	Type	Null	Key	Default	Extra
json_data	json	YES		NULL	

Затем мы взяли этот JSON и преобразовали поля в столбцы, используя подходящий тип данных для полей. В результате получилась следующая схема:

```
mysql> DESC asteroids_sql;
```

Field	Type	Null	Key	Default	Extra
designation	varchar(30)	YES		NULL	
discovery_date	date	YES		NULL	
h_mag	float	YES		NULL	
moid_au	float	YES		NULL	
q_au_1	float	YES		NULL	
q_au_2	float	YES		NULL	
period_yr	float	YES		NULL	
i_deg	float	YES		NULL	
pha	char(3)	YES		NULL	
orbit_class	varchar(30)	YES		NULL	

Первое сравнение — по размеру данных:

```
mysql> SHOW TABLE STATUS\G
***** 1. row *****
Name: asteroids_json
Engine: InnoDB
Version: 10
Row_format: Dynamic
Rows: 202
Avg_row_length: 405
Data_length: 81920
Max_data_length: 0
Index_length: 0
***** 2. row *****
Name: asteroids_sql
Engine: InnoDB
Version: 10
Row_format: Dynamic
Rows: 202
Avg_row_length: 243
Data_length: 49152
Max_data_length: 0
Index_length: 0
```

Наша SQL-версия использует три страницы по 16 Кбайт, а JSON-версия — пять страниц по 16 Кбайт. Это не вызывает большого удивления. Тип данных JSON требует больше места для хранения дополнительных символов для определения JSON (фигурные и квадратные скобки, двоеточия и т. д.), а также пробелов. В этом небольшом примере размер хранилища данных можно увеличить, преобразовав JSON в определенные типы данных.

Вероятно, есть сценарии использования, когда размер данных не так важен.

Какова же задержка выполнения запросов для рассматриваемых подходов?

Для выборки данных из одного столбца синтаксис SQL запроса очень прост:

```
SELECT designation FROM asteroids_sql;
```

Когда мы выполнили этот запрос в первый раз, он не был кэширован буферным пулом InnoDB и был получен через 1,14 мс. При втором выполнении, с кэшированием запроса в памяти, результат был получен через 0,44 мс.

Для JSON-запроса мы можем получить доступ к полю внутри структуры JSON:

```
SELECT json_data->'$.designation' FROM asteroids_json
```

Аналогично наш первый не кэшированный запрос был выполнен за 1,13 мс. Скорость выполнения последующих запросов была около 0,8 мс. Мы ожидаем, что при такой скорости выполнения разница будет разумная — сотни микросекунд в среде виртуальной машины. По нашему мнению, оба запроса выполняются

достаточно быстро, хотя стоит отметить, что JSON-запрос все же примерно в два раза дольше.

А как насчет доступа к определенным строкам? Для поиска одной строки мы воспользуемся преимуществом, получаемым при использовании индексов:

```
ALTER TABLE asteroids_sql ADD INDEX ( designation );
```

Когда выполняем однострочный поиск, SQL-версия выполняется за 0,33 мс, а JSON-версия — за 0,58 мс, что дает преимущество SQL-версии. Это легко объяснить: индекс позволяет InnoDB возвращать одну строку вместо 202 строк.

Однако сравнивать индексированный запрос с полным сканированием таблицы несправедливо. Чтобы уравнивать условия игры, нужно использовать функцию *сгенерированных столбцов* для извлечения столбца *designation*, а затем создать индекс для этого виртуального сгенерированного столбца:

```
ALTER TABLE asteroids_json ADD COLUMN designation VARCHAR(30) GENERATED ALWAYS AS (json_data->"$.designation"), ADD INDEX ( designation );
```

Это дает схему нашей JSON-таблицы:

```
mysql> DESC asteroids_json;
```

Field	Type	Null	Key	Default	Extra
json_data	json	YES		NULL	
designation	varchar(30)	YES	MUL	NULL	VIRTUAL GENERATED

Сейчас схема генерирует виртуальный столбец *designation* из столбца *json_data* и индексирует его. Теперь мы повторно запускаем поиск одной строки, чтобы использовать индексированный столбец вместо оператора пути к столбцу JSON (->). Поскольку данные поля заключены в кавычки в JSON, нужно искать их в кавычках и в SQL:

```
SELECT * FROM asteroids_json WHERE designation='"(2010 GW62)";
```

Этот запрос выполняется за 0,4 мс, что довольно близко к нашей SQL-версии с временем выполнения 0,33 мс.

Из рассмотренного простого тестового примера видно, что объем используемого табличного пространства, по-видимому, является основным фактором, объясняющим, почему вы должны задействовать столбцы SQL, а не хранить необработанный документ JSON. Скорость работы со столбцами SQL по-прежнему выше. В целом решение об использовании собственного SQL или JSON сводится к тому, что перевешивает: производительность или простота хранения JSON в базе данных. Если вы обращаетесь к этим данным миллионы или миллиарды раз в день, разница в скорости будет увеличиваться.

Выбор идентификаторов

В общем, идентификатор — это уникальные поле или комбинация полей, которые служат ссылкой на строку. Например, если у вас есть таблица с данными о пользователях, можно назначить каждому из них числовой идентификатор или уникальное имя пользователя. Это поле может быть или первичным ключом (PRIMARY KEY), или его частью.

Выбор хорошего типа данных для столбца идентификатора очень важен. Вы наверняка будете сравнивать эти столбцы с другими значениями (например, в соединениях) и использовать их для поиска чаще, чем прочие столбцы. Кроме того, возможно, станете задействовать идентификаторы в качестве внешних ключей в других таблицах, поэтому, когда выбираете тип данных для столбца идентификатора, вы, скорее всего, выбираете тип столбцов и в связанных таблицах. (Как мы показали ранее в этой главе, рекомендуется использовать в связанных таблицах одни и те же типы данных, потому что вы, вероятно, будете задействовать их для соединений.)

При выборе типа для столбца идентификатора вам необходимо учитывать 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 преобразует их в строки при выполнении сравнений в строковом контексте.

Выбрав тип, убедитесь, что используете его и во всех связанных таблицах. Типы должны в точности совпадать, включая такие свойства, как UNSIGNED¹. Смешение различных типов данных может вызвать проблемы с производительностью, но, даже если этого не произойдет, неявное преобразование типов во время сравнений может породить труднообнаруживаемые ошибки. К тому же они могут проявиться гораздо позже, когда вы забудете, что сравниваете разные типы данных. Выберите наименьший размер, который может вместить требуемый диапазон значений, и при необходимости оставьте место для увеличения в дальнейшем. Например, если у вас есть столбец `state_id`, в котором хранятся названия штатов США, вам не нужны тысячи или миллионы значений, поэтому не используйте тип `INT`. Вполне достаточно типа `TINYINT`, который на 3 байта короче. Если вы применяете это значение в качестве внешнего ключа в других таблицах, 3 байта могут иметь большое значение. Далее приведем несколько советов.

¹ При использовании подсистемы хранения InnoDB в принципе невозможно создать внешние ключи, если типы данных не совпадают в точности. Появляющееся сообщение об ошибке ERROR 1005 (HY000): Can't create table может сбивать с толку (в определенном контексте), а вопросы на эту тему часто возникают в списках рассылки MySQL. (Как ни странно, вы можете создавать внешние ключи между столбцами VARCHAR разной длины.)

Целочисленные типы

Целочисленные типы обычно являются лучшим выбором для идентификаторов, потому что они работают быстро и допускают `AUTO_INCREMENT`. Атрибут столбца `AUTO_INCREMENT` обеспечивает автоматическое генерирование нового значения целочисленного типа для каждой новой строки. Например, биллинговой системе может потребоваться создать новый счет для каждого клиента. Использование `AUTO_INCREMENT` означает, что первый сгенерированный счет будет 1, второй — 2 и т. д. Имейте в виду: вы должны убедиться, что выбрали правильный размер целочисленного типа, обеспечивающий ожидаемый рост объема данных. Известны несколько случаев простоя системы, связанных с неожиданной нехваткой диапазона целых чисел.

ENUM и SET

Типы `ENUM` и `SET` обычно не подходят для идентификаторов, хотя могут подойти для статических таблиц определений, которые содержат значения состояния или типа. Столбцы `ENUM` и `SET` подходят для хранения такой информации, как статус заказа или тип продукта.

Например, если вы используете поле `ENUM` для определения типа продукта, вам может понадобиться справочная таблица с первичным ключом по идентичному полю `ENUM`. (Можете добавить в справочную таблицу столбцы для описательного текста, создать глоссарий или добавить комментарии в элементах раскрывающегося меню на веб-сайте.) В этом случае вы захотите использовать `ENUM` в качестве идентификатора, но в большинстве случаев этого делать не следует.

Строковые типы

По возможности избегайте строковых типов для идентификаторов, поскольку они занимают много места и обычно обрабатываются медленнее, чем целочисленные типы.

Следует также быть очень внимательными, работая со случайными строками, такими как строки, сгенерированные функциями `MD5()`, `SHA1()` или `UUID()`. Случайные значения, созданные с их помощью, распределяются случайным образом в большом пространстве, что может замедлить работу команды `INSERT` и некоторых типов запросов `SELECT`¹.

- Они замедляют запросы `INSERT`, поскольку вставляемое значение должно быть помещено в случайное место в индексах. Это приводит к разделению

¹ В то же время в больших таблицах, в которые одновременно записывают данные множество клиентов, такие псевдослучайные значения могут помочь избежать «горячих» мест.

страниц, появлению необходимости произвольного доступа к диску и фрагментации кластерного индекса в подсистемах хранения, которые его поддерживают.

- Они замедляют выполнение запросов `SELECT`, поскольку логически смежные строки будут сильно разбросаны по всему диску и в памяти.
- Случайные значения приводят к низкой производительности кэшей для всех типов запросов, поскольку нарушают принцип локальности ссылок, лежащий в основе его работы. Если весь набор данных одинаково «горячий», нет никакого преимущества в том, чтобы кэшировать какую-то конкретную часть данных в памяти, а если рабочие данные не помещаются в памяти, то часто будут возникать сбросы из кэша.

Если вы решили сохранять значения универсального уникального идентификатора (UUID), вам следует удалить тире или, что еще лучше, преобразовать значения UUID в 16-байтовые числа с помощью `UNHEX()` и сохранить их в столбце типа `BINARY(16)`. Можно извлекать значения в шестнадцатеричном формате с помощью функции `HEX()`.

ОСТЕРЕГАЙТЕСЬ АВТОМАТИЧЕСКИ СГЕНЕРИРОВАННЫХ СХЕМ

Мы рассмотрели наиболее важные особенности типов данных (одни больше влияют на производительность, другие — меньше), но еще не рассказали вам о недостатках автоматически сгенерированных схем.

Плохо написанные программы миграции схем и программы, автоматически генерирующие схемы, могут вызвать серьезные проблемы с производительностью. Некоторые программы используют большие поля `VARCHAR` для всех типов данных или разные типы данных для столбцов, которые будут сравниваться при соединениях. Обязательно тщательно изучите схему, если она была сгенерирована автоматически.

Системы объектно-реляционного отображения (object-relational mapping, ORM) (и фреймворки, которые их используют) часто становятся еще одним кошмаром для желающих добиться высокой производительности. Некоторые из этих систем позволяют хранить данные любого типа в бэкенд-хранилище данных любого типа, что обычно означает, что они не могут использовать сильные стороны конкретного хранилища данных. Иногда они записывают каждое свойство каждого объекта в отдельной строке, даже с применением хронологического контроля версий, поэтому существует несколько версий каждого свойства!

Вероятно, такой подход привлекателен для разработчиков, потому что позволяет им применять объектно-ориентированный стиль, не задумываясь о том, как хранятся данные. Однако приложения, которые скрывают сложность от разработчиков, обычно плохо масштабируются. Мы рекомендуем вам тщательно подумать, прежде чем променять производительность на удобство разработки. И всегда тестируйте на действительно большом реалистичном наборе данных, чтобы не обнаруживать проблемы с производительностью слишком поздно.

Специальные типы данных

Некоторые виды данных не соответствуют напрямую доступным встроенным типам. Хорошим примером таких данных является адрес IPv4. Для хранения IP-адресов часто используют столбцы типа `VARCHAR(15)`. Однако на самом деле IP-адрес является 32-битным беззнаковым целым числом, а не строкой. Запись с точками, разделяющими байты, предназначена только для того, чтобы человеку было удобнее его воспринимать. Лучше хранить IP-адреса как целые числа без знака. MySQL предоставляет функции `INET_ATON()` и `INET_NTOA()` для преобразования между двумя представлениями. Используемое пространство сокращается с ~16 байт для `VARCHAR(15)` до 4 байт для 32-битного целого числа без знака. Если вас беспокоит удобочитаемость базы данных и вы не хотите продолжать применять функции для просмотра данных строк, помните, что в MySQL есть представления и с их помощью удобнее просматривать свои данные.

Подводные камни проектирования схемы в MySQL

Есть универсально плохие и хорошие принципы проектирования, существуют также проблемы, возникающие из-за особенностей реализации MySQL, и это означает вероятность появления ошибок, характерных только для MySQL. В этом разделе обсуждаются проблемы, которые мы наблюдали при разработке схем с MySQL. Этот материал может помочь избежать этих ошибок и выбрать альтернативы, которые лучше работают с конкретной реализацией MySQL.

Слишком много столбцов

Во время работы MySQL строки копируются между сервером и подсистемой хранения данных в формате строкового буфера, затем сервер преобразует буфер в столбцы. При этом преобразование из строкового буфера в структуру строковых данных может оказаться весьма затратным. Формат строки InnoDB всегда требует преобразования. Затраты на него зависят от количества столбцов. Мы обнаружили, что эта процедура может быть очень затратной при высоком уровне задействования процессора и огромных таблицах (сотни столбцов), даже если фактически использовались только несколько столбцов. Если вы планируете работать с сотнями столбцов, имейте в виду, что характеристики производительности сервера будут разными.

Слишком много соединений

Так называемый паттерн проектирования «Сущность — атрибут — значение» (Entity — Attribute — Value, EAV) является классическим примером неудачного паттерна проектирования, который особенно плохо работает в MySQL. В MySQL существует ограничение на 61 таблицу для каждого соединения, а базы данных EAV требуют много соединений таблицы самой с собой. Мы неоднократно видели, как базы данных, использующие паттерн EAV, в итоге превышают этот предел. Однако даже при гораздо меньшем количестве соединений, чем 61, затраты на планирование и оптимизацию запроса могут вызывать проблемы MySQL. Эмпирическое правило гласит, что, если вам нужно, чтобы запросы выполнялись очень быстро, с высокой степенью параллелизма, лучше иметь дюжину или меньше таблиц на запрос.

Всемогущий тип ENUM

Остерегайтесь чрезмерного использования типа ENUM. Приведем пример из реальной жизни:

```
CREATE TABLE ... (  
  country enum('','0','1','2',..., '31')
```

Схема была обильно одобрена этим паттерном. Скорее всего, такое проектное решение было бы сомнительным в любой базе данных с перечисляемым типом значений. В действительности здесь должно использоваться целочисленное значение, являющееся внешним ключом таблицы словаря или справочника.

Замаскированный тип ENUM

Тип ENUM позволяет столбцу содержать одно значение из набора определенных значений. SET позволяет столбцу содержать одно или несколько значений из набора определенных значений. Иногда их легко перепутать. Приведем пример:

```
CREATE TABLE ...(  
  is_default set('Y','N') NOT NULL default 'N'
```

Здесь почти наверняка должен быть ENUM, а не SET, если предположить, что значение не может быть и истинным, и ложным одновременно.

NULL изобрели не здесь

Ранее мы рекомендовали не использовать NULL и действительно советуем рассмотреть альтернативные варианты, когда это возможно. Даже если вам нужно сохранить в таблице факт «нет значения», можно обойтись без NULL. Как вариант, можете использовать ноль, специальное значение или пустую строку.

Однако в крайнем случае можно применить и NULL. Не бойтесь NULL, если нужно представить неизвестное значение. В некоторых случаях лучше использовать NULL, чем магическую константу. Выбор одного значения из домена ограниченного типа, например использование `-1` для представления неизвестного целого числа, может сильно усложнить код, вызвать ошибки и, как правило, привести к полной неразберихе. Обработка NULL не всегда проста, но часто лучше альтернативных вариантов.

Вот довольно часто встречающийся пример:

```
CREATE TABLE ... (  
  dt DATETIME NOT NULL DEFAULT ' 0000-00-00 00: 00: 00 '
```

Это фиктивное значение с одними нулями может вызвать множество проблем. (Вы можете настроить `SQL_MODE`, чтобы запретить бессмысленные даты, что особенно хорошо для нового приложения, которое еще не успело наполнить базу данных плохими данными.)

Кстати, MySQL все-таки индексирует значения NULL, в отличие от Oracle, который не включает незначащие значения в индексы.

Теперь, после рассмотрения множества практических советов о типах данных и о том, как их выбирать и чего не делать, перейдем к другой части хорошего итеративного дизайна схемы — управлению схемой.

Управление схемой

Реализация изменений схемы — одна из наиболее распространенных задач, которую приходится выполнять инженеру баз данных. Когда вы переходите к этапу запуска десятков или сотен экземпляров базы данных с различными бизнес-контекстами и развивающимися функциями, нужно быть осторожными, чтобы применение этих изменений схемы не стало узким местом для всей организации, а по-прежнему выполнялось безопасно и без сбоев. В этом разделе будет рассказано, как рассматривать управление изменениями схемы в качестве части платформы хранилища данных, достижения каких основных ценностей должна добиваться эта стратегия, какие инструменты вы можете внедрить для ее реализации и как все это сочетается с вашим более широким жизненным циклом доставки программного обеспечения.

Управление схемой как часть платформы хранения данных

Поговорив с любым техническим руководителем быстро растущей организации, вы поймете, что главными в их списке того, что необходимо оптимизировать, являются скорость разработки и время от начала проектирования функций

до их запуска в производственную эксплуатацию. В этом контексте задача планирования масштабного управления схемами заключается в том, чтобы не допустить превращения управления схемами в ручной процесс, когда один или несколько сотрудников тормозят работу всей команды разработки.

Обеспечьте успех своих партнеров

По мере того как количество команд, использующих экземпляры MySQL, в организации растет, вы должны становиться фактором, способствующим их успеху, а не слагаемым, который им нужно преодолеть, чтобы выполнить свою работу. Это относится и к изменениям схемы, а это означает, что вам нужно создать способ развертывания изменений схемы, который позволяет уйти от принципа «Это делает только команда базы данных».

Интеграция управления схемой с непрерывной интеграцией

Рассмотрев ряд инструментов, позволяющих масштабировать управление схемами, поговорим о том, как интегрировать их с конвейерами CI. Но прямо сейчас мы хотели бы подчеркнуть: если вы исходите из предпосылки, что изменениями схемы будут управлять функциональные группы, а не только команда базы данных, то вам нужно максимально приблизить рабочий процесс к тому, как эти группы развертывают изменения кода. Научные исследования показали, что команды, которые относятся к управлению схемой так же, как к развертыванию кода, замечают улучшение процесса предоставления функций и отмечают повышение скорости работы команд. Мы обсудим инструменты, обеспечивающие эту итерацию, с учетом лучших практик доставки программного обеспечения.

Управление версиями для изменений схемы

Мы все используем систему управления версиями для кода, который развертываем, верно? Тогда почему бы не посмотреть, как это должно выглядеть для схемы базы данных? Один из самых первых шагов к масштабному управлению схемой — убедиться, что у вас есть система контроля версий, поддерживающая и отслеживающая вносимые изменения. Это не только полезно, но во многих случаях нужно вашей дружной команде для соблюдения требований, как вы увидите в главе 13. Рассмотрим инструменты, которые позволяют выполнять итерации по схемам баз данных.



Для получения максимальной отдачи в вашей организации используйте тот же инструментарий непрерывной интеграции, что и для развертывания кода.

Платные опции. За последние несколько лет возможности управления схемами баз данных как корпоративного инструмента значительно возросли, в особенности расширилась поддержка настройки MySQL. Если вы ищете готовое решение, которое поможет вашей организации управлять изменениями схемы, вот несколько вопросов, которые следует рассмотреть.

- *Стоимость.* Модели затрат различаются, поэтому вам следует быть осторожными, если выбранное решение будет взимать плату за каждую цель (схему для управления), так как расходы могут быстро накапливаться.
- *Оперативное управление схемой.* На момент написания этого текста платные решения, такие как Flyway, не имеют ясного варианта запуска изменений вашей схемы неблокирующим образом, хотя у его конкурента Liquibase есть хорошо поддерживаемый плагин для онлайн-изменения схемы Percona. Вы должны знать о компромиссах, которые каждый поставщик выбирает от вашего имени, и о том, что они означают для вашей доступности, особенно если вы планируете использовать эти поставщики для управления изменениями схемы для баз данных большого размера (несколько терабайтов на диске).
- *Готовые интеграции.* Большинство этих инструментов поставляются с предположениями о том, на каких языках написано ваше программное обеспечение и, следовательно, какие хуки нужно обеспечить для интеграции с существующим процессом доставки программного обеспечения. Если ваше предприятие многоязычно или находится в процессе изменения основных языков программного обеспечения, некоторые из этих поставщиков можно исключить. В следующем разделе мы расскажем, что делать, если вам нужно выполнить это самостоятельно при внедрении системы управления версиями схемы.

Использование открытого исходного кода. Если приобретение платного инструмента невозможно или есть веские причины, по которым ни одно из существующих решений не подходит для вашей организации, вы можете достичь тех же результатов, задействуя существующие инструменты с открытым исходным кодом и конвейер CI организации.

Известным решением с открытым исходным кодом для управления изменениями схемы в системе контроля версий в нескольких средах является Skeema. Сама Skeema не реализует изменения схемы за вас в рабочей среде — мы вскоре расскажем, как это сделать, — но это отличный инструмент для отслеживания изменений в репозитории системы управления версиями для каждого кластера базы данных и в нескольких средах. Его реализация CLI обеспечивает большую гибкость при интеграции с выбранным вами решением CI. То, как вы интегрируете Skeema непосредственно со своим решением CI, потребует рассмотрения возможностей последнего. В блоге Sendgrid команды Twilio <https://oreil.ly/8YhBS> объясняется, как они интегрировали Skeema с Buildkite, чтобы обеспечить

автономность для функциональных групп, желающих управлять изменениями в своих базах данных.

Обратите внимание: как бы это решение ни интегрировалось с вашим CI, ему требуется доступ для реализации изменений схемы во всех ваших средах, включая производственную. Это подразумевает сотрудничество с командой безопасности, гарантирующее, что вы создадите правильные элементы управления доступом, чтобы воспользоваться преимуществами автоматизации развертывания схемы с применением непрерывной интеграции.



Если вы уже начали масштабировать инфраструктуру базы данных с помощью Vitess, вам следует знать, что Vitess самостоятельно управляет изменениями схемы. Обязательно изучите соответствующий раздел документации.

За последние несколько лет область управления изменениями схемы значительно расширилась в разных средах как с точки зрения автоматизации, так и с точки зрения соответствия требованиям. Вот несколько заключительных выводов, которые помогут вам сделать свой выбор.

- Внедряя инструменты в рабочий процесс, оставайтесь как можно ближе к существующему программному обеспечению. Желательно, чтобы оно уже было знакомо вашей организации.
- Используйте инструмент, который может интегрировать базовые проверки привязки к изменениям схемы, чтобы обеспечить выполнение некоторых базовых требований. Ваше решение должно автоматически отклонять pull-запрос, если новая таблица не использует правильную кодировку или есть внешние ключи, которые вы не хотите разрешать.
- Если вы работаете в многоязычной и быстрорастущей организации, убедитесь, что случайно не создаете искусственные узкие места, такие как единый репозиторий для всех баз данных и всех изменений схемы. Помните, что цель здесь — скорость работы инженерной команды.

Реализация изменений схемы в производственной среде

Теперь, когда мы рассмотрели варианты отслеживания изменений схемы развертывания в вашей организации и управления ими, обсудим, как реализовать эти изменения в производственной среде, не влияя на время безотказной работы ваших баз данных или служб, которые на них полагаются.

Нативные операторы DDL. Неблокирующие изменения схемы MySQL были представлены в версии 5.6, но там они сопровождались рядом предостережений, из-за которых их фактическое использование ограничивалось очень специфическими типами изменений схемы.

К тому времени, когда версия 8.0 стала общедоступной, поддержка нативного DDL в MySQL значительно расширилась, хотя и не стала универсальной. Изменения в первичном ключе и кодировках, включение шифрования для каждой таблицы, добавление или удаление внешних ключей — все это примеры изменений схемы, которые вы по-прежнему не можете сделать изначально с помощью изменения `INPLACE alter`¹. Настоятельно рекомендуем вам ознакомиться с документацией о том, какие изменения разрешены с использованием алгоритмов `INPLACE` или `INSTANT` в качестве предпочтительного, нативного способа внесения изменений в схему в MySQL без простоя.

Однако, даже если необходимое вам изменение технически поддерживается нативным DDL в версии 8.0 и выше, когда изменяемая таблица очень большая, вы можете столкнуться с откатами, если файл журнала изменений таблицы, который InnoDB хранит внутри, становится слишком большим, что приводит к отмене изменений, потребовавших нескольких часов или дней работы. Еще одна причина, по которой вам может понадобиться использовать внешний инструмент, заключается в том, что вы будете стремиться контролировать скорость, с которой происходит изменение таблицы, с помощью механизма регулирования. Это то, чем можно управлять с помощью внешних инструментов, которые мы собираемся обсудить.

Использование внешних инструментов для реализации изменения схемы.

Если вы еще не можете запустить последнюю наилучшую версию MySQL, обеспечивающую максимальную гибкость изменений схемы, то все равно можете комбинировать инструменты непрерывной интеграции с доступными инструментами с открытым исходным кодом, чтобы автоматически запускать изменения схемы в рабочей среде, не влияя на свой сервис. Два основных варианта достижения этой цели — `pt-online-schema-change` от Percona и `gh-ost` от GitHub. В документации к обоим инструментам содержится вся информация, необходимая для того, чтобы научиться устанавливать и использовать их. Поэтому здесь мы сосредоточимся на том, как выбрать подходящий инструмент, какие основные компромиссы следует учитывать и как повысить безопасность применения любого инструмента в рамках автоматизированного конвейера развертывания схемы в рабочей среде.



Следует отметить одну вещь: любой внешний инструмент, реализующий изменения вашей схемы, должен будет сделать полные копии изменяемой таблицы. Инструмент просто уменьшает число коллизий при выполнении этого процесса и не требует разрушительных блокировок записи, но только нативный DDL в MySQL может изменять схемы таблиц без создания полных копий последних.

¹ Подробнее об этом см. в документации по MySQL.

Основным преимуществом `pt-online-schema-change` является его стабильность и то, как долго он используется в сообществе MySQL. В первую очередь он задействует триггеры, позволяющие изменять схемы таблиц любого размера, очень незначительно влияя на доступность базы данных при переходе на новую версию таблицы. Но его основной дизайн также имеет компромиссы. Имейте это в виду, когда учитесь использовать `pt-online-schema-change` для управления конвейером развертывания схемы.

- *Триггеры имеют ограничения.* До MySQL 8.0 у вас не могло быть более одного триггера с одним и тем же действием в одной и той же таблице. Что это значит? Если у вас есть таблица с именем `sales` и вам нужно поддерживать в ней триггер для события вставки, MySQL до версии 8.0 не разрешает другой триггер для события вставки в эту таблицу. Если вы попытаетесь запустить для него изменение схемы `pt-onlineschema-change`, инструмент выдаст ошибку при попытке добавить триггеры, необходимые для его работы. Хотя мы, как правило, настоятельно не рекомендуем использовать триггеры таблиц как часть бизнес-логики, все же возникают случаи, когда устаревшие варианты создают ограничение, и это становится частью выбора компромисса при реализации механизма изменения схемы.
- *Триггеры влияют на производительность.* Есть несколько отличных тестов от Percona, показывающих влияние на производительность даже наличия триггеров, определенных для таблицы. Это снижение производительности может быть незаметным для большинства установок, но если вы используете экземпляры базы данных с очень высокой пропускной способностью транзакций в секунду, вам может потребоваться более тщательно наблюдать за влиянием триггеров, введенных `pt-online-schema-change`, и настроить ее на более консервативное прерывание.
- *Выполнение параллельных миграций.* Из-за использования триггеров и ограничений триггеров в MySQL до версии 8.0 вы обнаружите, что не можете выполнить несколько изменений схемы в одной и той же таблице с помощью `pt-online-schema-change`. Поначалу это может быть незначительным неудобством, но если вы интегрируете инструмент в полностью автоматизированный конвейер миграции схемы, он может стать узким местом для ваших команд.
- *Ограничения, связанные с внешними ключами.* Хотя инструмент имеет некоторый уровень поддержки изменений схемы с использованием внешних ключей, вам необходимо внимательно прочитать документацию и определить, какой компромисс оказывает наименьшее влияние на ваши данные и пропускную способность транзакций.

`gh-ost` был создан командой инженеров данных в GitHub специально как решение для управления процессом изменения схемы без влияния на сервис, но также без использования триггеров. Вместо того чтобы применять триггеры для отслеживания изменений на этапе копирования таблицы, он подключается как реплика к одной из реплик вашего кластера и задействует журналы репликации на основе строк в качестве журнала изменений.

Прежде чем применять `gh-ost` для изменения схемы, вам нужно тщательно обдумать одну вещь: использует существующая база данных внешние ключи или нет. Хотя `pt-online-schemachange` делает серьезную попытку поддерживать изменения схемы для таблиц, которые являются родительскими или дочерними в отношении внешнего ключа, это сложный выбор, полный компромиссов. Жертвуем ли мы временем безотказной работы ради согласованности? Или рискуем некоторым окном возможной несогласованности? В то же время `gh-ost` в основном делает этот выбор за вас, если в таблице, которую вы хотите изменить, существуют внешние ключи. Как объясняет основной разработчик `gh-ost` Шломи Ноах в длинном, но очень полезном сообщении в блоге (<https://oreil.ly/6A10o>), использование внешних ключей и онлайн-инструментов изменения схемы, которые в конечном итоге все еще являются внешними по отношению к движку базы данных, создает среду, в которой трудно найти компромиссы. Он предлагает вообще не задействовать внешние ключи, если вам требуются также изменения онлайн-схемы.

Если вы и ваша команда новички в этой области и начинаете путь к внесению изменений в схемы в своей организации, то, мы считаем, `gh-ost` — лучшее решение для вас, если вы также дисциплинированно относитесь к тому, чтобы не вводить внешние ключи. Учитывая использование двоичных журналов вместо триггеров для отслеживания изменений, мы считаем его более безопасным вариантом: когда не нужно беспокоиться о снижении производительности триггеров, он гораздо более независим от того, какую версию MySQL вы запускаете (с некоторыми оговорками может даже работать с репликацией на основе операторов), и это уже было доказано при крупномасштабных развертываниях.

Когда `pt-online-schema-change` является предпочтительным вариантом? Если вы работаете со старыми базами данных, в которых уже существуют внешние ключи и их удаление представляет собой трудную задачу, то обнаружите, что `pt-online-schema-change` пытается обеспечить более широкую поддержку внешних ключей, но вам придется самостоятельно выбирать наиболее безопасный вариант для обеспечения целостности данных и времени безотказной работы. Кроме того, `gh-ost` использует в своей работе двоичные журналы, и если они по какой-то причине недоступны для инструмента, `pt-online-schema-change` остается жизнеспособным вариантом.

В идеале когда-нибудь мы все сможем вносить изменения в онлайн-схему непосредственно в MySQL, но этот день еще не наступил. Экосистема с открытым исходным кодом прошла долгий путь к тому, чтобы сделать изменение схемы более легко автоматизируемым процессом. Поговорим о том, как объединить все инструменты, чтобы создать полноценный конвейер CI/CD для изменения схемы.

Конвейер CI/CD для изменения схемы

Теперь, когда мы рассмотрели ряд инструментов, от тех, которые помогают управлять версиями определения схемы, до предназначенных для внесения изменений в рабочую среду с минимальным временем простоя, вы можете заметить, что у нас есть все элементы для полной непрерывной интеграции и развертывания изменений схемы, которые могут устранить гигантское узкое место, препятствующее повышению производительности работы инженеров в вашей организации. Соберем их вместе.

- *Организация системы управления версиями схемы.* Начать следует с выделения определений схемы из каждого из кластеров базы данных в отдельные репозитории. Если цель здесь состоит в том, чтобы обеспечить для разных групп гибкость при выполнении их изменений с разной скоростью, то нет смысла объединять все определения схем всех баз данных в одном репозитории. Это разделение позволяет каждой команде определять различные варианты проверок при статическом анализе кода в репозитории. Некоторым командам может потребоваться очень специфический набор символов и сортировки, а другим, вероятно, подойдут настройки по умолчанию. Гибкость работы партнерских команд здесь является ключевым фактором.

Обязательно задокументируйте рабочий процесс того, как инженер функциональной группы может перейти от изменения схемы на своем ноутбуке к той схеме, которая запускается во всех средах и выполняет тесты перед запуском в производственной среде. Модель pull-запроса здесь может оказаться очень полезной, чтобы помочь каждой команде определить, какие тесты следует запускать в автоматическом режиме, когда запрашивается изменение схемы, прежде чем продвигать и запускать изменение в других средах или производственной среде.

- *Базовая конфигурация для обеспечения безопасности.* Определите базовую конфигурацию для выбранного вами онлайн-инструмента изменения схемы. Вы команда, предоставляющая инструменты для партнерских команд, которые полагаются на вас в предоставлении гибких, масштабируемых, но

в то же время безопасных решений. Пока вы будете обдумывать, как станете реализовывать онлайн-инструмент изменения схемы, возможно, придет время определиться с соображениями проектирования схемы, которые должны быть частью общего процесса тестирования запросов на внесение изменений в схему. Например, если вы решите, что предпочитаете безопасность `gh-ost` и его дизайн без триггеров, это будет означать, что вас должна устроить платформа базы данных, свободная от внешних ключей. Не вдаваясь в компромиссы такого выбора, если вы в конечном итоге примете решение «смерть внешним ключам», то должны убедиться, что это закодировано в том, как вы тестируете изменения схемы в хуках перед коммитами или в своем репозитории `Skeema`, чтобы избежать случайного внесения нежелательных изменений схемы слишком далеко в иерархию среды. Аналогично следует выбрать базовую конфигурацию для своего онлайн-инструмента изменения схемы, который обеспечивает базовую систему безопасности для выполнения изменений в рабочей среде. Примеры того, что вы, возможно, захотите ввести в такую конфигурацию, включают максимальное количество запущенных потоков MySQL или максимально допустимую нагрузку на систему. Шаблоны репозитория могут быть мощным инструментом, позволяющим легко сделать правильный выбор, когда какая-либо специализированная команда разработчиков создает новую базу данных и хочет, чтобы репозиторий отслеживал изменения схемы и управлял ими.

- *Гибкость конвейера для каждой команды.* Организуя определения схемы в репозитории для каждой базы данных, вы обеспечиваете для всех команд, владеющих этой базой данных, максимально гибкое принятие решения о том, насколько автоматизированным или управляемым человеком должен быть ее конвейер. Одна команда может все еще находиться на стадии итерации разработки нового продукта, и они согласны использовать pull-запросы для извлечения схемы, автоматически выполняемые до тех пор, пока проходят определенные тесты. Другая команда может владеть критически важной базой данных и требовать более осторожного подхода, предпочитая, чтобы оператор утвердил запрос на извлечение, прежде чем система CI сможет передать его в следующую среду.

Разрабатывая способы масштабируемого развертывания с изменением схемы в своей организации, следите за конечной целью — обеспечением скорости развертывания в сочетании с безопасностью при развитии вашей инженерной организации. При этом команда инженеров базы данных не должна быть узким местом в том, как компания переходит от идей к функциям в производственной системе.

Резюме

Хороший дизайн схемы довольно универсален, однако в MySQL есть особые детали реализации, которые необходимо учитывать. В двух словах, основная идея заключается в том, чтобы все было как можно меньше и проще. MySQL любит простоту, как и люди, которым приходится работать с вашей базой данных. Помните о следующих рекомендациях.

- Старайтесь избегать в дизайне крайностей, таких как схемы, которые будут вызывать чрезвычайно сложные запросы или таблицы с огромным количеством столбцов.
- Задействуйте небольшие, простые, подходящие типы данных и избегайте NULL, если только это не единственно правильный способ моделирования реальных данных.
- Старайтесь применять одни и те же типы данных для хранения похожих или связанных значений, особенно если они будут использоваться в соединении.
- Остерегайтесь строк переменной длины, которые могут привести к пессимистичному выделению максимальной длины строк в памяти для временных таблиц и сортировки.
- Старайтесь использовать целые числа для идентификаторов, если это возможно.
- Избегайте устаревших MySQL-измов, таких как указание точности для чисел с плавающей запятой или ширины отображения для целых чисел.
- Будьте осторожны с ENUM и SET. Они удобны, но ими реально злоупотребить, и иногда они сложны. ВПР лучше избегать.

Дизайн базы данных — это целая наука. Если вы серьезно озабочены дизайном базы данных, рекомендуем изучить материалы, посвященные этой теме¹.

Помните, что схема будет развиваться вместе с вашими бизнес-потребностями и тем, что вы узнаете от пользователей. Это означает, что наличие надежного программного обеспечения, управляющего жизненным циклом изменения схемы, — важная часть обеспечения безопасности и масштабируемости эволюции в вашей организации.

¹ Для углубленного чтения рекомендуем книгу: *Hernandez M.J. Database Design for Mere Mortals.* — Pearson.

Повышение производительности с помощью индексирования

Индексы в MySQL, также называемые *ключами*, представляют собой структуры данных, которые подсистемы хранения используют для быстрого поиска строк. У них есть еще несколько полезных свойств, которые мы рассмотрим в этой главе.

Индексы имеют решающее значение для хорошей производительности и становятся все более важными по мере увеличения объема данных. Небольшие базы данных с незначительной нагрузкой зачастую могут хорошо работать даже без правильно построенных индексов, но по мере роста объема хранимой в базе информации производительность может очень быстро упасть¹. К сожалению, на практике об индексах часто забывают или неправильно понимают их смысл, поэтому плохое индексирование становится основной причиной реальных проблем с производительностью. Вот почему в книге мы разместили этот материал даже раньше, чем обсуждение оптимизации запросов.

Оптимизация индекса — это, пожалуй, самый мощный способ повысить производительность запросов. Индексы могут увеличить производительность на много порядков, а оптимальные индексы иногда способны повысить ее примерно на два порядка больше, чем просто хорошие. Для создания действительно оптимальных индексов часто требуется переписать запросы, поэтому текущая и следующая главы тесно связаны между собой.

¹ Твердотельные накопители имеют различные характеристики производительности, которые мы рассмотрели в главе 4. Принципы индексации остаются верными, но потери, которых мы пытаемся избежать, для твердотельных накопителей не так велики, как для обычных дисков.

В этой главе используются примеры баз данных, такие как Sakila Sample Database, доступные на веб-сайте MySQL (<https://oreil.ly/cIabb>). Sakila — это пример базы данных, которая моделирует пункт проката с коллекцией актеров, фильмов, клиентов и т. д.

Основы индексирования

Самый простой способ понять, как работает индекс в MySQL, — представить себе алфавитный указатель в книге. Чтобы определить, где в книге обсуждается конкретная тема, вы в алфавитном указателе находите термин и номер страницы, на которой он упоминается.

В MySQL подсистема хранения задействует индексы аналогичным образом. Она ищет значение в структуре данных индекса. Обнаружив совпадение, может перейти к самой строке, содержащей совпадение. Предположим, вы выполняете следующий запрос:

```
SELECT first_name FROM sakila.actor WHERE actor_id = 5;
```

Индекс построен по столбцу `actor_id`, поэтому MySQL будет использовать его для поиска строк, у которых значение поля `actor_id` равняется 5. Другими словами, система выполняет поиск значений в индексе и возвращает все строки, содержащие указанное значение.

В индекс включены данные из одного или нескольких столбцов таблицы. Если он построен по нескольким столбцам, то их порядок очень важен, потому что MySQL может эффективно выполнять поиск только по крайнему левому префиксу индекса. Как вы увидите в дальнейшем, создание индекса по двум столбцам — это совсем не то же самое, что создание двух отдельных индексов, каждый по одному столбцу.

ДОЛЖЕН ЛИ Я ЭТО ЧИТАТЬ, ЕСЛИ У МЕНЯ ORM?

Краткий ответ: да, вам все равно нужно узнать об индексации, даже если вы полагаетесь на объектно-реляционное отображение (object-relational mapping, ORM).

Инструменты ORM выполняют логически и синтаксически правильные запросы (в большинстве случаев), но они редко создают запросы, удобные для индекса, за исключением случаев, когда вы используете их только для самых основных типов запросов, таких как поиск по первичному ключу. Не стоит ожидать, что ваш инструмент ORM, каким бы сложным он ни был, справится с тонкостями и сложностями индексирования. Прочтите оставшуюся часть этой главы, если вы не согласны! Иногда даже опытному человеку сложно разобраться во всех возможностях индексирования, не говоря уже об ORM.

Типы индексов

Существует множество типов индексов, каждый из которых предназначен для достижения той или иной цели. Индексы реализуются на уровне подсистемы хранения, а не на уровне сервера. Таким образом, они не стандартизированы: индексирование работает по-разному в зависимости от подсистемы и далеко не все подсистемы поддерживают все типы индексов. Даже когда несколько подсистем хранения поддерживают определенный тип индекса, его внутренняя реализация может существенно различаться.

Учитывая, что в этой книге предполагается, что вы используете InnoDB в качестве подсистемы хранения для всех ваших таблиц, мы будем рассматривать конкретно реализацию индексов в InnoDB.

Тем не менее поговорим о двух наиболее часто используемых типах индексов, которые в настоящее время поддерживает MySQL, а также об их достоинствах и недостатках.

Индексы, упорядоченные на основе В-дерева

Когда говорят об индексе без упоминания типа, обычно имеют в виду *индексы, упорядоченные на основе В-дерева*, в которых для хранения данных задействуется структура данных, называемая В-деревом¹. Большинство подсистем хранения MySQL поддерживают этот тип индекса.

Мы используем термин «В-дерево» для этих индексов, потому что именно так MySQL называет их в `CREATE TABLE` и других командах. Однако на внутреннем уровне подсистемы хранения могут задействовать совершенно иные структуры данных. Например, в подсистеме хранения NDB Cluster для этих индексов применяется структура данных Т-дерево, хотя они по-прежнему называются `BTREE`, а в InnoDB — деревья типа В+. Однако рассмотрение вариантов структур и алгоритмов выходит за рамки этой книги.

Общая идея В-дерева заключается в том, что все значения хранятся по порядку и все листья-страницы находятся на одинаковом расстоянии от корня. На рис. 7.1 показано общее представление индекса, упорядоченного на основе В-дерева, которое приблизительно соответствует тому, как работают индексы InnoDB.

¹ Во многих подсистемах хранения на самом деле используются индексы типа «В+-дерево», в которых каждый лист содержит указатель на следующий лист для ускорения обхода дерева по диапазону значений. Более подробное описание индексов со структурой В-дерева можно найти в литературе по информатике.



Рис. 7.1. Индекс, упорядоченный на основе структуры В-дерева (технически дерева B+)

Индекс, упорядоченный на основе структуры В-дерева, ускоряет доступ к данным, поскольку подсистеме хранения не нужно сканировать всю таблицу, чтобы найти искомое. Вместо этого он начинается с корневого узла (на этом рисунке не показан). В корневом узле имеется массив указателей на дочерние узлы, и подсистема хранения следует этим указателям. Для нахождения подходящего указателя она просматривает значения в узловых страницах, которые определяют верхнюю и нижнюю границы значений в дочерних узлах. В конце концов подсистема хранения либо определяет, что требуемого значения не существует, либо благополучно достигает конечной страницы.

Листья-страницы представляют собой особый случай, поскольку в них находятся указатели на проиндексированные данные, а не указатели на другие страницы. (Разные подсистемы хранения имеют разные типы указателей на данные.) На нашем рисунке показаны только одна узловая страница и соответствующие ей листья-страницы, но между корнем и листьями может быть много уровней узловых страниц. Глубина дерева зависит от размера таблицы.

Поскольку В-деревья хранят индексированные столбцы по порядку, они полезны для поиска диапазонов данных. Например, спуск по дереву для индекса текстового поля проходит через значения в алфавитном порядке, поэтому поиск всех, чье имя начинается с букв от I до K, эффективен.

Предположим, у вас есть следующая таблица:

```
CREATE TABLE People (
  last_name varchar(50) not null,
  first_name varchar(50) not null,
  dob date not null,
  key(last_name, first_name, dob)
);
```

Индекс будет содержать значения из столбцов `last_name`, `first_name` и `dob` для каждой строки таблицы. На рис. 7.2 показано, как индекс упорядочивает хранящиеся в нем данные.

Обратите внимание на то, что в индексе значения отсортированы в том же порядке, в котором столбцы указаны в команде `CREATE TABLE`. Посмотрите на две последние записи: есть два человека с одинаковыми фамилией и именем, но разными датами рождения, и они отсортированы именно по дате рождения.

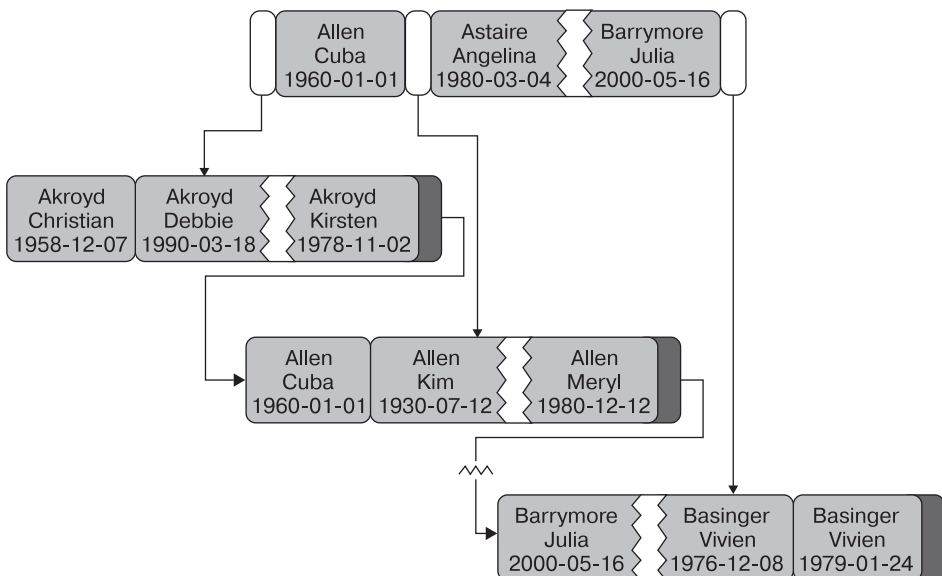


Рис. 7.2. Примеры записей из индекса, упорядоченного на основе структуры В-дерева (технически дерева В+)

Адаптивный хеш-индекс. Подсистема хранения InnoDB имеет специальную функцию, называемую адаптивными хеш-индексами. Когда InnoDB замечает, что к некоторым значениям индекса обращаются очень часто, она строит для них хеш-индекс в памяти поверх индексов, упорядоченных на основе структуры В-дерева. Это придает его индексам, упорядоченным на основе структуры В-дерева, некоторые свойства хеш-индексов, такие как очень быстрый хешированный поиск. Этот процесс полностью автоматический, и вы не можете его контролировать или настраивать, хотя можете вообще отключить адаптивный хеш-индекс.

Типы запросов, которые могут использовать индекс сбалансированного дерева. Индексы В-дерева хорошо подходят для поиска по полному значению ключа, диапазону ключей или префиксу ключа. Они полезны, только если поиск использует крайний левый префикс индекса¹. Индекс, который мы показали в предыдущем разделе, будет полезен для следующих типов запросов.

- *Поиск по полному значению.* При поиске по полному значению ключа задаются критерии для всех столбцов, по которым построен индекс. Например, этот индекс может помочь вам найти человека по имени Куба Аллен, родившегося 1 января 1960 года.
- *Поиск по полному совпадению левого префикса.* Этот индекс может помочь найти всех людей с фамилией Аллен. Используется только первый столбец в индексе.
- *Поиск по префиксу столбца.* Можно искать соответствие по началу значения столбца. Рассматриваемый индекс позволит найти всех людей, чьи фамилии начинаются с буквы J. При этом используется только первый столбец индекса.
- *Поиск по диапазону значений.* Этот индекс может помочь вам найти людей, чьи фамилии находятся между Аллен и Бэрримор. Используется также только первый столбец.
- *Точное соответствие одной части и соответствие диапазону другой части.* Этот индекс может помочь вам найти всех, чья фамилия Аллен, а имя начинается на букву К (Ким, Карл и т. д.). В данном случае выполняется поиск по полному значению в столбце `last_name` и поиск по диапазону значений в столбце `first_name`.

¹ Это присуще только MySQL и даже зависит от версии. Некоторые другие базы данных могут использовать неначальные части индекса, хотя обычно более эффективен полный префикс. MySQL может предложить эту опцию в будущем. Мы покажем обходные пути позже в этой главе.

- *Запросы только по индексу.* Индексы, упорядоченные на основе В-дерева, обычно могут поддерживать запросы только по индексу, то есть обращение именно к индексу, а не к самой строке. Мы обсуждаем эту оптимизацию в подразделе «Покрывающие индексы» далее в этой главе.

Поскольку узлы дерева отсортированы, их можно использовать как для поиска значений, так и для запросов с фразой `ORDER BY` (поиск значений в отсортированном порядке). В общем, если В-дерево может помочь вам найти строку по определенному критерию, его можно применять и для сортировки строк по тому же критерию. Таким образом, наш индекс будет полезен для запросов с фразой `ORDER BY`, в которой сортировка соответствует видам поиска, которые мы только что перечислили.

Для применения индексов, упорядоченных на основе В-дерева, есть ряд ограничений.

- Они бесполезны, если в критерии поиска не указан крайний левый из индексированных столбцов. Например, рассматриваемый индекс не поможет вам найти всех людей по имени Билл или всех родившихся в определенный день, потому что эти столбцы не самые левые в индексе. Точно так же вы не можете использовать индекс, чтобы найти людей, чья фамилия заканчивается на определенную букву.
- Вы не можете пропускать столбцы в индексе, то есть не сможете найти всех людей с фамилией Смит, родившихся в конкретный день. Если вы не укажете значение для столбца `first_name`, MySQL сможет использовать только первый столбец индекса.
- Подсистема хранения не может оптимизировать поиск по столбцам, находящимся правее первого столбца. Например, для запроса с условием `WHERE last_name="Smith" AND first_name LIKE 'J%' AND dob='1976-12-23'` будут задействованы только первые два столбца индекса, поскольку `LIKE` задает диапазон условий (однако сервер может использовать остальные столбцы для других целей). Для столбца, имеющего ограниченный набор значений, эту проблему можно обойти, указав условия равенства вместо условия, задающего диапазон.

Теперь вы знаете, почему порядок столбцов чрезвычайно важен: все эти ограничения так или иначе связаны именно с ним. Для достижения оптимальной производительности вам может потребоваться создать индексы с одними и теми же столбцами в разном порядке, чтобы удовлетворить свои запросы.

Некоторые из названных ограничений не присущи индексам, упорядоченным на основе В-дерева, а являются следствием того, как оптимизатор запросов MySQL и подсистемы хранения используют индексы. Возможно, некоторые ограничения в дальнейшем будут устранены.

Полнотекстовые индексы

Полнотекстовый (FULLTEXT) индекс представляет собой специальный тип индекса, который находит ключевые слова в тексте, а не сравнивает значения непосредственно со значениями в индексе. Полнотекстовый поиск кардинально отличается от других типов поиска. Он позволяет использовать стоп-слова, морфологический поиск, учет множественного числа, а также логический поиск. Он гораздо больше похож на работу поисковых систем, чем на простое сравнение с критерием в разделе `WHERE`.

Наличие полнотекстового индекса по столбцу не делает индекс, упорядоченный на основе В-дерева по этому же столбцу, менее полезным. Полнотекстовые индексы предназначены для операций `MATCH AGAINST`, а не для обычных команд с фразой `WHERE`.

Преимущества индексов

Индексы позволяют серверу быстро перейти к нужной позиции в таблице, но этим их достоинства не исчерпываются. Как вы, вероятно, уже поняли, у индексов есть несколько дополнительных преимуществ, основанных на свойствах структур данных, используемых для их создания.

Индексы, упорядоченные на основе В-дерева, — наиболее широко распространенный тип, они функционируют, сохраняя данные в отсортированном порядке, и MySQL может применять этот прием для запросов с такими фразами, как `ORDER BY` и `GROUP BY`. Поскольку данные предварительно отсортированы, индекс, упорядоченный на основе В-дерева, также хранит связанные значения близко друг к другу. Наконец, индекс фактически хранит копию значений, поэтому некоторые запросы могут быть удовлетворены только с помощью индекса. Из этих свойств вытекают три основных преимущества.

- Индексы уменьшают объем данных, которые сервер должен просмотреть для нахождения искомого значения.
- Индексы помогают серверу избежать сортировки и временных таблиц.
- Индексы превращают произвольный ввод/вывод в последовательный.

Рассмотрению индексов можно посвятить целую книгу. Для тех, кто хотел бы изучить этот вопрос глубже, подробнее, можем порекомендовать книгу Тапио Лахденмаки и Майка Лича *Relational Database Index Design and the Optimizers* (издательство Wiley). Из нее вы, помимо прочего, узнаете, как рассчитать затраты и выгоды от применения индексов, как оценить скорость запросов и как определить, будет ли обслуживание индексов дороже, чем преимущества, которые они предоставляют.

Кроме того, в этой книге представлена трехзвездочная система оценки того, насколько индекс подходит для запроса. Индекс получает одну звезду, если он размещает соответствующие строки рядом друг с другом, вторую звезду, если его строки отсортированы в порядке, необходимом для запроса, и последнюю звезду, если содержит все столбцы, необходимые для запроса. Мы вернемся к этим принципам далее в текущей главе.

Стратегии индексирования для достижения производительности

Создание правильных индексов и их правильное применение необходимы для достижения хорошей производительности запросов. Мы рассказали о различных типах индексов и изучили их сильные и слабые стороны. Теперь посмотрим, как на практике максимально продуктивно задействовать силу индексов.

Существует множество способов эффективного выбора и использования индексов, поскольку предусмотрено множество оптимизаций в особых случаях и вариантов специализированного поведения¹. Определение того, что и когда применять, и оценка влияния вашего выбора на производительность — это навыки, которые вы получите со временем. Следующие разделы помогут понять, как эффективно использовать индексы.

Префиксные индексы и селективность индексов

Часто можно сэкономить пространство и добиться хорошей производительности, проиндексировав первые несколько символов вместо всего значения. Это позволяет вашим индексам занимать меньше места, однако делает их менее *селективными (избирательными)*. Селективность индекса — это отношение количества различных индексированных значений (*кардинальности*) к общему количеству строк в таблице ($\#T$). Диапазон возможных значений селективности варьируется в диапазоне от $1/\#T$ до 1. Высокоселективный индекс хорош тем, что он позволяет MySQL отфильтровывать больше строк при поиске совпадений. Уникальный индекс имеет селективность 1, лучше не бывает.

Префикс столбца часто является достаточно избирательным, чтобы обеспечить хорошую производительность. Если вы индексируете столбцы типа BLOB либо TEXT

¹ Оптимизатор MySQL — это очень загадочное и мощное устройство. Рекомендуем вам в своих запросах и при расчете рабочей нагрузки для определения наиболее оптимальных стратегий полагаться на команду EXPLAIN.

или очень длинные столбцы типа `VARCHAR`, вы должны определить префиксные индексы, поскольку MySQL не позволяет индексировать их по полной длине.

Основная проблема заключается в выборе длины префикса, которая должна быть достаточно велика, чтобы обеспечить хорошую селективность, но не слишком велика, чтобы сэкономить место. Префикс должен быть достаточно длинным, чтобы польза от его применения была почти такой же, как от использования индекса по полному столбцу. Другими словами, кардинальность префикса должна быть почти такой же, как кардинальность всего столбца.

Для определения оптимальной длины префикса найдите наиболее часто встречающиеся значения и сравните их перечень со списком наиболее часто встречающихся префиксов. В тестовой базе данных Sakila Sample Database нет подходящей таблицы для демонстрации этого, поэтому создадим таблицу на основе таблицы `city`, чтобы у нас было достаточно данных для работы:

```
CREATE TABLE sakila.city_demo(city VARCHAR(50) NOT NULL);
INSERT INTO sakila.city_demo(city) SELECT city FROM sakila.city;
-- Repeat the next statement five times:
INSERT INTO sakila.city_demo(city) SELECT city FROM sakila.city_demo;
-- Now randomize the distribution (inefficiently but conveniently):
UPDATE sakila.city_demo
  SET city = (SELECT city FROM sakila.city ORDER BY RAND() LIMIT 1);
```

Теперь у нас есть тестовый набор данных. Данные распределяются не очень реалистично. Кроме того, поскольку мы использовали `RAND()`, ваши результаты будут отличаться от полученных нами, но для примера это не принципиально. Сначала найдем наиболее часто встречающиеся города:

```
mysql> SELECT COUNT(*) AS c, city
        -> FROM sakila.city_demo
        -> GROUP BY city ORDER BY c DESC LIMIT 10;
```

c	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 примерно от 45 до 65 раз. Теперь мы находим наиболее часто встречающиеся префиксы названий городов. Начнем с трехбуквенных:

```
mysql> SELECT COUNT(*) AS c, LEFT(city, 3) AS pref
-> FROM sakila.city_demo GROUP BY pref ORDER BY c DESC LIMIT 10;
```

c	pref
483	San
195	Cha
177	Tan
167	Sou
163	al-
163	Sal
146	Shi
136	Hal
130	Val
129	Bat

Каждый префикс встречается значительно чаще, поэтому уникальных префиксов гораздо меньше, чем уникальных полных названий городов. Идея состоит в том, чтобы увеличивать длину префикса до тех пор, пока он не станет почти таким же селективным, как полная длина столбца. По результатам экспериментов мы определили, что семи символов вполне достаточно:

```
mysql> SELECT COUNT(*) AS c, LEFT(city, 7) AS pref
-> FROM sakila.city_demo GROUP BY pref ORDER BY c DESC LIMIT 10;
```

c	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 хорошую длину префикса состоит в вычислении селективности полного столбца и попытках подобрать длину префикса, обеспечивающую близкую селективность. Селективность полного столбца можно найти следующим образом:

```
mysql> SELECT COUNT(DISTINCT city)/COUNT(*) FROM sakila.city_demo;
```

COUNT(DISTINCT city)/COUNT(*)
0.0312

В среднем префикс будет примерно таким же хорошим (с небольшой оговоркой), если его селективность около 0,031. В одном запросе можно посчитать селективность нескольких разных длин префиксов, что особенно полезно для очень больших таблиц. Вот как можно найти селективность нескольких длин префиксов 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 увеличение длины дает небольшое улучшение селективности по мере приближения к семи символам.

Недостаточно обращать внимание только на среднюю селективность. Следует подумать (в этом и состояла оговорка) также о селективности в худшем случае. На основе средней селективности вы можете прийти к выводу, что префикс из четырех или пяти символов достаточно хорош, но, если ваши данные распределены очень неравномерно, это может завести вас в ловушку. Если вы посмотрите на количество вхождений наиболее распространенных префиксов названий городов, используя значение 4, то ясно увидите неравномерность:

```
mysql> SELECT COUNT(*) AS c, LEFT(city, 4) AS pref
-> FROM sakila.city_demo GROUP BY pref ORDER BY c DESC LIMIT 5;
```

c	pref
205	San
200	Sant
135	Sout
104	Chan
91	Toul

При длине четыре символа наиболее распространенные префиксы встречаются немного чаще, чем наиболее часто встречающиеся полноразмерные значения. То есть селективность по этим значениям ниже средней селективности. Если у вас есть более реалистичный набор данных, чем этот сгенерированный случайным образом образец, то, вероятно, данный эффект может оказаться значительно более выраженным. Например, построение индекса префикса из четырех символов для реальных названий городов даст высокую селективность для начинающих с San и New, которых много.

Теперь, определив подходящую длину префикса для нашего примера данных, создадим индекс префикса для столбца:

```
ALTER TABLE sakila.city_demo ADD KEY (city(7));
```

Префиксные индексы могут быть отличным способом уменьшения размера и повышения быстродействия индекса, но у них есть и недостатки: MySQL не может использовать префиксные индексы для запросов с фразами `ORDER BY` или `GROUP BY`, а также задействовать их в качестве покрывающих индексов.

Еще один распространенный способ получения выигрыша от префиксных индексов — использование длинных шестнадцатеричных идентификаторов. Мы обсуждали более эффективные методы хранения таких идентификаторов в предыдущей главе, но что, если вы примените пакетное решение, которое не можете модифицировать? Нам неоднократно встречалось использование длинных шестнадцатеричных строк с `vBulletin` и другими приложениями, которые применяют MySQL для хранения сеансов веб-сайтов. Добавление индекса к первым восьми символам или около того часто значительно повышает производительность таким образом, что это полностью прозрачно для приложения.

Многостолбцовые индексы

Люди зачастую плохо разбираются в многостолбцовых индексах. Распространенными ошибками являются индексирование многих или всех столбцов по отдельности или индексирование столбцов в неправильном порядке.

Мы обсудим порядок столбцов в следующем разделе. У первой ошибки — индексирования множества столбцов по отдельности — есть характерная подпись в команде `SHOW CREATE TABLE`:

```
CREATE TABLE t (  
  c1 INT,  
  c2 INT,  
  c3 INT,  
  KEY(c1),  
  KEY(c2),  
  KEY(c3)  
);
```

Такая стратегия индексации часто оказывается результатом того, что люди дают расплывчатые, но авторитетно звучащие рекомендации, например «Создавайте индексы для столбцов, которые появляются в предложении `WHERE`». Этот совет ошибочен. В лучшем случае вы получите индекс с одной звездой. Такие индексы могут быть на много порядков медленнее, чем действительно оптимальные индексы. Иногда, когда вы не можете создать трехзвездочный индекс, гораздо лучше игнорировать фразу `WHERE` и обратить внимание на оптимальный порядок строк либо создать покрывающий индекс.

Отдельные индексы по множеству столбцов не помогут MySQL повысить производительность для большинства запросов. MySQL может отчасти решить проблему таких плохо проиндексированных таблиц, применяя стратегию, известную как слияние индексов, которая позволяет запросу ограничить применение нескольких индексов в одной таблице для поиска нужных строк. Запрос может использовать оба индекса, сканируя их одновременно и объединяя результаты. Могут применяться три варианта алгоритма: объединение для условий OR, пересечение для условий AND и объединение пересечений для их комбинаций. В следующем запросе работа двух индексов объединяется, в чем вы можете убедиться, изучив столбец 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
partitions: NULL
type: index_merge
possible_keys: PRIMARY,idx_fk_film_id
key: PRIMARY,idx_fk_film_id
key_len: 2,2
ref: NULL
rows: 29
filtered: 100.00
Extra: Using union(PRIMARY,idx_fk_film_id); Using where
```

MySQL может использовать эту технику для сложных запросов, поэтому для некоторых запросов вы увидите вложенные операции в столбце Extra.

Стратегия слияния индексов иногда работает очень хорошо, но чаще всего это означает, что таблица плохо проиндексирована.

- Если сервер использует пересечение индексов (обычно для условий AND), это обычно означает, что вам нужен один индекс со всеми их столбцами, а не несколько индексов, которые нужно комбинировать.
- Если сервер применяет объединение индексов (обычно для условий OR), то операции буферизации, сортировки и слияния могут задействовать большое количество ресурсов процессора и памяти. Это особенно верно, если не все индексы очень селективные, поэтому сканирование возвращает много строк для операции слияния.
- Напомним, что оптимизатор не учитывает эти затраты — он оптимизирует только количество произвольных чтений страниц. В итоге запрос может оказаться недооцененным, фактически он будет выполняться медленнее, чем обычное сканирование таблицы. Интенсивное использование памяти и процессора также имеет тенденцию влиять на конкурентные запросы, однако вы

не увидите этого воздействия, если будете выполнять запрос изолированно. Иногда более оптимально переписать такие запросы с фразой `UNION`.

Видя слияние индексов в результатах команды `EXPLAIN`, вы должны изучить структуру запроса и таблицы, чтобы убедиться, что их нельзя улучшить. Можете отключить слияние индексов с помощью параметра `optimizer_switch` или переменной. Либо использовать команду `IGNORE INDEX`.

Выбор правильного порядка столбцов

Одна из наиболее распространенных причин проблем, которую мы видели, — это порядок столбцов в индексе. Правильный порядок зависит от запросов, которые будут использовать индекс, и вы должны подумать о том, как выбрать порядок индекса, чтобы строки были отсортированы и сгруппированы наилучшим с точки зрения запроса образом.

Порядок столбцов в многостолбцовом индексе, упорядоченном на основе В-дерева, означает, что индекс сортируется сначала по самому левому столбцу, затем по следующему и т. д. Таким образом, индекс можно просматривать как в прямом, так и в обратном порядке, чтобы удовлетворить запросы с фразами `ORDER BY`, `GROUP BY` и `DISTINCT`, в которых порядок столбцов точно соответствует индексу.

Таким образом, порядок столбцов жизненно важен для многостолбцовых индексов. Он позволяет индексу зарабатывать звезды по трехзвездочной системе Лахденмаки и Лича (подробнее о трехзвездочной системе говорится в подразделе «Преимущества индексов» ранее в этой главе). В оставшейся части этой главы мы покажем множество примеров того, как это работает.

Существует старое эмпирическое правило выбора порядка столбцов: помещайте наиболее селективные столбцы в индекс первыми. Насколько полезна эта рекомендация? В некоторых случаях она может быть весьма полезна, но обычно гораздо важнее избегать произвольных операций ввода/вывода и сортировок. (Конкретные ситуации отличаются друг от друга, поэтому универсального правила не существует. Уже одно это говорит о том, что это эмпирическое правило, скорее всего, не так полезно, как кажется.) Помещать в индекс сначала наиболее селективные столбцы может быть целесообразно, когда не нужно учитывать сортировку или группировку, поэтому целью индекса является только оптимизация поиска `WHERE`. В таких случаях действительно может оказаться полезным спроектировать индекс так, чтобы он отфильтровывал строки и поэтому лучше всего подходил для запросов, в которых указывается только префикс индекса в разделе `WHERE`. Однако это зависит не только от селективности (общей кардинальности) столбцов, но и от фактических значений, которые вы применяете для поиска строк, — распределения значений. Тем же принципом мы пользовались, когда выбирали оптимальную длину префикса. На самом

деле вам может понадобиться выбрать такой порядок столбцов, чтобы он был максимально селективным для запросов, которые вы будете выполнять чаще всего. В качестве примера возьмем следующий запрос:

```
SELECT * FROM payment WHERE staff_id = 2 AND customer_id = 584;
```

Нужно ли создать индекс для (`staff_id`, `customer_id`) или стоит изменить порядок столбцов на обратный? Мы можем запустить несколько быстрых запросов, чтобы изучить распределение значений в таблице и определить, какой столбец имеет более высокую селективность. Преобразуем запрос, чтобы подсчитать кардинальность каждого предиката в запросе с фразой `WHERE`:

```
mysql> SELECT SUM(staff_id = 2), SUM(customer_id = 584) FROM payment\G
***** 1. row *****
SUM(staff_id = 2): 7992
SUM(customer_id = 584): 30
```

В соответствии с эмпирическим правилом мы должны размещать `customer_id` первым в индексе, потому что предикат соответствует меньшему количеству строк в таблице. Затем можем снова запустить запрос, чтобы узнать, насколько селективен `staff_id` в пределах диапазона строк, выбранных для клиента с данным идентификатором:

```
mysql> SELECT SUM(staff_id = 2) FROM payment WHERE customer_id = 584\G
***** 1. row *****
SUM(staff_id = 2): 17
```

Применяйте эту методику с осторожностью, потому что результаты зависят от конкретных констант, предоставленных для выбранного запроса. Если вы оптимизируете свои индексы для этого запроса, но не для других, производительность сервера в целом может ухудшиться, а некоторые запросы могут выполняться непредсказуемо.

Если вы используете худшую выборку запроса из такого отчета какого-либо инструмента, например `pt-querydigest`, эта методика может помочь определить наиболее полезные индексы для ваших запросов и данных. Но если у вас нет конкретных образцов для запуска, может оказаться, что лучше применять старое эмпирическое правило, которое заключается в том, чтобы проверять кардинальность не по одному запросу:

```
mysql> SELECT COUNT(DISTINCT staff_id)/COUNT(*) AS staff_id_selectivity,
-> COUNT(DISTINCT customer_id)/COUNT(*) AS customer_id_selectivity,
-> COUNT(*)
-> FROM payment\G
***** 1. row *****
staff_id_selectivity: 0.0001
customer_id_selectivity: 0.0373
COUNT(*): 16049
```

Столбец `customer_id` имеет самую высокую селективность, поэтому можно рекомендовать поместить его первым в индексе:

```
ALTER TABLE payment ADD KEY(customer_id, staff_id);
```

Как и в случае с префиксными индексами, проблемы часто возникают из-за особых значений, кардинальность которых превышает нормальную. Например, нам доводилось встречать приложения, которые квалифицировали незалогированных пользователей как гостей. Такие пользователи получали специальный идентификатор пользователя в таблице сеанса и в других местах, где фиксировалась пользовательская активность. Запросы с этим идентификатором пользователя, скорее всего, будут отличаться от других запросов, поскольку, как правило, существует множество сеансов, пользователи которых не залогинились. Мы также не раз видели, как системные аккаунты вызывают аналогичные проблемы. У одного приложения был волшебный аккаунт администратора, который был не настоящим пользователем, а был «другом» всех пользователей сайта. С этого аккаунта могли отправляться уведомления о статусе и другие сообщения. Огромный список «друзей» этого пользователя вызывал серьезные проблемы с производительностью сайта.

Это на самом деле довольно типично. Любое отклонение от стандартной ситуации, даже если оно не является результатом неправильного решения по проектированию и управлению приложением, может вызвать проблемы. Пользователи, у которых действительно много друзей, фотографий, сообщений о статусе и т. п., могут создавать такие же проблемы, как и боты.

Вот реальный пример, с которым мы однажды столкнулись на форуме продукта, где пользователи обменивались историями и впечатлениями о продукте. Запросы этой конкретной формы выполнялись очень медленно:

```
SELECT COUNT(DISTINCT threadId) AS COUNT_VALUE
FROM Message
WHERE (groupId = 10137) AND (userId = 1288826) AND (anonymous = 0)
ORDER BY priority DESC, modifiedDate DESC
```

Кажется, что у этого запроса не слишком удачный индекс, поэтому клиент попросил нас посмотреть, можно ли его улучшить. Команда `EXPLAIN` выдала следующую информацию:

```
id: 1
select_type: SIMPLE
table: Message
type: ref
key: ix_groupId_userId
key_len: 18
ref: const,const
rows: 1251162
Extra: Using where
```

Индекс, который MySQL применила для этого запроса по столбцам (`groupId`, `userId`), кажется удачным выбором при отсутствии информации о кардинальности столбцов. Однако, когда мы посмотрели, сколько строк соответствует этому идентификатору пользователя и идентификатору группы, возникла совсем другая картина:

```
mysql> SELECT COUNT(*), SUM(groupId = 10137),
      -> SUM(userId = 1288826), SUM(anonymous = 0)
      -> FROM Message\G
***** 1. row *****
count(*): 4142217
sum(groupId = 10137): 4092654
sum(userId = 1288826): 1288496
sum(anonymous = 0): 4141934
```

Оказалось, что этой группе соответствовали почти все строки в таблице, а пользователю — 1,3 млн строк. В этом случае подходящего индекса просто не существует! Это вызвано тем, что данные были перенесены из другого приложения, а все сообщения приписаны пользователю с правами администратора и группе как части процесса импорта. Решение проблемы состояло в следующем: изменить код приложения таким образом, чтобы распознавать этого специального пользователя и его группу и не выполнять для него запросов.

Мораль этой небольшой истории заключается в том, что эмпирические правила и эвристики могут быть полезны, но вы должны быть внимательными и не рассчитывать на то, что усредненная производительность соответствует производительности любого случая, включая особые. Особые случаи могут ухудшить производительность всего приложения.

Напоследок добавим, что хотя интересно исследовать эмпирическое правило селективности и кардинальности, другие факторы, такие как сортировка, группировка и наличие условий диапазона в разделе запроса `WHERE`, могут иметь гораздо большее значение для производительности запроса.

Кластерные индексы

*Кластерные индексы*¹ — это не отдельный тип индексов. Скорее, это подход к хранению данных. Точные детали различаются в зависимости от реализации, но кластерные индексы InnoDB фактически хранят в одной и той же структуре и индекс, упорядоченный на основе B-дерева, и сами строки.

¹ Пользователи Oracle уже знакомы с термином `index-organized table`, который означает то же самое.

Когда над таблицей построен кластерный индекс, ее строки хранятся в листьях индекса. Термин «кластерный» означает, что строки с близкими значениями ключа хранятся недалеко друг от друга¹. У вас может быть только один кластерный индекс для каждой таблицы, потому что невозможно хранить строки в двух местах одновременно. (Покрывающие индексы позволяют эмулировать несколько кластерных индексов, но об этом поговорим позднее.)

Подсистемы хранения отвечают за реализацию индексов, однако не все подсистемы хранения поддерживают кластерные индексы. В этом разделе мы сосредоточимся на InnoDB, но принципы, которые обсудим, скорее всего, будут хотя бы частично верны для любой подсистемы хранения, которая поддерживает кластерные индексы сейчас или станет поддерживать их в будущем.

На рис. 7.3 показано, как записи располагаются в кластерном индексе. Обратите внимание на то, что листья содержат сами строки, а узловые страницы — только проиндексированные столбцы. В рассматриваемом примере индексированный столбец содержит целочисленные значения.

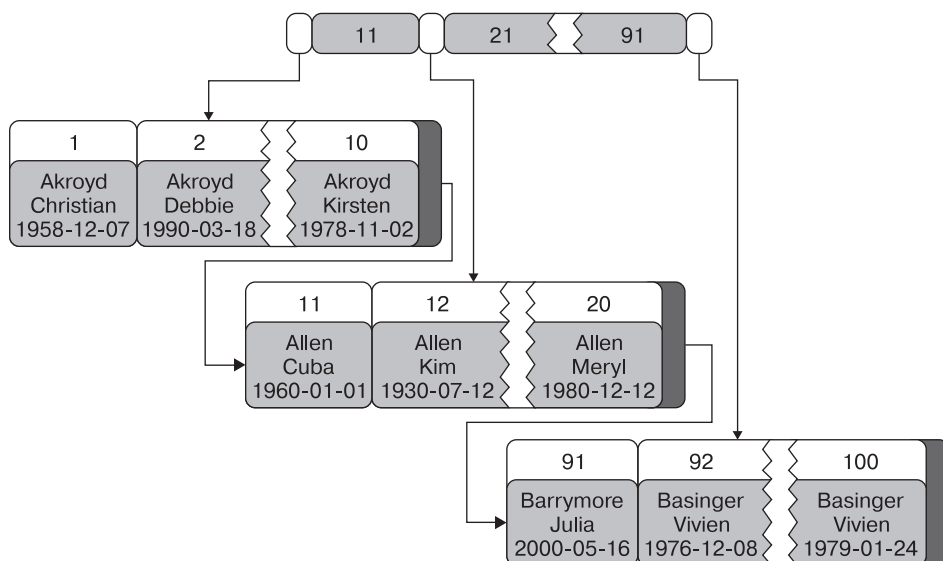


Рис. 7.3. Расположение записей в кластерном индексе

Некоторые серверы баз данных позволяют выбрать, какой индекс сделать кластерным, но ни одна из встроенных подсистем хранения MySQL на момент

¹ Это не всегда верно, как вы увидите через некоторое время.

написания книги не делает этого. InnoDB кластеризует данные по первичному ключу. Это означает, что индексированный столбец на рис. 7.3 является столбцом, содержащим первичный ключ.

Если вы не определите первичный ключ, InnoDB вместо этого попытается использовать уникальный индекс, не допускающий значений NULL. Если такого индекса не существует, InnoDB определит для вас скрытый первичный ключ, а затем по нему выполнит кластеризацию таблицы. Недостаток скрытых первичных ключей заключается в том, что увеличенное значение для них совместно задействуется всеми таблицами, использующими скрытый первичный ключ, что приводит к увеличению числа конфликтов мьютексов за общий ключ.

Кластеризация данных имеет несколько очень важных преимуществ.

- Вы можете хранить связанные данные близко друг к другу. Например, при реализации почтового ящика можете выполнить кластеризацию таблицы по столбцу `user_id`, тем самым для выборки всех сообщений одного пользователя нужно будет прочитать с диска лишь небольшое количество страниц. Если вы не применяли кластеризацию, то для каждого сообщения может потребоваться отдельная операция дискового ввода/вывода.
- Быстрый доступ к данным. Кластерный индекс содержит как индекс, так и данные вместе в одном B-дереве, поэтому извлечение строк из кластерного индекса обычно выполняется быстрее, чем сопоставимый поиск в некластерном индексе.
- Запросы, использующие покрывающие индексы, могут задействовать значения первичного ключа, содержащиеся в листе.

Эти преимущества могут дать возможность значительно повысить производительность, если вы спроектируете свои таблицы и запросы с учетом их преимуществ. Однако у кластерных индексов есть и недостатки.

- Кластеризация дает наибольшее улучшение, когда рабочая нагрузка характеризуется большим количеством операций ввода/вывода. Если данные помещаются в памяти и порядок доступа к ним не имеет большого значения, то кластерные индексы не принесут большой пользы.
- Скорость операций вставки сильно зависит от порядка вставки. Вставка строк в порядке, соответствующем первичному ключу, — самый быстрый способ загрузки данных в таблицу InnoDB. После загрузки большого количества данных, если вы не загружали строки в порядке первичного ключа, целесообразно реорганизовать таблицу с помощью `OPTIMIZE TABLE`.
- Обновление столбцов кластерного индекса весьма затратно, поскольку InnoDB вынуждена перемещать каждую обновленную строку на новое место.

- Для таблиц с кластерным индексом вставка новых строк или обновление первичного ключа, требующее перемещения строки, могут привести к разделению страницы. Оно происходит тогда, когда значение ключа строки таково, что строка должна быть помещена в страницу, заполненную данными. Подсистема хранения вынуждена разделить страницу на две, чтобы разместить строку. Разделение страниц может привести к тому, что таблица займет больше места на диске.
- Полное сканирование кластерных таблиц может оказаться медленным, особенно если строки менее плотно упакованы или хранятся неупорядоченно из-за разделения страниц.
- Вторичные (некластерные) индексы могут оказаться больше, чем можно было бы ожидать, поскольку в их листьях хранятся значения столбцов, составляющих первичный ключ.
- Для получения доступа к данным по вторичному индексу требуется просматривать два индекса вместо одного.

Последний пункт может быть не совсем ясен. Почему для вторичного индекса требуется два поиска по индексу? Ответ заключается в природе указателей на строки, хранящихся во вторичном индексе. Помните, что лист содержит не указатель на физический адрес строки, а значение ее первичного ключа.

Это означает, что для нахождения строки по вторичному индексу подсистема хранения сначала находит лист во вторичном индексе, а затем использует хранящиеся там значения первичного ключа для навигации по первичному ключу и поиска строки. Это двойная работа — два прохода по В-дереву вместо одного¹. В InnoDB адаптивный хеш-индекс (упомянутый ранее в пункте «Индексы, упорядоченные на основе В-дерева») может помочь уменьшить эти потери.

Структура данных InnoDB

Чтобы лучше понять кластерные индексы, посмотрим, как InnoDB разместит данные следующей таблицы:

```
CREATE TABLE layout_test (  
  col1 int NOT NULL,  
  col2 int NOT NULL,  
  PRIMARY KEY(col1),  
  KEY(col2)  
);
```

¹ Кстати, некластерные индексы не всегда могут обеспечить поиск строк за один проход. Когда строка изменяется, она может больше не помещаться на свое исходное место, поэтому вы можете столкнуться с фрагментированными строками или переадресацией в таблице, что увеличит работу по поиску строки.

Предположим, что таблица заполнена значениями первичного ключа от 1 до 10 000, вставленными в случайном порядке, а затем выполнена оптимизация с помощью команды `OPTIMIZE TABLE`. Другими словами, данные размещены на диске оптимальным образом, но строки могут располагаться в случайном порядке. Элементам столбца `col2` присвоены случайные значения в диапазоне от 1 до 100, поэтому в таблице существует множество дубликатов. InnoDB хранит таблицу, как показано на рис. 7.4.

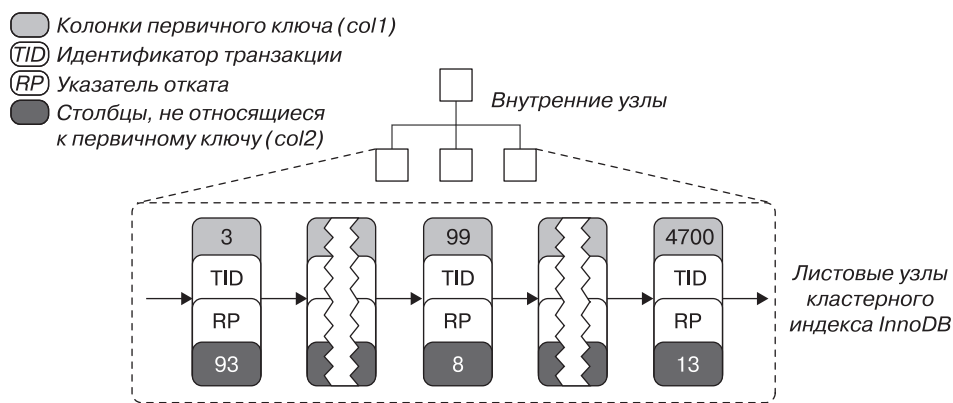


Рис. 7.4. Структура первичного ключа InnoDB для таблицы `layout_test`

Каждый лист в кластерном индексе содержит значение первичного ключа, идентификатор транзакции и указатель отката, которые InnoDB использует для поддержки транзакций и механизма MVCC, а также остальные столбцы (в данном случае `col2`). Если первичный ключ создан по префиксу столбца, InnoDB включает полное значение столбца с остальными столбцами.

Листовые узлы вторичного индекса InnoDB содержат значения первичного ключа, которые служат указателями на строки. Эта стратегия уменьшает объем работы, необходимой для поддержки вторичных индексов при перемещении строк или разделении страницы данных. Использование значений первичного ключа строки в качестве указателя увеличивает размер индекса, но это означает, что InnoDB может перемещать строку без обновления указателей на нее.

Рисунок 7.5 иллюстрирует индекс по столбцу `col2` для таблицы из примера. Каждый лист содержит индексированные столбцы (в данном случае только `col2`), за которыми следуют значения первичного ключа (`col1`).

На этих диаграммах показаны листовые узлы В-дерева, но мы намеренно опустили подробности о нелистовых узлах. Каждый неконечный узел В-дерева InnoDB содержит индексированный столбец (столбцы) плюс указатель на следующий,

более глубокий узел, который может быть либо другим неконечным, либо конечным узлом. Это относится ко всем индексам В-дерева, кластеризованным и вторичным.

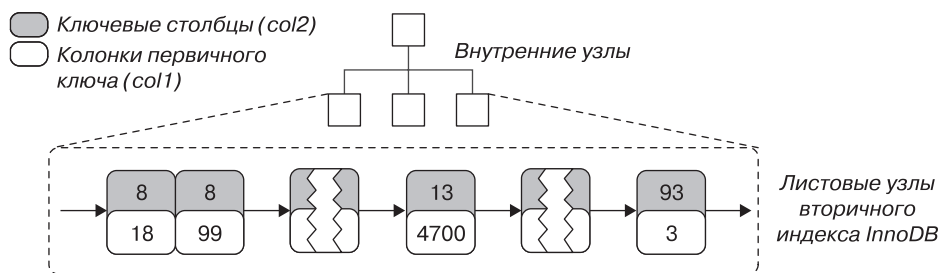


Рис. 7.5. Размещение вторичного индекса для таблицы layout_test в InnoDB

Вставка строк в порядке первичного ключа в InnoDB

Если вы используете InnoDB и вам не требуется какая-либо специфическая кластеризация, целесообразно определить *суррогатный ключ*, являющийся первичным ключом, значение которого не вытекает из данных вашего приложения. Обычно проще всего это сделать с помощью столбца с атрибутом `AUTO_INCREMENT`. Это гарантирует, что значение поля, по которому построен первичный ключ, монотонно возрастает, тем самым обеспечивается лучшая производительность соединений с применением первичных ключей.

Стоит избегать случайных (непоследовательных и распределенных по большому набору значений) кластерных ключей, особенно для рабочих нагрузок, связанных с большим вводом-выводом. Например, использование значений `UUID` — это плохой выбор с точки зрения производительности: вставка в кластерный индекс будет случайной, что является худшим сценарием и не обеспечивает полезной кластеризации данных.

Чтобы продемонстрировать это, мы провели эталонное тестирование производительности для двух ситуаций. В первом случае выполнялась вставка в таблицу `userinfo` с целочисленным идентификатором. Таблица была определена 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 ключ¹.

Во втором случае рассмотрим таблицу с именем `userinfo_uuid`. Она практически идентична таблице `userinfo`, за исключением того, что ее первичным ключом является UUID, а не целое число:

```

CREATE TABLE userinfo_uuid (
    uuid varchar(36) NOT NULL,
    ...

```

Мы выполнили эталонное тестирование обеих таблиц. Сначала вставили 1 млн записей в обе таблицы на сервере с объемом памяти, достаточным для размещения в ней индексов. Затем вставили 3 млн строк в те же таблицы, что сделало индексы больше настолько, что они перестали помещаться в памяти. В табл. 7.1 сравниваются результаты эталонного тестирования.

Таблица 7.1. Эталонное тестирование вставки строк в таблицы InnoDB

Таблица	Число строк	Время, с	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

¹ Стоит отметить, что это реальная таблица с вторичными индексами и множеством столбцов. Если мы их удалим и проведем эталонное тестирование производительности первичного ключа, разница будет еще больше.

Обратите внимание на то, что не только для вставки строк с первичным ключом UUID требуется больше времени, но и размер результирующих индексов немного больше. Отчасти это связано с большим размером первичного ключа. Но, несомненно, влияние оказали также разделение страниц и неизбежная фрагментация.

Чтобы понять, почему так происходит, посмотрим, что случилось в индексе, когда мы вставили данные в первую таблицу. На рис. 7.6 показано, как вставляемые строки сначала заполняют одну страницу, а затем переходят на следующую.

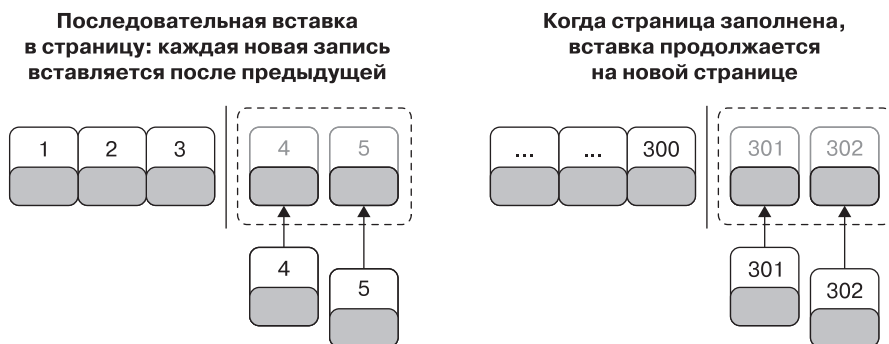


Рис. 7.6. Вставка последовательных значений индекса в кластерный индекс

Как показано на рис. 7.6, InnoDB сохраняет каждую запись непосредственно после предыдущей, потому что значения первичного ключа являются последовательными. Когда коэффициент заполнения страницы достигает максимально допустимого значения (первоначальный коэффициент заполнения InnoDB составляет всего 15/16, чтобы оставалось место для последующих модификаций), следующая запись размещается на новой странице. После окончания последовательной загрузки данных страницы первичных ключей оказались почти заполненными упорядоченными записями, что весьма желательно. (Однако страницы вторичного индекса вряд ли будут отличаться.)

Сравните этот процесс с тем, что произошло, когда мы вставили данные во вторую таблицу с кластерным индексом по столбцу, содержащему UUID (рис. 7.7).

Поскольку значение первичного ключа в каждой последующей строке не обязательно больше, чем в предыдущей, InnoDB не всегда может поместить новую строку в конец индекса. Она должна найти для строки подходящее положение — обычно где-то в середине уже существующих данных — и освободить для нее

место. Это обуславливает значительную дополнительную работу и приводит к неоптимальному размещению данных. Приведем краткий список недостатков такой ситуации.

- Страница, на которую должна попасть строка, может быть сброшена на диск и удалена из кэша или вообще никогда не помещалась в кэши, и в этом случае InnoDB придется искать ее и считывать с диска, прежде чем она сможет вставить новую строку. Из-за этого приходится выполнять много произвольных операций ввода/вывода.
- Когда операции вставки осуществляются не по порядку, InnoDB часто приходится разделять страницы, чтобы освободить место для новых строк. Это требует перемещения большого количества данных и изменения как минимум трех страниц вместо одной.
- Из-за разделения страницы оказываются заполненными беспорядочно и неплотно, что нередко приводит к фрагментации.

Вставка UUID: новые записи могут быть вставлены между ранее вставленными записями, вынуждая их перемещаться



Рис. 7.7. Вставка непоследовательных значений в кластерный индекс

После загрузки таких случайных значений в кластерный индекс вам, вероятно, следует выполнить `OPTIMIZE TABLE`, чтобы перестроить таблицу и оптимально заполнить страницы.

Мораль этой истории заключается в том, что при использовании InnoDB вы должны стремиться вставлять данные в порядке, соответствующем первичному ключу, а также применять такой кластерный ключ, который будет монотонно возрастать для каждой новой строки.

КОГДА ВСТАВКА В ПОРЯДКЕ ПЕРВИЧНОГО КЛЮЧА — ЭТО ПЛОХО

Для рабочих нагрузок с высокой степенью параллелизма вставка в порядке первичного ключа может фактически создать точки конкуренции в InnoDB. Горячей точкой является последняя страница первичного индекса. Поскольку все вставки происходят именно здесь, возникает состязание за блокировку следующего ключа. Еще одна горячая точка — механизм блокировки `AUTO_INCREMENT`. Если вы сталкиваетесь с такими проблемами, можете перепроектировать таблицу или приложение или настроить `innodb_autoinc_lock_mode`. Если ваша версия сервера не поддерживает `innodb_autoinc_lock_mode`, стоит перейти на более новую версию InnoDB, которая будет работать лучше для этой конкретной рабочей нагрузки.

Покрывающие индексы

Обычно индексы рекомендуется создавать для раздела `WHERE` запроса, но это только часть дела. Индексы должны быть разработаны для всего запроса, а не только для предложения `WHERE`. Индексы действительно позволяют эффективно находить строки, но MySQL также может использовать индекс для извлечения данных столбца, не считывая строку таблицы. В конце концов, листья индекса содержат те значения, которые они индексируют. Так зачем просматривать саму строку, если чтение индекса уже может дать вам нужные данные? Индекс, который содержит (покрывает) все данные, необходимые для формирования результатов запроса, называется *покрывающим*. Важно отметить, что для покрытия данных можно использовать только индексы, упорядоченные на основе структуры В-дерева.

Покрывающие индексы могут стать очень мощным инструментом и значительно увеличить производительность. Рассмотрим преимущества считывания индекса вместо самих данных.

- Записи индекса обычно намного компактнее полной строки, поэтому, читая только индексы, MySQL может обращаться к значительно меньшему количеству данных. Это очень важно для кэшированных рабочих нагрузок, когда большая часть времени отклика определяется в основном копированием данных. Это полезно и для рабочих нагрузок, связанных с большим количеством операций ввода/вывода, поскольку индексы меньше данных и лучше помещаются в память.
- Индексы отсортированы по индексируемым значениям (по крайней мере в пределах страницы), поэтому для поиска по диапазону, характеризующегося большим объемом ввода/вывода, потребуется меньше операций обращения к диску по сравнению с извлечением каждой строки из произвольного места

хранения. Вы даже можете оптимизировать таблицу командой `OPTIMIZE`, чтобы получить полностью отсортированные индексы, в результате чего для простых запросов по диапазону доступ к индексу будет вообще последовательным.

- Покрывающие индексы особенно полезны для таблиц InnoDB из-за кластерных индексов InnoDB. Вторичные индексы InnoDB хранят значения первичного ключа строки в листьях. Таким образом, вторичный индекс, покрывающий запрос, позволяет избежать повторного поиска индекса в первичном ключе.

Во всех этих сценариях выполнение запроса с использованием индекса обычно становится значительно менее затратным, чем извлечение всей записи из таблицы.

Запустив команду `EXPLAIN` для запроса, который покрывается индексом (он так и называется — покрываемый индексом запрос), вы увидите в столбце `Extra` сообщение `Using index`¹. Например, таблица `sakila.inventory` имеет много-столбцовый индекс (`store_id, film_id`). MySQL может использовать его для выполнения запроса, который обращается только к этим двум столбцам, например:

```
mysql> EXPLAIN SELECT store_id, film_id FROM sakila.inventory\G
***** 1. row *****
id: 1
select_type: SIMPLE
table: inventory
partitions: NULL
type: index
possible_keys: NULL
key: idx_store_id_film_id
key_len: 3
ref: NULL
rows: 4581
filtered: 100.00
Extra: Using index
```

В большинстве подсистем хранения индекс может покрывать только те запросы, которые обращаются к столбцам, являющимся частью индекса. Тем не менее он позволяет немного развить эту оптимизацию. Напомним, что вторичные индексы InnoDB хранят в листьях значения первичного ключа. Это означает, что вторичные индексы InnoDB фактически имеют дополнительные столбцы, которые InnoDB может задействовать для покрытия запросов.

¹ Легко спутать сообщение `Using index` в столбце `Extra` с сообщением `index` в столбце `type`. Однако это совершенно разные сообщения. Столбец `type` не имеет ничего общего с покрывающими индексами — он показывает тип доступа запроса или поясняет, как запрос будет находить строки. В руководстве пользователя по MySQL это называется типом соединения.

Например, в таблице `sakila.actor`, использующей InnoDB, построен индекс по столбцу `last_name`, 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
partitions: NULL
type: ref
possible_keys: idx_actor_last_name
key: idx_actor_last_name
key_len: 182
ref: const
rows: 2
filtered: 100.00
Extra: Using index
```

Использование сканирования индекса для сортировки

В MySQL есть два способа получения упорядоченных результатов: она может использовать операцию сортировки или просматривать индекс по порядку. О том, что MySQL собирается просматривать индекс, можно понять, обнаружив слово `index` в столбце `type` таблицы, выводимом командой `EXPLAIN` (не путайте с сообщением `Using index` в столбце `Extra`).

Просмотр самого индекса выполняется быстро, потому что для этого просто требуется переместиться от одной записи индекса к другой. Однако, если MySQL не использует индекс для покрытия запроса, ей придется 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 медленнее, чем последовательный просмотр таблицы, особенно для рабочих нагрузок, характеризующихся большим объемом ввода/вывода.

MySQL может использовать один и тот же индекс как для сортировки, так и для поиска строк. По возможности рекомендуется спроектировать индексы так, чтобы они были полезны для решения обеих задач. Упорядочение результатов по индексу работает только в том случае, если порядок индекса точно такой же, как в разделе `ORDER BY`, и все столбцы отсортированы в одном направлении — по возрастанию или по убыванию¹. Если запрос объединяет несколько таблиц,

¹ Если вам нужно сортировать в разных направлениях, воспользуйтесь приемом, предусматривающим хранение обратных или отрицательных значений.

он работает, только если все столбцы в предложении **ORDER BY** ссылаются на первую таблицу. Предложение **ORDER BY** имеет то же ограничение, что и поисковые запросы: оно должно формировать крайний левый префикс индекса. Во всех остальных случаях MySQL использует сортировку.

Есть один случай, когда в разделе **ORDER BY** не обязательно должен быть указан самый левый префикс индекса: если для начальных столбцов индекса в параметрах **WHERE** или **JOIN** заданы константы, они могут заполнить пропуски в индексе. Например, в таблице **rental** в стандартной базе данных **Sakila Sample Database** имеется индекс (**rental_date, inventory_id, customer_id**):

```
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),
...
);
```

MySQL использует индекс **rent_date** для сортировки результатов в следующем запросе, как видно из отсутствия **filesort**¹ в **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 в условии **ORDER BY** указан не левый префикс индекса, потому что мы задали условие равенства для первого столбца в индексе.

Вот еще несколько запросов, которые могут использовать индекс для сортировки. В следующем примере это работает, потому что в запросе задана константа для первого столбца индекса, а в разделе **ORDER BY** предусмотрена сортировка по второму столбцу. В совокупности эти два элемента образуют крайний левый префикс в индексе:

```
... WHERE rental_date = '2005-05-25' ORDER BY inventory_id DESC;
```

¹ В MySQL это называется **filesort**, но не обязательно применяет файлы. Она обращается к диску, только если не может отсортировать данные в памяти.

Следующий запрос тоже работает¹, потому что два столбца, указанные в разделе **ORDER BY**, являются крайним левым префиксом индекса:

```
... WHERE rental_date > '2005-05-25' ORDER BY rental_date, inventory_id;
```

Приведем несколько примеров, в которых индекс не может применяться для сортировки.

В следующем запросе используются 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 **ORDER BY** относится к столбцу, которого нет 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 крайний левый префикс индекса:

```
... WHERE rental_date = '2005-05-25' ORDER BY customer_id;
```

Этот запрос содержит условие поиска по диапазону в первом столбце, 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, когда MySQL теоретически могла бы использовать индекс для сортировки соединения, но не делает этого, потому что оптимизатор помещает таблицу **film_actor** на второе место в соединении:

```
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                            |
+-----+-----+-----+
```

¹ Следует отметить, что, хотя для сортировки может использоваться индекс, в наших тестах оптимизатор в версии 8.0.25 не применял индекс до тех пор, пока мы не задействовали условие **FORCE INDEX FOR ORDER BY** — еще одно напоминание о том, что оптимизатор может делать не то, что вы ожидаете, и вы всегда должны проверять запрос с помощью **EXPLAIN**.

Одним из наиболее важных способов применения упорядочения по индексу является запрос, в котором есть как раздел `ORDER BY`, так и раздел `LIMIT`.

Избыточные и дублирующиеся индексы

К сожалению, MySQL позволяет создавать дублирующиеся индексы для одного и того же столбца. Она только вернет предупреждение, но не помешает вам сделать это. MySQL должна поддерживать каждый дублирующийся индекс отдельно, и оптимизатор запросов будет учитывать каждый из них при оптимизации запросов. Это может серьезно ухудшить производительность, а также требует дополнительного места на диске.

Дублирующиеся индексы — это индексы одного типа, созданные для одного и того же набора столбцов в одном и том же порядке. Старайтесь избегать их создания, а обнаружив — удаляйте.

Иногда вы можете создавать дублирующиеся индексы, даже не подозревая об этом. Например, посмотрите на следующий код:

```
CREATE TABLE test (  
  ID INT NOT NULL PRIMARY KEY,  
  INT NOT NULL,  
  INT NOT NULL,  
  UNIQUE(ID),  
  INDEX(ID)  
) ENGINE=InnoDB;
```

Неопытный пользователь может подумать, что здесь столбец является первичным ключом, добавляет ограничение `UNIQUE` — и, кроме того, создан индекс для применения в запросах. На самом деле MySQL реализует ограничения `UNIQUE` и `PRIMARY KEY` с индексами, так что это фактически создает три индекса для одного и того же столбца! Обычно в таких действиях нет смысла, если только вы не хотите иметь разные типы индексов для одного и того же столбца с целью удовлетворения различных типов запросов¹.

Избыточные индексы немного отличаются от дублирующихся. Если существует индекс по паре столбцов (A, B), то отдельный индекс по столбцу (A) будет избыточным, поскольку он является префиксом первого индекса. То есть индекс по (A, B) может использоваться и как индекс только по одному столбцу (A). (Этот тип избыточности применяется только к индексам, упорядоченным на

¹ Индекс не обязательно является дубликатом, если это другой тип индекса, — часто есть веские причины иметь `KEY(col)` и `FULLTEXT KEY(col)`.

основе В-дерева.) Однако индекс по столбцам (В, А) не будет избыточным, как и индекс по столбцу (В), потому что В не является крайним левым префиксом (А, В). Более того, индексы различных типов (например, хеш-индексы или полнотекстовые индексы) не являются избыточными по отношению к индексам, упорядоченным на основе В-дерева, независимо от того, какие столбцы они покрывают. Избыточные индексы обычно появляются при добавлении индексов к таблице. Например, кто-то может добавить индекс по (А, В) вместо расширения существующего индекса по столбцу (А) для покрытия (А, В). Другой сценарий, при котором это может произойти, — изменение индекса таким образом, чтобы он покрывал (А, ID). Столбец ID является первичным ключом, поэтому, если вы используете InnoDB, он уже включен в индекс.

В большинстве случаев вам не нужны избыточные индексы, и, чтобы избежать их появления, вам следует расширять существующие индексы, а не добавлять новые. Тем не менее бывают случаи, когда избыточные индексы необходимы вам из соображений производительности. Расширение существующего индекса может значительно увеличить его размер и снизить производительность для некоторых запросов.

Например, если у вас есть индекс по целочисленному столбцу и вы расширяете его длинным столбцом типа VARCHAR, он может стать значительно медленнее. В частности, это относится к случаям, когда ваши запросы используют индекс в качестве покрывающего. Рассмотрим следующую таблицу `userinfo`:

```
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',  
  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)  
) ENGINE=InnoDB
```

Эта таблица содержит 1 млн строк, и для каждого значения столбца `state_id` существует около 20 000 записей. В ней есть индекс по столбцу `state_id`, который полезен для следующего запроса (мы называем его Q1):

```
SELECT count(*) FROM userinfo WHERE state_id=5;
```

Простой эталонный тест показывает скорость выполнения этого запроса — почти 115 запросов в секунду (QPS). Теперь рассмотрим связанный запрос Q2, который извлекает несколько столбцов, а не просто подсчитывает строки:

```
SELECT state_id, city, address FROM userinfo WHERE state_id=5;
```

Результат для него — меньше 10 запросов в секунду¹. Простое решение для повышения его производительности состоит в расширении индекса до трех столбцов (`state_id`, `city`, `address`) так, чтобы индекс покрывал запрос:

```
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 действительно хотим ускорить оба запроса, нужно оставить оба индекса, даже несмотря на то, что индекс по одному столбцу избыточен. В табл. 7.2 показаны подробные результаты для обоих запросов и стратегий индексирования.

Таблица 7.2. Сравнительные результаты по количеству запросов в секунду для запросов SELECT с различными стратегиями индексирования

Запрос	Только <code>state_id</code>	Только <code>state_id_2</code>	Оба, <code>state_id</code> и <code>state_id_2</code>
Q1	108,55	100,33	107,97
Q2	12,12	28,04	28,06

Недостатком наличия двух индексов являются затраты на их поддержку. В табл. 7.3 показано, сколько времени требуется, чтобы вставить в таблицу 1 млн строк.

Таблица 7.3. Скорость вставки миллиона строк с различными стратегиями индексирования

Сценарий	Только <code>state_id</code>	Оба, <code>state_id</code> и <code>state_id_2</code>
InnoDB, достаточно памяти для обоих индексов	108,55	107,97

¹ В этом случае все данные помещаются в память. Если размер таблицы велик и рабочая нагрузка характеризуется большим объемом операций ввода/вывода, то разница между этими числами будет намного больше. Нередко запросы COUNT() становятся в 100 и более раз быстрее с покрывающим индексом.

Как видите, вставка новых строк в таблицу с большим количеством индексов происходит 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.

Избыточные и дублирующиеся индексы нужно просто удалять, однако сначала идентифицировать их. Вы можете писать различные сложные запросы к таблицам `INFORMATION_SCHEMA`, но есть и более простые методы. Можно использовать инструмент `ptduplicate-key-checker`, включенный в пакет `Percona Toolkit`, который анализирует структуру таблиц и выявляет дублирующиеся или избыточные индексы.

Будьте осторожны при определении того, какие индексы являются кандидатами на удаление или расширение. Напомним, что в `InnoDB` индекс по столбцу (`A`) в нашем примере действительно эквивалентен индексу по столбцам (`A, ID`), потому что первичный ключ добавляется к листу вторичного индекса. Если вы выполняете запрос типа `WHERE A = 5 ORDER BY ID`, индекс будет очень полезен. Но если расширите индекс до двух столбцов (`A, B`), тогда он на самом деле станет (`A, B, ID`) и запрос начнет использовать файловую сортировку для раздела `ORDER BY` запроса. Стоит тщательно проверить запланированные изменения с помощью такого инструмента, как `ptupgrade` из `Percona Toolkit`.

В обоих случаях рассмотрите возможность применения функции невидимого индекса `MySQL 8.0` перед удалением индекса. С ее помощью вы можете выполнить оператор `ALTER TABLE`, изменяя индекс таким образом, чтобы он был помечен как невидимый. Это значит, что оптимизатор будет игнорировать его при планировании запросов. Обнаружив, что индекс, который вы собирались удалить, был важен, вы можете легко снова сделать его видимым, а не создавать повторно.

Неиспользуемые индексы

Помимо повторяющихся и избыточных индексов, у вас могут быть индексы, которые сервер просто не использует. Они лежат мертвым грузом, и вам следует подумать о том, чтобы избавиться от них¹.

Лучший способ определить неиспользуемые индексы — это `performance_schema` и `sys`, которые мы подробно рассмотрели в главе 3. Схема `sys` создает

¹ Некоторые индексы функционируют как ограничивающие уникальность, поэтому, даже если индекс не используется для запросов, с его помощью можно предотвращать дублирование значений.

представление таблицы `table_io_waits_summary_by_index_usage`, которая может легко сказать нам, какие индексы не используются:

```
mysql> SELECT * FROM sys.schema_unused_indexes;
```

object_schema	object_name	index_name
sakila	actor	idx_actor_last_name
sakila	address	idx_fk_city_id
sakila	address	idx_location
sakila	payment	fk_payment_rental

.. trimmed for brevity ..

Обслуживание индексов и таблиц

Вы создали таблицы с нужными типами данных и добавили индексы, но работа не закончена: теперь нужно поддерживать таблицы и индексы, чтобы убедиться, что они работают нормально. Тремя основными целями обслуживания таблиц являются обнаружение и исправление повреждений, поддержка точной статистики по индексам и уменьшение фрагментации.

Поиск и исправление повреждений таблицы

Самое плохое, что может случиться с таблицей, — она может быть повреждена. Все подсистемы хранения могут столкнуться с повреждением индекса из-за аппаратных проблем или внутренних ошибок в MySQL или в операционной системе, хотя в InnoDB они возникают очень редко. Поврежденные индексы могут привести к тому, что запросы будут возвращать неправильные результаты, вызывать ошибки дублирования ключей при отсутствии дубликатов или даже приводить к блокировкам и аварийным остановкам. Если вы столкнулись со странным поведением, например с ошибкой, которой, по вашему мнению, быть не должно, — запустите команду `CHECK TABLE`, чтобы проверить, не повреждена ли таблица. (Обратите внимание на то, 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 таблиц и индексов.

Вы можете исправить поврежденные таблицы с помощью команды `REPAIR TABLE`, но ее поддерживают не все подсистемы хранения. В этих случаях вы можете выполнить пустую команду `ALTER`, например, просто указав подсистему, которая

и так уже используется для таблицы в настоящее время. Приведем пример для таблицы типа InnoDB:

```
ALTER TABLE <table> ENGINE=INNODB;
```

В качестве альтернативы можете выгрузить все данные в файл, а затем загрузить обратно. Однако, если повреждение зафиксировано в системной области или в области данных строк таблицы, а не в индексе, вы не сможете использовать ни один из этих вариантов. В этом случае может потребоваться восстановить таблицу из резервных копий или попытаться восстановить данные из поврежденных файлов.

Если вы столкнулись с повреждением в подсистеме хранения InnoDB, значит, произошло что-то серьезное и нужно немедленно с этим разобраться. InnoDB просто не должна повреждаться. Она спроектирована очень устойчивой к повреждениям. Повреждение свидетельствует либо о проблеме с оборудованием, например об ошибках памяти или дисков (вероятно), об ошибках администрирования, например при управлении файлами базы данных извне (вероятно), или программной ошибке InnoDB (маловероятно). Чаще всего это такие ошибки, как, например, попытка создания резервных копий с помощью `rsync`. Не существует ни одного запроса, которого вы должны избегать, потому что он может повредить данные InnoDB. Вы никак не сможете выстрелить себе в ногу. Если вы нарушаете данные InnoDB, делая к ним запросы, это программная ошибка InnoDB, а не ваша вина.

При столкновении с повреждением данных самое главное — попытаться определить, почему это произошло. Не стоит просто восстанавливать данные, поскольку все может повториться. Вы можете восстановить данные, запустив InnoDB в режиме принудительного восстановления с параметром `innodb_force_recovery` (подробности см. в руководстве по MySQL).

Обновление статистики индекса

Когда подсистема хранения предоставляет оптимизатору неточную информацию о количестве строк, которые должен будет просмотреть запрос, или когда план запроса слишком сложен и количество строк будет варьироваться на разных этапах его выполнения, оптимизатор использует статистику индекса для оценки количества строк. Оптимизатор MySQL основан на стоимости, а основной ее показатель — это количество данных, к которым будет обращаться запрос. Если статистика не была сгенерирована или устарела, оптимизатор может принимать неправильные решения. Выход состоит в том, чтобы запустить команду `ANALYZE TABLE`, которая регенерирует статистику.

Вы можете определить кардинальность своих 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 дает довольно много информации об индексах, которая подробно объясняется в руководстве по MySQL. Тем не менее мы хотим обратить ваше внимание на колонку `Cardinality`. Она показывает, сколько различных значений, по оценкам подсистемы хранения, находится в индексе. Можно получить эти данные также из таблицы `INFORMATION_SCHEMA.STATISTICS`. Например, вы можете написать запросы к таблицам `INFORMATION_SCHEMA`, чтобы найти индексы с очень низкой селективностью. Однако имейте в виду, что эти таблицы метаданных могут сильно нагружать сервер, если количество данных на нем велико.

Статистика InnoDB заслуживает более подробного изучения. Она создается выборкой нескольких случайных страниц в индексе в предположении, что оставшая часть индекса выглядит аналогично. Количество выбранных страниц можно устанавливать с помощью переменной `innodb_stats_sample_pages`. Если задать для нее значение большее, чем принятое по умолчанию (8), теоретически это может помочь создать более репрезентативную статистику индекса, особенно для очень больших таблиц.

InnoDB вычисляет статистику для индексов при первом открытии таблиц, запуске `ANALYZE TABLE`, а также значительном изменении размера таблицы.

InnoDB также вычисляет статистику при выполнении запросов к некоторым таблицам `INFORMATION_SCHEMA`, `SHOW TABLE STATUS` и `SHOW INDEX`, а также когда в клиенте командной строки MySQL включено автозаполнение. В действительности это может стать довольно серьезной проблемой на больших серверах со значительным количеством данных или при медленном вводе/выводе. Клиентские программы или инструменты мониторинга, которые делают выборку, могут создать много блокировок и большую нагрузку на сервер, а пользователей станет раздражать медленный запуск. И вы не можете увидеть статистику индекса, не изменяя ее, поскольку команда `SHOW INDEX` будет обновлять статистику. Если отключить параметр `innodb_stats_on_metadata`, вы избежите всех этих проблем.

Уменьшение фрагментации индекса и данных

Индексы, упорядоченные на основе В-дерева, могут становиться фрагментированными, а это способно снизить производительность. Фрагментированные индексы могут быть плохо заполнены и/или располагаются на диске непоследовательно.

Индексы, упорядоченные на основе В-дерева, требуют произвольного доступа к диску для погружения к листьям, поэтому произвольный доступ является правилом, а не исключением. Тем не менее производительность работы с листьями можно улучшить, когда они физически последовательны и плотно упакованы. Если это не так, мы говорим, что они *фрагментированы*, в этом случае поиск по диапазону и полное сканирование индекса могут выполняться во много раз медленнее. Это особенно справедливо для покрывающих индексы запросов.

Данные, хранящиеся в таблицах, также могут стать фрагментированными. Однако механизм фрагментации данных сложнее, чем механизм фрагментации индексов. Существует три типа фрагментации данных.

- *Фрагментация строки.* Этот тип фрагментации наблюдается, когда строка хранится в виде нескольких фрагментов в нескольких местах. Фрагментация строк уменьшает производительность, даже если запросу требуется только одна строка из индекса.
- *Межстрочная фрагментация.* Этот вид фрагментации возникает, когда логически последовательные страницы или строки не хранятся последовательно на диске. Это влияет на такие операции, как полное сканирование таблицы и сканирование диапазона кластерного индекса, которые обычно выигрывают от последовательного расположения данных на диске.
- *Фрагментация свободного пространства.* Этот тип фрагментации возникает, когда на страницах данных много пустого пространства. Это заставляет сервер считывать много ненужных данных и тем самым тратить время впустую.

Для дефрагментации данных вы можете либо запустить команду `OPTIMIZE TABLE`, либо выгрузить данные в файл и заново загрузить их в базу. Эти подходы

работают для большинства подсистем хранения. Для подсистем хранения, которые не поддерживают `OPTIMIZE TABLE`, можно перестроить таблицу с помощью пустой команды `ALTER TABLE`. Достаточно указать ту же подсистему хранения, которая используется для таблицы в данный момент:

```
ALTER TABLE <таблица> ENGINE=<engine>;
```

Резюме

Как видите, индексация — сложная тема! Организация доступа к данным в MySQL и подсистемах хранения данных в сочетании со свойствами индексов делает последние очень мощным и гибким инструментом для управления доступом к данным как на диске, так и в памяти.

В большинстве случаев вы будете использовать индексы, упорядоченные на основе В-дерева. Другие типы индексов больше подходят для специальных целей, и, как правило, будет очевидно, когда вы должны их применять и как они могут улучшить время ответа на запрос. Мы не будем останавливаться на них в этой главе, но стоит напомнить свойства и способы использования индексов, упорядоченных на основе В-дерева.

Приведем три принципа, которые следует учитывать при выборе индексов и написании запросов для их применения.

- Доступ к одной строке — операция медленная, особенно на дисковом хранилище. (Твердотельные накопители работают быстрее при произвольном вводе/выводе, но этот принцип остается справедливым.) Если сервер считывает блок данных из хранилища, а затем обращается только к одной строке в нем, он тратит много времени впустую. Гораздо лучше читать из блока, который содержит много нужных вам строк.
- Доступ к упорядоченным строкам выполняется быстро по двум причинам. Во-первых, последовательный ввод/вывод не требует поиска на диске, поэтому он выполняется быстрее, чем произвольный ввод/вывод, особенно в системах хранения на основе дискового хранилища. Во-вторых, если сервер может считывать данные в требуемом вам порядке, ему не нужно выполнять какую-либо дополнительную работу по их сортировке, а запросам `GROUP BY` не нужно сортировать и группировать строки для вычисления агрегатных значений.
- Доступ только по индексу выполняется быстро. Если индекс содержит все столбцы, необходимые для запроса, подсистеме хранения не нужно искать другие столбцы, просматривая строки в таблице. Это позволяет избежать большого количества операций однострочного доступа, которые, как мы знаем из первого пункта, работают медленно.

В общем, старайтесь выбирать индексы и писать запросы так, чтобы можно было избежать поиска по одной строке, применять встроенный порядок данных, позволяющий не выполнять операции сортировки, и использовать доступ только к индексу. Это соответствует трехзвездочной системе ранжирования, изложенной в книге Лахденмаки и Лича, упомянутой в начале этой главы.

Было бы здорово иметь возможность создавать идеальные индексы для каждого запроса к своим таблицам. К сожалению, иногда для этого требуется чрезмерно большое количество индексов, а в других случаях просто нет возможности создать трехзвездочный индекс для заданного запроса, например, если запрос делается по двум столбцам, данные в одном из которых упорядочены по возрастанию, а в другом — по убыванию. В этих случаях вы должны соглашаться на лучший из возможных вариантов либо использовать альтернативные стратегии, такие как денормализация или сводные таблицы.

Очень важно понимать, как именно работают индексы, и выбирать их, основываясь на этом понимании, а не на эмпирических правилах или эвристиках, таких как «размещайте наиболее селективные столбцы в начале многостолбцовых индексов» или «вы должны индексировать все столбцы, которые появляются в разделе WHERE».

Как узнать, достаточно ли хорошо проиндексирована ваша схема? Как всегда, мы предлагаем ответить на этот вопрос, основываясь на времени отклика. Найдите запросы, которые либо выполняются слишком долго, либо порождают слишком большую нагрузку на сервер. Исследуйте схему, SQL и структуры индексов на наличие запросов, требующих внимания. Определите, должен ли запрос проверять слишком много строк, выполнять сортировку после извлечения данных или использовать временные таблицы, обращаться к данным с помощью произвольного ввода/вывода или искать полные строки в таблице для извлечения столбцов, не включенных в индекс.

Если вы найдете запрос, который не выигрывает от перечисленных преимуществ индексов, посмотрите, можно ли создать лучший индекс для повышения производительности. Если нет, возможно, стоит изменить запрос так, чтобы он мог использовать индекс, который либо уже существует, либо может быть создан. Об этом поговорим в следующей главе.

ГЛАВА 8

Оптимизация производительности запросов

В предыдущих главах мы рассказали про оптимизацию схемы и индексирование, являющиеся необходимыми условиями достижения высокой производительности. Но только этого недостаточно — вам также нужно хорошо спроектировать запросы. Если они составлены плохо, то даже самые лучшие схема и индексы не помогут.

Оптимизация запросов, индексов и схемы идут рука об руку. По мере приобретения опыта написания запросов в MySQL вы начнете понимать, как проектировать таблицы и индексы для поддержки эффективных запросов. Точно так же знания о создании оптимального дизайна схемы будут положительно влиять на запросы, которые вы пишете. Этот процесс требует времени, поэтому рекомендуем возвращаться к трем предыдущим главам по мере освоения нового материала.

Начнем данную главу с общих соображений по разработке запросов — что вы должны учитывать в первую очередь, когда запрос выполняется неэффективно. Затем углубимся в механизмы оптимизации запросов и детали внутреннего устройства сервера. Мы покажем, как узнать, как именно MySQL выполняет конкретный запрос, и вы поймете, как изменить план выполнения запроса. Наконец, рассмотрим несколько случаев, когда MySQL не слишком хорошо оптимизирует запросы, и изучим типичные шаблоны оптимизации запросов, которые помогают MySQL выполнять их более эффективно.

Наша цель состоит в том, чтобы вы лучше разобрались в механизмах выполнения запросов, понимали, что считать эффективным, а что — неэффективным, научились использовать сильные стороны MySQL и избегать слабых.

Почему запросы бывают медленными

Прежде чем пытаться писать быстрые запросы, вспомните, что речь идет о времени отклика. Запросы — это задачи, но они состоят из подзадач, а их выполнение требует времени. Чтобы оптимизировать запрос, вы должны оптимизировать его подзадачи. Для достижения этой цели подзадачи можно вообще устранить, сделав так, чтобы они выполнялись реже или оказались более быстрыми.

В общем, вы можете думать о времени жизни запроса, мысленно проходя за запросом по его диаграмме последовательности от клиента к серверу, где он анализируется, планируется и выполняется, а затем обратно к клиенту. Выполнение — один из наиболее важных этапов жизненного цикла запроса. Оно включает в себя множество обращений к подсистеме хранения для извлечения строк, а также операции после извлечения, такие как группировка и сортировка. Выполняя все эти задачи, запрос затрагивает сеть и процессор, реализует такие операции, как сбор статистики, планирование и блокировка (ожидание мьютекса), и, что наиболее важно, вызывает подсистему хранения для извлечения строк. На эти вызовы затрачивается время в ходе работы с памятью, действий процессора и особенно операций ввода/вывода, если данных нет в памяти. Кроме того, в зависимости от подсистемы хранения может быть задействовано множество переключений контекста и/или системных вызовов.

В любом случае может потребоваться дополнительное время, поскольку операции выполняются, когда это не нужно, выполняются слишком много раз или слишком медленно. Цель оптимизации состоит в том, чтобы избежать этого, устранив или сократив операции или сделав их быстрее.

Однако это не полная и не точная картина жизни запроса. Наша цель здесь — показать важность понимания жизненного цикла, чтобы вы могли учитывать в работе все процессы, требующие затрат времени. Имея это в виду, посмотрим, как оптимизировать запросы.

Основная причина медленных запросов — оптимизация доступа к данным

Главная причина, из-за которой запрос может выполняться медленно, заключается в том, что он обрабатывает слишком большое количество данных. Безусловно, встречаются запросы, которые по своей природе должны перерабатывать очень много всевозможных значений, и с этим ничего не поделаешь. Но это довольно редкая ситуация — большинство плохих запросов можно изменить так, чтобы

они обращались к меньшему объему данных. Мы полагаем, что анализировать медленно выполняющийся запрос следует в два этапа:

- выяснить, не получает ли приложение больше данных, чем вам нужно. Обычно это означает, что оно обращается к слишком большому количеству строк, но не исключено, что может обращаться и к слишком большому количеству столбцов;
- выяснить, не анализирует ли сервер MySQL больше строк, чем ему необходимо.

Не запрашивает ли вы лишние данные у базы?

Некоторые запросы запрашивают больше данных, чем им необходимо, а затем отбрасывают некоторые из них. Это требует дополнительной работы сервера MySQL, приводит к росту расходов на передачу по сети¹ и потребляет ресурсы памяти и процессорного времени на стороне сервера приложений.

Приведем несколько типичных ошибок.

- *Выбор лишних строк.* Одним из широко распространенных заблуждений является предположение, что MySQL предоставляет результаты по мере необходимости, а не формирует и не возвращает результирующий набор целиком. Подобную ошибку мы часто встречали в приложениях, написанных людьми, привыкшими работать с другими СУБД. Эти разработчики привыкли к такой методике, как выдача инструкции `SELECT`, которая возвращает много строк, затем выборка первых N строк и закрытие результирующего набора (например, выборка 100 самых последних статей для новостного сайта, хотя на начальной странице нужно показать только 10). Они полагают, что MySQL предоставит им эти 10 строк, после чего прекратит выполнение запроса, но на самом деле MySQL генерирует полный результирующий набор. Затем клиентская библиотека, получив полный набор данных, отбрасывает большую их часть. Вместо этого следовало бы включить в запрос предложение `LIMIT`.
- *Выбор всех столбцов из соединения нескольких таблиц.* Если вы хотите получить всех актеров, снимавшихся в фильме *Academy Dinosaur*, не стоит писать такой запрос:

```
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 Сетевые накладные расходы оказываются наибольшими, если приложение находится не на том хосте, где сервер, но передача данных между MySQL и приложением небесплатна, даже если они располагаются на одном сервере.

Он возвращает все столбцы из всех трех таблиц. Вместо этого напишите запрос следующим образом:

```
SELECT sakila.actor.* FROM sakila.actor...;
```

- *Выбор всех столбцов.* Всегда следует насторожиться, встречая команду `SELECT *`. Неужели действительно нужны все столбцы без исключения? Скорее всего, нет. Получение всех столбцов может помешать применению таких методов оптимизации, как использование покрывающих индексов, и к тому же увеличит потребление сервером ресурсов: ввода/вывода, памяти и процессора. Некоторые администраторы баз данных вообще запрещают применять команду `SELECT *` повсеместно из-за этого факта и для снижения риска возникновения проблем, когда кто-то изменяет перечень столбцов таблицы.

Разумеется, запрашивать больше данных, чем вам действительно необходимо, не всегда плохо. Во многих случаях, которые мы исследовали, нам говорили, что такой затратный подход упрощает разработку, поскольку позволяет использовать один и тот же фрагмент кода более чем в одном месте. Это разумное соображение, если 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 получения еще какого-то видимого вам преимущества. Выборка и кэширование полных объектов может быть предпочтительнее выполнения множества отдельных запросов, извлекающих только части объекта.

- *Повторный выбор одних и тех же данных.* Действуя неосторожно, довольно легко написать код приложения, которое многократно получает одни и те же данные с сервера базы данных, выполняя один и тот же запрос. Например, желая найти URL-адрес аватара пользователя, который будет отображаться рядом со списком комментариев, вы можете запрашивать его повторно для каждого комментария. Или кэшировать его при первом получении и затем использовать повторно. Последний подход намного эффективнее.

Не слишком ли много данных анализирует MySQL?

Как только вы убедитесь, что все ваши запросы *извлекают* только необходимые вам данные, можете поискать те из них, которые для получения результата анализируют слишком много данных. В MySQL простейшими параметрами, с помощью которых можно определить затраты на запрос, являются:

- время отклика;
- количество проанализированных строк;
- количество возвращенных строк.

Ни один из этих показателей не является идеальным способом измерения стоимости запроса, но они дают грубое представление о том, сколько данных MySQL должна прочитать для выполнения запроса, и приблизительно показывают, насколько быстро работает этот запрос. Все три параметра регистрируются в журнале медленных запросов, поэтому просмотр этого журнала — один из лучших способов выявить запросы, анализирующие слишком много данных.

Время отклика

Не придавайте большого значения времени отклика на запрос. Эй, разве это не противоречит тому, что мы вам говорили? На самом деле нет. По-прежнему верно то, что время отклика важно, но все немного сложнее.

Время отклика состоит из двух частей: времени обслуживания и времени нахождения в очереди. Время обслуживания — это время, которое требуется серверу для фактической обработки запроса. Время нахождения в очереди — это часть времени ответа, в течение которой сервер фактически не выполняет запрос — он чего-то ожидает, например завершения операции ввода/вывода, блокировки строки и т. д. Проблема в том, что вы не можете разбить время отклика на эти составляющие, если не можете измерить их по отдельности, что обычно довольно трудно сделать. В общем, наиболее распространенными и важными ожиданиями, с которыми вы будете сталкиваться, являются ожидания ввода/вывода и блокировки, но стоит учитывать, что подобные ситуации все равно бывают очень разными. Ожидание ввода/вывода и блокировок важны, потому что они наиболее вредны для производительности.

В результате время отклика неодинаково при различных условиях нагрузки. Другие факторы, такие как блокировки подсистемы хранения (например, блокировки строк), высокая степень конкурентности и особенности оборудования, также могут оказывать значительное влияние на время отклика. Большое время отклика может быть как симптомом, так и причиной проблем, и не всегда очевидно, с чем именно мы столкнулись.

Глядя на время отклика на запрос, вы должны спросить себя, является ли время отклика разумным для запроса. В этой книге нет места для подробного объяснения, но вы можете быстро оценить верхнюю границу (quick upper-bound estimate, QUBE) времени отклика на запрос, используя методы, описанные в книге Тапио Лахденмаки и Майка Лича (Tapio Lahdenmaki, Mike Leach) *Relational Database Index Design and the Optimizers* (Wiley). В двух словах это выглядит так: просмотрите план выполнения запроса и задействованные в нем индексы, определите, сколько может потребоваться последовательных и произвольных операций ввода/вывода, и умножьте их количество на время, необходимое вашему оборудованию для их выполнения. Сложив вычисленное время, вы получите эталон, сравнивая с которым поймете, медленно выполняется запрос или нет.

Количество проанализированных и возвращенных строк

При анализе запросов полезно принимать во внимание, какое количество строк было просмотрено сервером, поскольку это показывает, насколько эффективно запросы находят нужные данные. Однако это не идеальный показатель для выявления плохих запросов. Не все обращения к строкам одинаковы. Доступ к более коротким строкам осуществляется быстрее, а выборка строк из памяти выполняется намного быстрее, чем их чтение с диска.

В идеале количество проанализированных строк должно совпадать с количеством возвращенных, но на практике так бывает редко. Например, когда в запросе соединяются несколько таблиц, серверу приходится обращаться к нескольким строкам, чтобы сгенерировать каждую строку в результирующем наборе. Отношение проанализированных строк к возвращенным обычно невелико — скажем, между 1:1 и 10:1, но иногда может быть на несколько порядков больше.

Проанализированные строки и типы доступа

Прикидывая затраты на запрос, учитывайте затраты на поиск одиночной строки в таблице. В MySQL могут использоваться несколько методов поиска и возврата строки. Иногда требуется просмотреть много строк, а иногда удастся создать результирующий набор, не анализируя ни одной.

Методы доступа отображаются в столбце `type` результата, возвращаемого командой `EXPLAIN`. Типами доступа могут быть полное сканирование таблицы, сканирование индекса, сканирование диапазона, поиск по уникальному индексу и возврат константы. Каждый следующий быстрее предыдущего, поскольку требует меньшего количества операций чтения данных. Вам не нужно запоминать все типы доступа, но очень важно понимать общие концепции: что означает сканирование таблицы, сканирование индекса, доступ по диапазону и доступ к единственному значению.

Если вы не получаете оптимальный тип доступа, обычно наилучший способ решить проблему — добавить соответствующий индекс. Мы подробно рассматривали вопросы, связанные с индексированием, в предыдущей главе — теперь вы понимаете, 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:

```
SELECT * FROM sakila.film_actor WHERE film_id = 1;
```

Он возвращает десять строк, и команда EXPLAIN показывает, что MySQL применяет для выполнения запроса тип доступа ref по индексу 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
partitions: NULL
type: ref
possible_keys: idx_fk_film_id
key: idx_fk_film_id
key_len: 2
ref: const
rows: 10
filtered: 100.00
Extra: NULL
```

Как показывает команда EXPLAIN, MySQL оценила, что придется прочитать всего десять строк. Другими словами, оптимизатор запросов знает, что выбранный тип доступа позволит эффективно выполнить запрос. Что бы случилось, если бы не было подходящего индекса для запроса? Тогда MySQL была бы вынуждена 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
partitions: NULL
type: ALL
possible_keys: NULL
key: NULL
key_len: NULL
ref: NULL
rows: 5462
filtered: 10.00
Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

Как и следовало ожидать, тип доступа изменился на полное сканирование таблицы (ALL), и теперь MySQL определила, что для выполнения запроса ей потребуется проанализировать 5462 строки. Фраза Using where в столбце Extra показывает, что сервер MySQL использует запрос WHERE для отбрасывания строк после того, как подсистема хранения их прочитает.

В целом MySQL может применять запрос `WHERE` тремя способами, которые перечислены в порядке от наилучшего к наихудшему.

- Применить указанные условия к операции поиска по индексу, чтобы исключить неподходящие строки. Это происходит на уровне подсистемы хранения.
- Использовать покрывающий индекс (фраза `Using index` в столбце `Extra`), чтобы избежать доступа к самим строкам, и отфильтровать неподходящие строки после выбора каждого результата из индекса. Это происходит на уровне сервера, но не требует чтения строк из таблицы.
- Выбрать строки из таблицы, а затем отфильтровать неподходящие (фраза `Using where` в столбце `Extra`). Это происходит на уровне сервера, причем перед фильтрацией он вынужден считывать строки из таблицы.

Этот пример показывает, насколько важно иметь хорошие индексы. Они позволяют применять при обработке запроса наиболее эффективный тип доступа и анализировать только нужные строки. Однако добавление индекса не всегда означает, что количество проанализированных и возвращенных строк совпадает. Например, вот запрос, использующий агрегатную функцию `COUNT()`¹:

```
mysql> SELECT actor_id, COUNT(*)
      -> FROM sakila.film_actor GROUP BY actor_id;
```

```
+-----+-----+
| actor_id | COUNT(*) |
+-----+-----+
| 1       | 19       |
| 2       | 25       |
| 3       | 22       |
.. omitted..
| 200     | 20       |
+-----+-----+
200 rows in set (0.01 sec)
```

Этот запрос, как показано, возвращает только 200 строк, но сколько строк сервер должен прочитать для построения результирующего набора? Мы можем проверить это с помощью команды `EXPLAIN`, как говорилось в предыдущей главе:

```
mysql> EXPLAIN SELECT actor_id, COUNT(*)
      -> FROM sakila.film_actor GROUP BY actor_id\G
***** 1. row *****
id: 1
select_type: SIMPLE
table: film_actor
partitions: NULL
```

¹ Дополнительные сведения по этой теме см. в подразделе «Оптимизация запросов `COUNT()`» далее в этой главе.

```
type: index  
possible_keys: PRIMARY  
key: PRIMARY  
key_len: 4  
ref: NULL  
rows: 5462  
filtered: 100.00  
Extra: Using index
```

Ой! Чтение тысяч строк только для того, чтобы получить 200, означает, что мы делаем гораздо больше работы, чем необходимо. Индекс не может уменьшить количество проверяемых строк для запроса, подобного этому, потому что нет предложения `WHERE` для исключения строк.

К сожалению, MySQL ничего не говорит о том, какое количество проанализированных строк было использовано для построения результирующего набора, сообщается лишь, сколько всего строк было проанализировано. Возможно, многие из них оказались отброшены условием `WHERE` и в конечном итоге не будут участвовать в результирующем наборе.

Если в предыдущем примере удалить индекс по столбцу `sakila.film_actor`, запрос будет обращаться к каждой строке таблицы, но из-за условия `WHERE` будут отброшены все строки, кроме десяти. Только оставшиеся десять строк будут использованы для построения результирующего набора. Чтобы понять, к какому числу строк обращается сервер и сколько из них действительно нужны, следует внимательно проанализировать запрос.

Если вы обнаружите, что для получения сравнительно небольшого результирующего набора пришлось проанализировать огромное количество строк, можете попробовать применить некоторые хитрости.

- Воспользуйтесь покрывающими индексами, в которых данные хранятся таким образом, что подсистема хранения вообще не должна извлекать полные строки (мы обсуждали это в главе 7).
- Измените схему. В качестве примера можно привести сводные таблицы, обсуждаемые в главе 6.
- Перепишите сложный запрос так, чтобы оптимизатор MySQL мог выполнить его наиболее оптимально (обсудим это позже в данной главе).

Способы реструктуризации запросов

При оптимизации проблемных запросов ваша цель должна состоять в том, чтобы найти альтернативные способы получения желаемого результата, хотя далеко не всегда это означает, что вы получите точно такой же результирующий набор от MySQL. Иногда можно преобразовывать запросы в эквивалентные формы, которые

возвращают те же результаты и обеспечивают более высокую производительность. Однако вам также следует подумать о переписывании запроса для получения других результатов, если это обеспечивает повышение скорости выполнения. Можно даже изменить не только запрос, но и код приложения. В этом разделе мы рассмотрим различные приемы, которые могут помочь реструктурировать широкий спектр запросов, и покажем, когда использовать каждый из них.

Один сложный или несколько простых запросов

При разработке запросов часто приходится отвечать на важный вопрос: не предпочтительнее ли разбить сложный запрос на несколько более простых? Традиционный подход к проектированию баз данных состоит в попытках выполнить как можно больший объем работы с использованием наименьшего количества запросов. Исторически такой подход был оправдан из-за высоких затрат на сетевые коммуникации и значительных накладных расходов на этапы синтаксического анализа и оптимизации запросов.

Однако к MySQL данная рекомендация относится в меньшей степени, поскольку она изначально проектировалась так, чтобы установление и разрыв соединения происходили максимально эффективно, а обработка небольших простых запросов выполнялась очень быстро. Современные сети также значительно быстрее, чем раньше, поэтому и сетевые задержки заметно сократились. В зависимости от версии сервера MySQL способна выполнять более 100 000 простых запросов в секунду на типичном серверном оборудовании и более 2000 запросов в секунду от одиночного клиента в гигабитной сети, поэтому выполнение нескольких запросов не обязательно является такой уж плохой альтернативой.

Хотя передача информации с использованием соединения все же происходит значительно медленнее по сравнению с тем, какой объем находящихся в памяти данных сама MySQL может перебрать в секунду, это количество измеряется миллионами строк. При прочих равных условиях все равно рекомендуется ограничиться минимальным количеством запросов, но иногда можно повысить скорость выполнения сложного запроса, разложив его на несколько более простых. Не бойтесь делать это. Взвесьте затраты и придерживайтесь стратегии, которая уменьшает общий объем работы. Чуть позже в этой главе мы покажем несколько примеров применения такой методики.

Несмотря на все сказанное, использование чрезмерно большого количества запросов — одна из наиболее распространенных ошибок при проектировании приложений. Например, в некоторых приложениях выполняется десять запросов, возвращающих по одной строке, вместо одного запроса, отбирающего десять 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 количество строк.

Отличный пример — удаление старых данных. В процессе периодической чистки иногда приходится удалять значительные объемы информации, и если делать это одним большим запросом, то возможны блокировка большого числа строк на длительное время, переполнение журналов транзакций, истощение ресурсов, блокировка небольших, не допускающих прерывания запросов. Разбив команду DELETE на части и используя запросы среднего размера, мы существенно повысим производительность и уменьшим отставания реплики в случае репликации запроса. Например, вместо выполнения монолитного запроса

```
DELETE FROM messages
WHERE created < DATE_SUB(NOW(),INTERVAL 3 MONTH);
```

вы могли бы выполнить следующие описанные псевдокодом действия:

```
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 10 000 строк за раз — слишком объемная операция, чтобы быть эффективной, и вместе с тем достаточно короткая, чтобы минимизировать негативное воздействие на сервер¹ (транзакционные 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 сократить время удерживания блокировок.

Декомпозиция соединения

Многие высокопроизводительные приложения используют *декомпозицию соединений*. Вы можете декомпонировать соединение, выполнив несколько запросов к одной таблице вместо соединения с несколькими таблицами, а затем выполнив соединение в приложении. Например, вместо одного запроса:

¹ Инструмент pt-archiver из Percona Toolkit делает такую работу простой и безопасной.

```
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 запустить запросы:

```
SELECT * FROM tag WHERE tag='mysql';
SELECT * FROM tag_post WHERE tag_id=1234;
SELECT * FROM post WHERE post.id in (123,456,567,9098,8904);
```

Зачем же вы это сделали? На первый взгляд это выглядит расточительно, поскольку вы просто увеличили количество запросов, не получив ничего взамен. Однако такая реструктуризация на самом деле может дать ощутимый выигрыш в производительности.

- Можно более эффективно реализовать кэширование. Во многих приложениях кэшируются объекты, которые полностью соответствуют таблицам. В данном примере, если объект, для которого поле `tag` равно `mysql`, уже кэширован, приложение может пропустить первый запрос. Если вы найдете сообщения с идентификатором 123, 567 или 9098 в кэше, можете удалить их из списка `IN()`.
- Запросы, обращающиеся только к одной таблице, могут иногда уменьшить количество конфликтов при блокировках.
- Соединение результатов на уровне приложения упрощает масштабирование базы данных за счет размещения таблиц на разных серверах.
- Сами запросы могут стать более эффективными. В данном примере использование списка `IN()` вместо соединения позволяет MySQL более эффективно сортировать идентификаторы строк и извлекать строки более оптимально, чем было бы возможно в процессе соединения.
- Можно сократить число избыточных обращений к строкам. Выполнение соединения в приложении означает, что вы извлекаете каждую строку только один раз, тогда как соединение в запросе, по сути, является денормализацией, в ходе которой обращение к одним и тем же данным может выполняться многократно. По той же причине такая реструктуризация может уменьшить общий сетевой трафик и использование памяти.

В результате выполнение объединений в приложении может быть более эффективным, если вы кэшируете и повторно используете большое количество данных из более ранних запросов, распределяете данные по нескольким серверам, заменяете соединения списками `IN()` в больших таблицах или в соединении несколько раз встречается одна и та же таблица.

Основные принципы выполнения запросов

Если вы хотите получать от своего сервера MySQL максимальную производительность, стоит потратить время на изучение того, как MySQL оптимизирует и выполняет запросы. Разобравшись в этом, вы обнаружите, что большая часть оптимизации запросов основана на следовании четким принципам, и оптимизация запросов станет для вас очень логичным процессом.

Другими словами, пришло время повторить то, что мы обсуждали ранее, — процесс, которому MySQL следует при выполнении запросов. На рис. 8.1 показано, что происходит, когда вы отправляете запрос к MySQL.

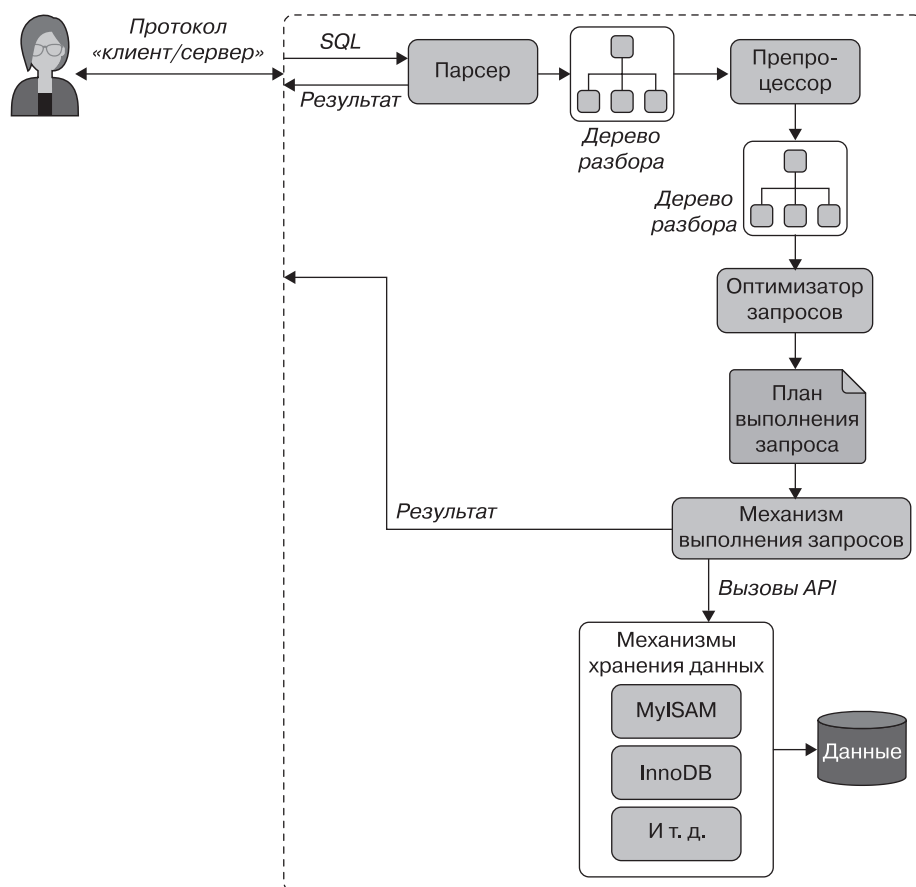


Рис. 8.1. Путь выполнения запроса

- Клиент отправляет SQL-команду на сервер.
- Сервер выполняет разбор, предварительную обработку и оптимизацию SQL-команды, преобразуя ее в план выполнения запроса.
- Подсистема выполнения запросов реализует этот план, вызывая API подсистемы хранения.
- Сервер отправляет результат клиенту.

На каждом из этих шагов есть свои тонкости, которые мы обсудим в следующих разделах. Мы также разберем, в каком состоянии будет находиться запрос на каждом шаге. Процесс оптимизации запросов особенно сложен, и важно разобраться в нем. Существуют также исключения или особые случаи, такие как разница в пути выполнения при использовании подготовленных операторов. Мы обсудим это в следующей главе.

Клиент-серверный протокол MySQL

Вам не обязательно разбираться во внутренних деталях клиент-серверного протокола MySQL, однако необходимо знать, как он работает на высоком уровне. Это полудуплексный протокол, что означает: в любой момент сервер MySQL может либо отправлять, либо получать сообщения, но не то и другое одновременно. Кроме того, это означает, что нет возможности оборвать сообщение на полуслове.

Данный протокол обеспечивает простое и очень быстрое взаимодействие с MySQL, но имеет и кое-какие ограничения. Так, в нем отсутствует механизм управления потоком данных: после того как одна сторона отправила сообщение, другая должна получить его целиком, прежде чем сможет ответить. Это похоже на игру с перебрасыванием мяча туда-сюда: в любой момент времени мяч находится только на одной стороне и вы не можете перекинуть мяч сопернику (отправить сообщение), если его у вас нет.

Клиент отправляет запрос на сервер в виде одного пакета данных. Вот почему конфигурационная переменная `max_allowed_packet` так важна в случаях, когда у вас встречаются большие запросы¹. Как только клиент отправляет запрос, мяч уже на другой стороне и клиенту остается только дожидаться результатов.

Напротив, ответ сервера обычно состоит из множества пакетов данных. Когда сервер отвечает, клиент должен получить весь отправленный им результирующий набор. Нельзя просто получить несколько строк, а затем попросить сервер не отправлять остальные. Если клиенту все-таки нужны только первые несколько

¹ Если запрос слишком длинный, сервер отказывается принимать его целиком и возвращает ошибку.

возвращенных строк, он должен либо дождаться прибытия всех пакетов сервера, а затем отбросить ненужные, либо бесцеремонно разорвать соединение. Оба метода не слишком хороши, поэтому соответствующий раздел `LIMIT` так важен.

Можно было бы подумать, что клиент, извлекающий строки с сервера, *вытаскивает* их. На самом деле это не так: сервер MySQL *выталкивает* строки по мере их создания. Клиент лишь получает вытолкнутые строки, но он не может заставить сервер прекратить их передачу. Он, так сказать, пьет из пожарного шланга (кстати, это технический термин).

Большинство библиотек, которые подключаются к MySQL, позволяют либо получить весь результирующий набор и буферизовать его в памяти, либо получить каждую строку по мере необходимости. Поведение по умолчанию, как правило, заключается в извлечении всего результирующего набора и буферизации его 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 «отправка данных». Когда клиентская библиотека извлекает все результаты сразу, это уменьшает объем работы, которую необходимо выполнить серверу: он может закончить выполнение запроса и освободить ресурсы максимально быстро.

Большинство клиентских библиотек позволяют обрабатывать результирующий набор так, как будто вы получаете его с сервера, хотя на самом деле — из буфера в памяти библиотеки. В большинстве случаев это отлично работает, но это нецелесообразно для огромных наборов результатов, получение которых может потребовать много времени, а для хранения нужно много памяти. Вы можете использовать меньше памяти и быстрее приступить к обработке результата, если зададите библиотеке режим работы без буферизации. Недостатком такого подхода является то, что на сервере останутся открытыми блокировки и другие ресурсы на все время, пока приложение взаимодействует с библиотекой¹.

Рассмотрим пример с использованием PHP. Вот как обычно отправляется запрос к MySQL из PHP:

```
<?php
$link = mysql_connect('localhost', 'user', 'password');
$result = mysql_query('SELECT * FROM HUGE_TABLE', $link);
while ( $row = mysql_fetch_array($result) ) {
    // Делаем что-то с результатом
}
?>
```

¹ Вы можете обойти это ограничение с помощью указания оптимизатору `SQL_BUFFER_RESULT`, которое описывается чуть позже.

Код, кажется, выглядит так, будто строки выбираются по мере необходимости в цикле `while`. Однако фактически весь результирующий набор копируется в буфер с помощью вызова функции `mysql_query()`. В цикле `while` мы просто перебираем данные в буфере. А вот в следующем коде результаты не буферизируются, поскольку используется `mysql_unbuffered_query()` вместо `mysql_query()`:

```
<?php
$link = mysql_connect('localhost', 'user', 'p4ssword');
$result = mysql_unbuffered_query('SELECT * FROM HUGE_TABLE', $link);
while ( $row = mysql_fetch_array($result) ) {
    // Делаем что-то с результатом
}
?>
```

В разных языках программирования приняты разные способы переопределения буферизации. Например, в драйвере `Perl DBD::mysql` необходимо задавать атрибут `mysql_use_result` из клиентской библиотеки `C` (по умолчанию предполагается наличие атрибута `mysql_buffer_result`). Вот пример:

```
#!/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() ) {
    # Делаем что-то с результатом
}
```

Обратите внимание: в вызове метода `prepare()` указан атрибут, говорящий о том, что следует использовать результат вместо его буферизации. Вы можете задать этот атрибут и на этапе установления соединения, что отключит режим буферизации для каждой команды:

```
my $dbh = DBI->connect('DBI:mysql;mysql_use_result=1', 'user', 'p4ssword');
```

Состояния запроса

Каждое соединение или *поток* MySQL имеет состояние, которое показывает, что происходит в любой момент времени. Существует несколько способов узнать эти состояния, самый простой — воспользоваться командой `SHOW FULL PROCESSLIST` (состояния отображаются в столбце `Command`). По мере того как запрос проходит по своему жизненному циклу, его состояние меняется много раз, а всего насчитывается несколько десятков состояний. Руководство по MySQL

является авторитетным источником информации обо всех состояниях, но мы перечислим лишь некоторые из них и объясним, что они означают.

- *Sleep*. Поток ожидает поступления нового запроса от клиента.
- *Query*. Поток либо выполняет запрос, либо отправляет результат клиенту.
- *Locked*. Поток ожидает предоставления табличной блокировки на уровне сервера. Блокировки, реализованные подсистемой хранения, такие как блокировки строк в InnoDB, не вызывают перехода потока в состояние Locked.
- *Analyzing and 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 раздела GROUP BY, файловой сортировки или удовлетворения запроса, содержащего раздел 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.

Очень полезно знать по крайней мере основные состояния, поскольку это позволяет понять, «на чьей стороне мяч» для запроса. На очень сильно загруженных серверах вы можете заметить, как редкое или обычно появляющееся на короткое время состояние, например *statistics*, вдруг начинает занимать значительное количество времени. Чаще всего это указывает на возникновение какой-то аномалии.

Процесс оптимизации запроса

Следующий шаг в жизненном цикле запроса — преобразование SQL-команды в план выполнения, необходимый подсистеме выполнения запросов. Он состоит из нескольких подэтапов: синтаксического анализа, предварительной обработки и оптимизации. Ошибки (например, синтаксические) могут возникнуть в любой момент этого процесса. В нашу задачу не входит строгое документирование внутреннего устройства MySQL, поэтому мы можем позволить себе некоторые вольности, например описывать шаги по отдельности, даже если они часто полностью или частично объединяются для повышения эффективности. Наша цель — просто помочь вам разобраться в том, как MySQL выполняет запросы, чтобы вы могли писать более качественные запросы.

Анализатор и препроцессор

Прежде всего *синтаксический анализатор* MySQL разбивает запрос на лексемы и строит из них дерево разбора. Анализатор использует грамматику SQL, относящуюся к MySQL, для интерпретации и проверки запроса. Например,

он гарантирует, что все лексемы в запросе допустимы и располагаются в правильном порядке, а также проверяет наличие ошибок, таких как непарные кавычки.

Затем *препроцессор* проверяет результирующее дерево синтаксического анализа на наличие дополнительной семантики, не входящей в компетенцию анализатора. Например, он проверяет существование таблиц и столбцов и разрешает имена и псевдонимы, чтобы гарантировать, что ссылки на столбцы не допускают неоднозначного толкования. Далее препроцессор проверяет привилегии. Обычно этот шаг выполняется очень быстро, если только на вашем сервере не определено слишком много привилегий.

Оптимизатор запросов

Теперь дерево синтаксического анализа тщательно проверено и готово для того, чтобы оптимизатор превратил его в план выполнения запроса. Часто запрос можно выполнить разными способами и получить один и тот же результат. Задача оптимизатора — найти наилучший вариант.

MySQL использует оптимизатор на основе затрат, что означает: он пытается предсказать затраты на реализацию различных планов выполнения и выбрать наименее дорогой. В качестве единицы стоимости ранее принимались затраты на считывание случайной страницы данных размером 4 Кбайт, однако сейчас она стала более сложной и включает такие факторы, как оценочная стоимость выполнения сравнения `WHERE`. Вы можете увидеть, как оптимизатор оценил запрос, выполнив его, а затем посмотрев 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 потребуются около 1040 случайных чтений страниц данных. Оценка вычисляется на основе различной статистической информации: количества страниц в таблице или индексе, *кардинальности* (количества различных значений) индексов, длины строк и ключей и распределения ключей. Оптимизатор не учитывает эффекты кэширования любого типа в своих оценках — предполагается, что каждое чтение сводится к операции дискового ввода/вывода.

Оптимизатор не всегда может выбрать наилучший план, и тому есть много причин.

- Некорректная статистика. Сервер полагается на статистическую информацию, получаемую от подсистемы хранения, которая может быть как абсолютно верной, так и не имеющей ничего общего с действительностью. Например, подсистема хранения InnoDB не поддерживает точную статистику о количестве строк в таблице из-за своей архитектуры MVCC.
- Принятый показатель стоимости не всегда эквивалентен истинной стоимости выполнения запроса. Поэтому даже при точной статистике запрос может быть более или менее дорогостоящим, чем можно предположить, исходя из оценки MySQL. План, предусматривающий чтение большего количества страниц, в некоторых случаях может оказаться менее затратным, например, когда операции чтения с диска производятся последовательно или страницы уже кэшированы в памяти. MySQL также не понимает, какие страницы находятся в памяти, а какие — на диске, поэтому она действительно не знает, сколько запросов ввода/вывода потребуется.
- Представление MySQL о том, что такое оптимально, может расходиться с вашим. Вам, вероятно, нужно получить результат как можно быстрее, однако MySQL не пытается выполнять запросы быстрее — она пытается минимизировать затраты на их выполнение, и, как мы видели, определение затрат не является точной наукой.
- MySQL не принимает в расчет конкурирующие запросы, а они могут влиять на скорость выполнения оптимизируемого запроса.
- MySQL не всегда выполняет оптимизацию на основе затрат. Иногда она просто следует правилам, таким как «если в запросе есть полнотекстовое предложение `MATCH()`, то используется индекс `FULLTEXT`, если он существует». Подобное решение будет принято даже тогда, когда быстрее было бы воспользоваться другим индексом и неполнотекстовым запросом с разделом `WHERE`.
- Оптимизатор не учитывает затраты на операции, не находящиеся под его контролем, такие как выполнение хранимых или определенных пользователем функций.
- Как мы увидим позже, оптимизатор не всегда способен оценить каждый возможный план выполнения, поэтому он может просто пропустить оптимальный план.

Оптимизатор запросов MySQL — очень сложное программное обеспечение, использующее множество оптимизаций для преобразования запроса в план выполнения. Существует два основных типа оптимизации — *статическая* и *динамическая*. *Статические оптимизации* можно выполнять просто путем исследования дерева синтаксического анализа. Например, оптимизатор может

преобразовать условие `WHERE` в эквивалентную форму, применяя алгебраические правила. Статические оптимизации не зависят от конкретных значений, таких как значение константы в условии `WHERE`. Будучи выполненной один раз, статическая оптимизация будет действительной всегда, даже если запрос выполняется повторно с другими значениями. Вы можете думать об этом как об оптимизации времени компиляции.

В противоположность этому *динамическая оптимизация* зависит от контекста и может определяться многими факторами, такими как конкретные значения в условии `WHERE` или количество строк в индексе. Их приходится заново вычислять при каждом выполнении запроса. Можно считать, что это «оптимизация времени выполнения».

Данное различие важно при выполнении подготовленных операторов или хранимых процедур. MySQL может выполнить статическую оптимизацию однократно, но динамическую оптимизацию она должна переоценивать каждый раз, когда выполняет запрос. MySQL иногда даже повторно оптимизирует запрос во время его выполнения¹.

Далее перечислены несколько типов оптимизации, которые MySQL умеет выполнять.

- *Изменение порядка соединений.* Таблицы не обязательно нужно соединять в порядке, который указан в запросе. Определение наилучшего порядка соединения — важная оптимизация. Мы подробнее объясним ее позже в этой главе.
- *Преобразование `OUTER JOIN` в `INNER JOIN`.* Оператор `OUTER JOIN` не обязательно выполнять как внешнее соединение. При определенных условиях, зависящих, например, от раздела `WHERE` и схемы таблицы, запрос с `OUTER JOIN` будет фактически эквивалентен `INNER JOIN`. MySQL умеет распознавать и переписывать такие запросы, после чего они могут быть подвергнуты оптимизации типа «изменение порядка соединения».
- *Применение правил алгебраической эквивалентности.* MySQL применяет алгебраические преобразования для упрощения выражений и приведения их к каноническому виду. Кроме того, она умеет вычислять константные выражения, исключая заведомо невыполнимые и всегда выполняющиеся условия. Например, терм $(5=5 \text{ AND } a>5)$ приводится к более простому $a>5$. Аналогично условие $(a<b \text{ AND } b=c) \text{ AND } a=5$ принимает вид $b>5 \text{ AND } b=c \text{ AND } a=5$.

¹ Например, план запроса проверки диапазона повторно оценивает индексы для каждой строки в `JOIN`. Вы можете увидеть этот план запроса, найдя «диапазон проверен для каждой записи» в столбце Extra в `EXPLAIN`. Этот план запроса также увеличивает серверную переменную `Select_full_range_join`.

Эти правила очень полезны для написания условных запросов, которые мы обсудим позже в этой главе.

- *Оптимизация COUNT(), MIN() и MAX().* Индексы и сведения о возможности хранения значений NULL в столбцах часто могут помочь MySQL оптимизировать эти выражения. Например, чтобы найти минимальное значение в столбце, являющемся крайней слева частью индекса, упорядоченного на основе В-дерева, MySQL может просто запросить первую строку из этого индекса. Это можно сделать даже на этапе оптимизации запроса и рассматривать значение как константу для остальной части запроса. Точно так же, чтобы найти максимальное значение в индексе, упорядоченном на основе В-дерева, сервер просто считывает последнюю строку. При такой оптимизации в плане, выведенном командой EXPLAIN, будет присутствовать фраза *Select tables optimized away*. Это буквально означает, что оптимизатор полностью исключил таблицу из плана выполнения запроса, подставив вместо нее константу.
- *Вычисление и свертка константных выражений.* Когда MySQL обнаруживает, что выражение можно преобразовать в константу, она делает это во время оптимизации. Например, определенная пользователем переменная может быть преобразована в константу, если она не изменилась в запросе. Другим примером являются арифметические выражения.

Как ни странно, даже такие вещи, которые вы, скорее всего, назвали бы запросом, можно свернуть в константу на этапе оптимизации. Одним из примеров является вычисление функции MIN() для индекса. Этот пример можно даже расширить до поиска констант по первичному ключу или уникальному индексу. Если в разделе WHERE встречается константное условие для такого индекса, оптимизатор знает, что MySQL может найти значение в самом начале выполнения запроса. Впоследствии найденное значение можно рассматривать как константу в остальной части запроса. 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\G
***** 1. row *****
id: 1
select_type: SIMPLE
table: film
partitions: NULL
type: const
possible_keys: PRIMARY
key: PRIMARY
key_len: 2
ref: const
rows: 1
filtered: 100.00
Extra: Using index
```

```
***** 2. row *****
id: 1
select_type: SIMPLE
table: film_actor
partitions: NULL
type: index
possible_keys: NULL
key: PRIMARY
key_len: 4
ref: NULL
rows: 5462
filtered: 10.00
Extra: Using where; Using index
```

MySQL выполняет этот запрос в два этапа, о чем свидетельствуют две строки в выведенной таблице. На первом этапе она находит нужную строку в таблице `film`. Оптимизатор MySQL знает, что такая строка единственная, так как столбец `film_id` — это первичный ключ, а оптимизатор уже на стадии оптимизации запроса уточнил количество найденных строк. Поскольку оптимизатору известна точная величина (значение в разделе `WHERE`), которая будет возвращена в результате поиска, то в столбце `ref` для этой таблицы стоит `const`.

На втором этапе MySQL считает известной величиной столбец `film_id` из строки, найденной на первом этапе. Она может так поступить, потому что в момент перехода ко второму шагу оптимизатор уже знает все значения, определенные на первом шаге. Обратите внимание на то, что тип `ref` для таблицы `film_actor` равен `const`, как и для таблицы `film`.

Константные условия могут применяться также вследствие распространения константности значения от одного места к другому при наличии разделов `WHERE`, `USING` или `ON` с ограничением типа «равно». В приведенном ранее примере оптимизатор знает: фраза `USING` гарантирует, что значение `film_id` будет одинаково на протяжении всего запроса — оно должно быть равно константе, заданной в разделе `WHERE`.

- *Покрывающие индексы.* Если индекс содержит все столбцы, необходимые для запроса, MySQL может воспользоваться индексом, вообще не читая данные из таблицы. Мы подробно рассматривали покрывающие индексы в предыдущей главе.
- *Оптимизация подзапросов.* MySQL может преобразовывать некоторые типы подзапросов в более эффективные альтернативные формы, сводя их к поиску по индексу.
- *Раннее завершение.* MySQL может прекратить обработку запроса (или какой-то шаг в запросе), как только поймет, что этот запрос или шаг выполнен. Очевидным примером является раздел `LIMIT`, но есть и другие виды досрочного завершения. Например, если MySQL обнаруживает заведомо невыполнимое

условие, она может прекратить обработку всего запроса. Это показано в следующем примере:

```
mysql> EXPLAIN SELECT film.film_id FROM sakila.film WHERE film_id = -1;
***** 1. row *****
id: 1
select_type: SIMPLE
table: NULL
partitions: NULL
type: NULL
possible_keys: NULL
key: NULL
key_len: NULL
ref: NULL
rows: NULL
filtered: NULL
Extra: Impossible WHERE
```

Этот запрос остановлен на этапе оптимизации, но в некоторых других случаях MySQL может досрочно завершить выполнение. Сервер может использовать эту оптимизацию, когда подсистема выполнения запросов распознает необходимость извлечения различных значений, или останавливается, когда значения не существует. Например, следующий запрос находит все фильмы, в которых нет ни одного актера¹:

```
SELECT film.film_id
FROM sakila.film
LEFT OUTER JOIN sakila.film_actor USING(film_id)
WHERE film_actor.film_id IS NULL;
```

Этот запрос исключает все фильмы, в которых есть хотя бы один актер. В каждом фильме может быть задействовано много актеров, но как только сервер находит одного актера, он прекращает обработку текущего фильма и переходит к следующему, поскольку знает, что условию `WHERE` такой фильм заведомо не удовлетворяет. Аналогичная оптимизация «отличается/не существует» может применяться к определенным типам запросов, 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. Например, в запросе:

¹ Мы согласны, что 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».

```
SELECT film.film_id
FROM sakila.film
INNER JOIN sakila.film_actor USING(film_id)
WHERE film.film_id > 500;
```

MySQL понимает, что условие `WHERE` применяется не только к таблице `film`, но и к таблице `film_actor`, поскольку из-за наличия раздела `USING` столбцы должны совпадать.

Если вы привыкли к другому серверу базы данных, в котором такая оптимизация не реализована, вам, возможно, рекомендовали помочь оптимизатору, вручную указав в разделе `WHERE` условия для обеих таблиц, например:

```
... WHERE film.film_id > 500 AND film_actor.film_id > 500
```

В MySQL это не обязательно. Это просто усложняет поддержку ваших запросов.

- *Сравнение по списку `IN()`*. Во многих серверах баз данных `IN()` является не более чем синонимом нескольких условий `OR`, поскольку они логически эквивалентны. Но это не так в MySQL, которая сортирует значения в списке `IN()` и применяет для работы с ним быстрый двоичный поиск, чтобы увидеть, находится ли значение в списке. Вычислительная сложность при этом составляет $O(\log n)$, где n — размер списка, тогда как сложность эквивалентной последовательности условий `OR` составляет $O(n)$, то есть поиск выполняется намного медленнее для больших списков.

Приведенный перечень, естественно, неполон, поскольку MySQL умеет выполнять больше оптимизаций, чем поместилось бы во всей этой главе, но он должен дать вам представление о сложности и развитых логических возможностях оптимизатора. Главная мысль, которую следует вынести из этого обсуждения: *не пытайтесь перехитрить оптимизатор*. В итоге вы просто потерпите неудачу или сделаете свои запросы чрезмерно запутанными и сложными для сопровождения, не получив ни малейшей выгоды. В общем, вы должны позволить оптимизатору сделать свою работу.

Конечно, каким бы умным ни был оптимизатор, бывают случаи, когда он не находит наилучшего результата. Иногда вы можете знать о своих данных что-то такое, чего не знает оптимизатор, например некое условие, которое гарантированно истинно вследствие логики приложения. Кроме того, иногда оптимизатор не обладает необходимой функциональностью, например хеш-индексами, а в других случаях, как упоминалось ранее, его алгоритм оценки стоимости может предпочесть план запроса, который окажется более затратным, чем возможная альтернатива.

Если вы знаете, что оптимизатор не дает хорошего результата, и понимаете, почему так произошло, вы можете ему помочь. Некоторые из вариантов включают добавление подсказки к запросу¹, переписывание запроса, изменение схемы или добавление индексов.

Статистика по таблицам и индексам

Вспомним различные архитектурные уровни сервера MySQL, которые мы проиллюстрировали на рис. 1.1. На уровне сервера, где расположен оптимизатор запросов, отсутствует статистика по данным и индексам. Работа по ее обеспечению возлагается на подсистемы хранения, поскольку каждая из них может собирать различную статистическую информацию или хранить ее разными способами.

Поскольку сервер не содержит статистики, оптимизатор запросов MySQL должен обращаться к подсистемам хранения за статистическими данными по участвующим в запросе таблицам. Подсистема может предоставить оптимизатору такие статистические данные, как количество страниц в таблице или индексе, кардинальность таблиц и индексов, длина строк и ключей и сведения о распределении ключей. Оптимизатор может использовать эту информацию, чтобы выбрать наилучший план выполнения. В последующих разделах мы увидим, как статистические данные влияют на решения оптимизатора.

Стратегия выполнения соединений в MySQL

В MySQL термин «соединение» используется более широко, чем вы, возможно, привыкли. Под соединениями понимаются все запросы, а не только те, что сопоставляют строки из двух таблиц. Еще раз подчеркнем, что имеются в виду все без исключения запросы, включая подзапросы и даже выборку `SELECT` для одной таблицы. Следовательно, очень важно понимать, как MySQL выполняет соединения.

Рассмотрим пример запроса `UNION`. MySQL выполняет `UNION` в виде последовательности одиночных запросов, результаты которых помещаются во временную таблицу, а затем снова считываются. Каждый из отдельных запросов представляет собой соединение в терминологии MySQL, и им же является операция чтения из результирующей временной таблицы.

Стратегия выполнения соединений в MySQL раньше была простой: все соединения реализуются алгоритмом вложенных циклов. Это означает, что MySQL в цикле перебирает строки из одной таблицы, а затем запускает вложенный

¹ См. разделы «Подсказки индекса» и «Подсказки оптимизатора» в руководстве по MySQL, чтобы узнать, какие подсказки доступны для конкретной версии и как их использовать.

цикл, чтобы найти соответствующую строку в следующей таблице. Так продолжается до тех пор, пока не будут найдены соответствующие строки во всех таблицах соединения. После этого строится и возвращается строка, составленная из перечисленных в списке `SELECT` столбцов. Далее MySQL пытается построить следующую строку в последней таблице. Если такой не оказывается, то происходит возврат на одну таблицу назад и делается попытка найти дополнительные строки в ней. MySQL продолжает выполнять возвраты, пока не найдет другую строку в какой-либо таблице, после чего ищет совпадающую строку в следующей таблице и т. д.¹

Начиная с версии 8.0.20, блочные соединения с вложенными циклами больше не используются — их заменило хеш-соединение. Это позволяет соединению выполняться так же быстро, как раньше, или еще быстрее, особенно если один из наборов данных может храниться в памяти.

План выполнения

MySQL не генерирует байт-код для выполнения запроса, как делают многие другие продукты баз данных. Вместо этого план выполнения запроса фактически представляет собой дерево² инструкций, которые выполняются подсистемой выполнения запроса для получения результатов запроса. Окончательный план содержит достаточно информации, позволяющей реконструировать исходный запрос. Если вы выполните команду `EXPLAIN EXTENDED` для запроса, а затем команду `SHOW WARNINGS`, то увидите реконструированный запрос³. Любой запрос с несколькими таблицами концептуально может быть представлен в виде дерева. Например, можно выполнить соединение четырех таблиц, как показано на рис. 8.2.

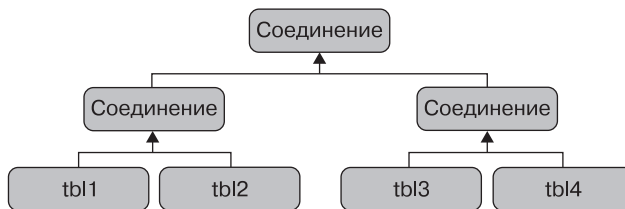


Рис. 8.2. Один из способов соединения нескольких таблиц

¹ Далее мы покажем, что на самом деле выполнение запроса не такое простое дело: существует немало оптимизаций, усложняющих алгоритм.

² Вы можете увидеть это, используя `EXPLAIN FORMAT=TREE...` перед своим выражением.

³ Сервер генерирует вывод из плана выполнения. Таким образом, он имеет ту же семантику, что и исходный запрос, но не обязательно тот же самый текст.

Это то, что специалисты по информатике называют *сбалансированным деревом*. Однако MySQL выполняет запрос иначе. Как мы описали в предыдущем разделе, MySQL всегда начинает с одной таблицы и находит соответствующие строки в следующей. Таким образом, планы выполнения запросов MySQL всегда имеют вид *дерева с левой глубиной (левоглубинного)* (left-deep tree), как показано на рис. 8.3.

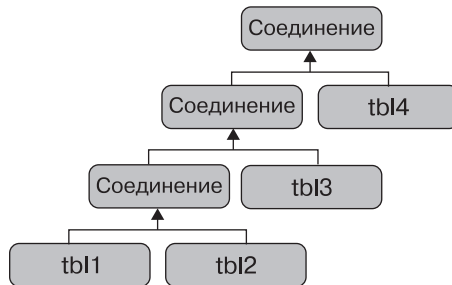


Рис. 8.3. Как MySQL соединяет несколько таблиц

Оптимизатор соединения

Наиболее важной частью оптимизатора запросов MySQL является *оптимизатор соединения*, который определяет наилучший порядок выполнения многотабличных запросов. Часто можно соединить таблицы в разном порядке и получить одинаковые результаты. Оптимизатор соединения вычисляет затраты на реализацию различных планов и пытается выбрать наименее затратный.

Приведем пример запроса, в котором таблицы можно соединять в разном порядке без изменения результатов:

```
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);
```

Скорее всего, вы можете придумать несколько разных планов выполнения запросов. Например, MySQL может начать с таблицы `film`, воспользоваться индексом таблицы `film_actor` по полю `film_id` для поиска значений `actor_id`, а затем искать строки по первичному ключу таблицы `actor`. Пользователи Oracle могут сформулировать это следующим образом: «Таблица `film` — это ведущая таблица для таблицы `film_actor`, которая является ведущей для таблицы `actor`». 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
partitions: NULL
type: ALL
possible_keys: PRIMARY
key: NULL
key_len: NULL
ref: NULL
rows: 200
filtered: 100.00
Extra: NULL
***** 2. row *****
id: 1
select_type: SIMPLE
table: film_actor
partitions: NULL
type: ref
possible_keys: PRIMARY,idx_fk_film_id
key: PRIMARY
key_len: 2
ref: sakila.actor.actor_id
rows: 27
filtered: 100.00
Extra: Using index
***** 3. row *****
id: 1
select_type: SIMPLE
table: film
partitions: NULL
type: eq_ref
possible_keys: PRIMARY
key: PRIMARY
key_len: 2
ref: sakila.film_actor.film_id
rows: 1
filtered: 100.00
Extra: NULL

```

Этот план существенно отличается от предложенного в предыдущем абзаце. MySQL хочет начать с таблицы **actor** (мы это знаем, потому что она указана первой в списке, выданном командой **EXPLAIN**) и двигаться в обратном порядке. Действительно ли это более эффективно? Попробуем разобраться. 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
partitions: NULL
type: ALL
possible_keys: PRIMARY
key: NULL
key_len: NULL
ref: NULL
rows: 1000
filtered: 100.00
Extra: NULL
***** 2. row *****
id: 1
select_type: SIMPLE
table: film_actor
partitions: NULL
type: ref
possible_keys: PRIMARY,idx_fk_film_id
key: idx_fk_film_id
key_len: 2
ref: sakila.film.film_id
rows: 5
filtered: 100.00
Extra: Using index
***** 3. row *****
id: 1
select_type: SIMPLE
table: actor
partitions: NULL
type: eq_ref
possible_keys: PRIMARY
key: PRIMARY
key_len: 2
ref: sakila.film_actor.actor_id
rows: 1
filtered: 100.00
Extra: NULL

```

Здесь видно, почему MySQL предпочла обратный порядок соединения: это позволит ей проверять меньше строк в первой таблице¹. В обоих случаях она сможет выполнять быстрый поиск по индексу во второй и третьей таблицах. Разница лишь в том, сколько таких операций поиска по индексу придется выполнить. Если начать с таблицы `film`, потребуется около 1000 обращений 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. Другими словами, измененный порядок соединений сокращает количество возвратов и повторных операций чтения.

¹ Строго говоря, MySQL не пытается уменьшить количество считываемых строк. Вместо этого он пытается оптимизировать работу за счет меньшего количества чтений страниц. Но подсчет строк может дать вам приблизительное представление о стоимости запроса.

Это простой пример того, как оптимизатор соединений MySQL, изменяя порядок выполнения запросов, может сделать его менее затратным. Изменение порядка соединений обычно оказывается очень эффективной оптимизацией. Однако бывает, что это не приводит к оптимальному плану, и в таких случаях вы можете использовать ключевое слово `STRAIGHT_JOIN` и написать запрос в том порядке, который вам представляется лучшим. Однако так происходит редко — в большинстве случаев оптимизатор соединения оказывается эффективнее человека.

Оптимизатор соединения пытается построить дерево плана выполнения запроса с минимальными затратами. Когда это возможно, он исследует все потенциальные комбинации поддеревьев, начиная со всех однотабличных планов.

К сожалению, для полного исследования соединения n таблиц нужно перебрать n -факториал возможных перестановок. Это называется пространством поиска всех возможных планов выполнения запросов. Его размер растет очень быстро: соединение десяти таблиц может быть выполнено 3 628 800 различными способами! Когда пространство поиска становится слишком большим, оптимизация запроса может занять слишком много времени, поэтому сервер отказывается от полного анализа. Если количество соединяемых таблиц превышает значение параметра `optimizer_search_depth` (которое вы при желании можете изменить), то сервер применяет сокращенные алгоритмы, например жадный поиск.

Для ускорения этапа оптимизации в MySQL реализовано множество эвристических приемов, выработанных за годы исследований и экспериментов. Это может быть полезно, но также может означать, что MySQL способна (в редких случаях) пропустить оптимальный план и выбрать менее оптимальный, потому что она не пыталась исследовать каждый возможный план запроса.

Иногда запросы нельзя выполнить в другом порядке, и оптимизатор соединения может использовать этот факт, чтобы уменьшить пространство поиска, исключив из него некоторые перестановки. Хорошим примером могут служить запросы с `LEFT JOIN`, как и коррелированные подзапросы (подробнее о подзапросах позже). Объясняется это тем, что результаты для одной таблицы зависят от данных, полученных из другой таблицы. Наличие таких зависимостей позволяет оптимизатору соединения сократить пространство поиска, исключая варианты выбора.

Оптимизация сортировки

Сортировка результатов может оказаться весьма затратной операцией, поэтому часто можно повысить производительность, исключив сортировку или выполняя ее для меньшего количества строк.

Когда MySQL не может использовать индекс для получения отсортированного результата, ей приходится сортировать строки самостоятельно. Это можно

сделать в памяти или на диске, но сама процедура всегда называется файловой сортировкой, даже если она на самом деле не использует файл.

Если обрабатываемые значения помещаются в буфер сортировки, MySQL может выполнить сортировку полностью в памяти, применяя алгоритм быстрой сортировки. Если MySQL не может выполнить сортировку в памяти, она реализуется на диске поблочно. Каждый блок обрабатывается методом быстрой сортировки, а затем отсортированные блоки сливаются.

Существует два алгоритма файловой сортировки:

- *двухпроходный* (старый). Считывает указатели строк и столбцы, упомянутые в разделе **ORDER BY**, сортирует их, а затем просматривает отсортированный список и повторно читает исходные строки, чтобы вывести результат.

Двухпроходный алгоритм может быть довольно затратным, поскольку строки из таблицы прочитываются дважды и повторное чтение вызывает много произвольных операций ввода/вывода;

- *однопроходный* (новый). Читает все столбцы, необходимые для запроса, сортирует их по столбцам, упомянутым в разделе **ORDER BY**, а затем просматривает отсортированный список и выводит указанные столбцы.

Этот алгоритм может быть намного эффективнее старого, особенно для больших наборов данных, связанных с большой нагрузкой на ввод/вывод. Однопроходный алгоритм не читает строки из таблицы дважды, а произвольный ввод/вывод в нем заменяется последовательным чтением. Однако ему может потребоваться гораздо больше памяти, поскольку для каждой строки приходится хранить все запрошенные столбцы, а не только те, что необходимы для сортировки строк. Следовательно, в буфер сортировки поместится меньше строк и при сортировке файлов придется выполнить больше циклов слияния.

При файловой сортировке серверу MySQL может потребоваться гораздо больше места на диске для хранения временных файлов, чем вы ожидаете, потому что каждому сортируемому кортежу выделяется запись фиксированной длины. Эти записи достаточно велики, чтобы сохранить максимально длинный кортеж, в котором для столбцов типа **VARCHAR** зарезервировано место в соответствии с указанным в схеме размером. Кроме того, если вы используете кодировку **utf8mb4**, MySQL выделяет 4 байта для каждого символа. Так что нам доводилось сталкиваться с тем, что из-за плохо оптимизированных схем 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 в разделе **ORDER BY** упомянуты только столбцы из первой (в порядке соединения) таблицы, MySQL может выполнить ее файловую сортировку, а затем приступить к соединению. В таком случае команда

EXPLAIN поместит в столбец Extra фразу `Using filesort`. Во всех других случаях — например, сортируемая таблица не первая по порядку или раздел `ORDER BY` содержит столбцы более чем из одной таблицы — MySQL должна сохранить результаты запроса во временную таблицу, а выполнять файловую сортировку временной таблицы после завершения соединения. В этом случае команда EXPLAIN поместит в столбец Extra фразу `Using temporary; Using filesort`. Если задано ключевое слово `LIMIT`, то оно применяется после сортировки, поэтому временная таблица и файловая сортировка могут быть очень большими.

Подсистема выполнения запросов

На стадиях разбора и оптимизации вырабатывается план выполнения запроса, который подсистема выполнения запросов MySQL использует для обработки запроса. План представляет собой структуру данных, а не исполняемый байт-код, как во многих других базах данных.

В отличие от стадии оптимизации стадия выполнения обычно не так уж сложна: MySQL просто следует инструкциям, содержащимся в плане выполнения запроса. Для многих указанных в плане операций нужно вызывать методы, реализованные интерфейсом подсистемы хранения, также известным как *API обработчика*. Каждая таблица в запросе представлена экземпляром обработчика. Например, если таблица встречается в запросе три раза, сервер сформирует три экземпляра обработчика. Хотя раньше мы об этом не упоминали, MySQL фактически создает экземпляры обработчика на ранних стадиях оптимизации. Оптимизатор использует их для получения информации о таблицах, такой как имена их столбцов и статистика индексов.

Интерфейс подсистемы хранения имеет множество функциональных возможностей, но для выполнения большинства запросов ему требуется всего около дюжины основных операций — строительных блоков. Например, имеются операции чтения первой и последующей строк индекса. Этого вполне достаточно для запроса, в котором просматривается индекс. Благодаря такому упрощенному подходу к выполнению запроса и стало возможным использование архитектуры подсистемы хранения в MySQL, но это, в свою очередь, накладывает определенные ограничения на оптимизатор, которые мы обсуждали ранее.



Не все, что включено в план выполнения запроса, — это операции обработчика. Например, сервер управляет табличными блокировками. Обработчик может реализовать собственную схему блокировки на нижнем уровне строк, как это делает InnoDB со строковыми блокировками (однако она не заменяет реализацию блокировок внутри сервера). Как отмечалось в главе 1, на сервере реализованы функции, общие для всех подсистем хранения, например функции работы с датой и временем, представления и триггеры.

Для выполнения запроса сервер просто повторяет инструкции до тех пор, пока не будут проанализированы все строки.

Возврат результатов клиенту

Последний шаг при выполнении запроса — отправка ответа клиенту. Даже запросы, которые не возвращают результирующий набор, все равно посылают клиенту информацию о результате обработки, например, о том, сколько строк он затронул.

Сервер генерирует и отправляет результаты поэтапно. Как только MySQL обрабатывает последнюю таблицу и успешно сгенерирует очередную строку, она может и должна отправить ее клиенту. У такого решения есть два преимущества: серверу не обязательно хранить строку в памяти, а клиент начинает получать результаты настолько быстро, насколько это вообще возможно. Каждая строка в результирующем наборе отправляется в отдельном пакете по клиент-серверному¹ протоколу MySQL, хотя пакеты протокола могут буферизоваться и отправляться вместе на уровне протокола TCP.

Ограничения оптимизатора запросов MySQL

Подход MySQL к выполнению запросов неидеален для оптимизации всех видов запросов. К счастью, существует лишь ограниченное число случаев, с которыми оптимизатор MySQL справляется заведомо плохо, и обычно такие запросы удастся переписать более эффективно.

Ограничения UNION

Иногда MySQL не может опустить условия с внешнего уровня UNION на внутренний, где их можно использовать с целью ограничения результатов или создания возможностей для дополнительной оптимизации.

Если вы считаете, что какой-либо из отдельных запросов, составляющих UNION, будет выполняться быстрее при наличии LIMIT, или знаете, что после объединения с другими запросами результаты будут отсортированы по условию ORDER BY, то имеет смысл поместить соответствующие ключевые слова в каждую часть

¹ При необходимости можете изменить это поведение, например, с помощью подсказки SQL_BUFFER_RESULT. Дополнительную информацию см. в разделе «Подсказки оптимизатора» в официальном руководстве MySQL.

UNION. Например, в ситуации, когда вы объединяете две очень большие таблицы и ограничиваете результат первыми 20 строками, MySQL сохранит обе таблицы во временной таблице, а затем извлечет из нее только 20 строк:

```
(SELECT first_name, last_name
  FROM sakila.actor
  ORDER BY last_name)
UNION ALL
(SELECT first_name, last_name
  FROM sakila.customer
  ORDER BY last_name)
LIMIT 20;
```

Этот запрос сохранит 200 строк из таблицы актеров и 599 из таблицы клиентов во временную таблицу, а затем извлечет из нее первые 20 строк. Вы можете избежать этого, добавляя **LIMIT 20** к каждому запросу внутри **UNION**:

```
(SELECT first_name, last_name
  FROM sakila.actor
  ORDER BY last_name
  LIMIT 20)
UNION ALL
(SELECT first_name, last_name
  FROM sakila.customer
  ORDER BY last_name
  LIMIT 20)
LIMIT 20;
```

Теперь во временной таблице будут храниться только 40 строк. Помимо улучшения производительности, вам, вероятно, придется исправить запрос: порядок, в котором строки извлекаются из временной таблицы, не определен, поэтому перед последним ключевым словом **LIMIT** должен быть общий **ORDER BY**.

Распространение равенства

Распространение равенства иногда может вызвать непредвиденные затраты. Например, рассмотрите гигантский список **IN()** для какого-то столбца. Оптимизатору известно, что он равен другим столбцам в других таблицах благодаря наличию ключевых слов **WHERE**, **ON** или **USING**, которые устанавливают равенство столбцов друг другу.

Оптимизатор прибегнет к разделению списка, скопировав его во все соответствующие столбцы всех соединенных таблиц. Обычно это полезно, потому что дает оптимизатору запросов и механизму выполнения больше возможностей определить, где именно проверять сравнение со списком **IN()**. Но когда список

очень большой, то и оптимизация, и выполнение могут оказаться медленными. На момент написания этой книги не существует встроенного обходного пути для этой проблемы, и если вы с ней столкнетесь, вам придется изменить исходный код (это не проблема для большинства разработчиков).

Параллельное выполнение

MySQL не умеет распараллеливать выполнение одного запроса на нескольких процессорах. Эта возможность предлагается некоторыми другими серверами баз данных, но только не MySQL. Мы упомянули об этом, чтобы вы не тратили много времени на то, чтобы понять, как добиться параллельного выполнения запросов на MySQL!

SELECT и UPDATE для одной и той же таблицы

MySQL не позволяет вам выполнять команду `SELECT` из таблицы, одновременно выполняя для нее команду `UPDATE`. На самом деле это не ограничение оптимизатора, но знание того, как MySQL выполняет запросы, может помочь вам обойти это ограничение. Далее приведен пример запроса, написанного в полном соответствии со стандартом SQL, но в MySQL недопустимого. Этот запрос обновляет каждую строку, записывая в нее количество похожих строк, имеющих в той же таблице:

```
mysql> UPDATE tbl AS outer_tbl
-> SET c = (
-> 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, можете использовать производную таблицу, поскольку MySQL материализует ее в виде временной таблицы. В результате выполняются эффективно два запроса: один `SELECT` внутри подзапроса и один многотабличный `UPDATE` с соединенными результатами таблицы и подзапроса. Подзапрос открывает и закрывает таблицу перед тем, как внешний `UPDATE` открывает ее, поэтому теперь запрос будет выполнен успешно:

```
mysql> UPDATE tbl
-> INNER JOIN(
-> SELECT type, count(*) AS c
-> FROM tbl
-> GROUP BY type
-> ) AS der USING(type)
-> SET tbl.c = der.c;
```

Оптимизация конкретных типов запросов

В этом разделе дадим советы по оптимизации определенных типов запросов. Большинство из этих тем мы подробно рассмотрели в других разделах книги, но хотим представить общий перечень типичных проблем оптимизации, чтобы потом на него можно было ссылаться.

Большинство советов в этом разделе зависят от версии, и, возможно, их нельзя будет применить для будущих версий MySQL. Нет никаких причин, по которым когда-нибудь сервер не сможет самостоятельно выполнить некоторые из этих оптимизаций или их все.

Оптимизация запросов COUNT()

Агрегатная функция `COUNT()` и способы оптимизации запросов, которые ее используют, — вероятно, одна из десяти самых неправильно понятых тем в MySQL. Вы можете выполнить поиск в Интернете и найти больше дезинформации по этой теме, чем нам хотелось бы думать.

Прежде чем мы перейдем к оптимизации, важно, чтобы вы поняли, что на самом деле делает функция `COUNT()`.

Что делает COUNT()

`COUNT()` — это специальная функция, которая решает две очень разные задачи: она подсчитывает значения и строки. Значение — это выражение, отличное от `NULL` (`NULL` — это отсутствие какого бы то ни было значения). Если вы указываете имя столбца или другое выражение в скобках, `COUNT()` подсчитывает, сколько раз это выражение имеет значение. Многих это сбивает с толку, 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 не обязательно является хорошим источником точной информации по этой теме.)

Другая форма `COUNT()` просто подсчитывает количество строк в результирующем наборе. Так MySQL поступает, когда точно знает, что выражение в круглых скобках никогда не может быть равно `NULL`. Наиболее очевидный пример — выражение `COUNT(*)` — особая форма `COUNT()`, которая вовсе не сводится к подстановке вместо специального символа `*` полного списка столбцов в таблице, как можно было бы ожидать. На самом деле столбцы вообще игнорируются, а подсчитываются сами строки.

Одна из наиболее распространенных ошибок, которую мы наблюдали, — это задание имени столбца в скобках, когда нужно подсчитать строки. Если вы хотите узнать количество строк в результирующем наборе, то всегда должны использовать `COUNT(*)`. Тем самым вы недвусмысленно сообщите о своем намерении и избежите низкой производительности.

Простые оптимизации

Часто спрашивают, как с помощью всего одного запроса подсчитать, сколько раз встречаются несколько разных значений в одном столбце. Предположим, что вы хотите написать один запрос, который подсчитывает количество элементов разных цветов. Использовать операцию `OR` (например, `SELECT COUNT(color = 'blue' OR color = 'red') FROM items;`) нельзя, потому что невозможно будет разделить счетчики разных цветов. Вы также не можете поместить цвета в предложение `WHERE` (например, `SELECT COUNT(*) FROM items WHERE color = 'blue' AND color = 'red';`), поскольку они являются взаимоисключающими. Тем не менее задачу можно решить с помощью такого запроса¹:

```
SELECT SUM(IF(color = 'blue', 1, 0)) AS blue, SUM(IF(color = 'red', 1, 0))
AS red FROM items;
```

А вот еще один эквивалент, в котором вместо функции `SUM()` используется `COUNT()`. Он основан на том, что выражение не будет иметь значения, когда условие ложно:

```
SELECT COUNT(color = 'blue' OR NULL) AS blue, COUNT(color = 'red' OR NULL)
AS red FROM items;
```

Использование приближенных значений

Иногда вам не нужен точный подсчет, поэтому можете задействовать приближенные значения. Оценка количества строк, выполненная оптимизатором в команде `EXPLAIN`, как правило, хорошо подходит для этих целей. Просто добавьте к запросу команду `EXPLAIN`.

В других случаях точное количество даже менее эффективно, чем приближенное. Один клиент попросил помощи в подсчете количества активных пользователей на своем сайте. Счетчик пользователей кэшировался и отображался в течение 30 мин, после чего обновлялся и снова кэшировался. Тем самым результат, по сути, был неточным. Поэтому приближенное значение было вполне приемлемо. Запрос включал несколько условий `WHERE`, чтобы гарантированно не учитывать

¹ Вы также можете написать выражения `SUM()` как `SUM(color = 'blue')`, `SUM(color = 'red')`.

неактивных пользователей и пользователя «по умолчанию», у которого в приложении был специальный идентификатор пользователя. Удаление этих условий незначительно изменило полученную величину, но сделало запрос гораздо более эффективным. Дальнейшая оптимизация заключалась в устранении ненужного ключевого слова `DISTINCT`, что позволило избежать файловой сортировки. Переписанный запрос был намного быстрее и возвращал почти те же результаты.

Более сложные оптимизации

В общем случае запросы, содержащие `COUNT()`, с трудом поддаются оптимизации, поскольку обычно они должны подсчитать много строк, то есть получить доступ к большому количеству данных. Единственной возможностью оптимизации внутри самой MySQL является использование покрывающего индекса. Если этого недостаточно, вам нужно внести изменения в архитектуру своего приложения. Рассмотрим внешнюю систему кэширования, такую как `memcached`. Вероятно, вам придется столкнуться с известной дилеммой: «Быстро, точно и просто — выберите любые два».

Оптимизация запросов с JOIN

Эта тема появляется на протяжении большей части книги, но сейчас мы упомянем несколько основных моментов.

- Убедитесь, что по столбцам, используемым в разделах `ON` или `USING`, построены индексы. При добавлении индексов учитывайте порядок соединения. Если таблицы `A` и `B` соединяются по столбцу `C`, а оптимизатор запросов решит, что их надо соединять в порядке `B, A`, вам не нужно индексировать столбец в таблице `B`. Неиспользуемые индексы — это дополнительные накладные расходы. Как правило, следует индексировать только вторую таблицу в порядке соединения, если, конечно, индекс не нужен по какой-либо другой причине.
- Постарайтесь убедиться, что в выражениях `GROUP BY` или `ORDER BY` встречаются столбцы только из одной таблицы, тогда у MySQL появится возможность воспользоваться для этой операции индексом.
- Будьте внимательны при переходе на новую версию MySQL, поскольку синтаксис соединения, приоритет операторов и другие параметры поведения менялись в разное время. Конструкция, которая раньше была обычным соединением, вдруг становится декартовым произведением, то есть совершенно другим типом соединения, возвращающим другие результаты, а то и вовсе оказывается синтаксически некорректной.

Оптимизация GROUP BY WITH ROLLUP

Разновидностью группирующих запросов является суперагрегирование результатов. Вы можете сделать это с помощью указания `WITH ROLLUP`, но ее оптимизация может оставлять желать лучшего. Проверьте, как выполняется запрос с помощью команды `EXPLAIN`, обращая внимание на то, как производится группировка — посредством файловой сортировки или созданием временной таблицы: попробуйте удалить `WITH ROLLUP` и посмотрите, получите ли вы тот же способ группировки. Возможно, вам придется принудительно указать метод группировки с помощью подсказок, которые мы упоминали ранее в этом разделе.

Иногда оказывается эффективнее выполнить суперагрегирование в самом приложении, даже если для этого придется запросить у сервера больше строк. Можно также воспользоваться вложенным подзапросом в условии `FROM` или задействовать временную таблицу для хранения промежуточных результатов, а затем выполнить запрос к ней с ключевым словом `UNION`.

Наилучшим подходом станет тот, в котором функциональность `WITH ROLLUP` будет отдана на откуп приложению.

Оптимизация LIMIT и OFFSET

Запросы, содержащие ключевые слова `LIMIT` и `OFFSET`, часто встречаются в системах, выполняющих разбиение на страницы, и почти всегда в сочетании с предложением `ORDER BY`. Полезно иметь индекс, который поддерживает нужное упорядочение, в противном случае серверу приходится слишком часто прибегать к файловой сортировке.

Типичной проблемой является слишком большое значение смещения. Если в вашем запросе встречается фраза `LIMIT 10000, 20`, то сервер генерирует 10 020 строк и отбрасывает первые 10 000 из них, что очень затратно. Если предположить, что доступ ко всем страницам выполняется с одинаковой частотой, такой запрос просматривает в среднем половину таблицы. Для оптимизации можно либо наложить ограничения на то, сколько страниц разрешено просматривать, либо попытаться реализовать обработку больших смещений более эффективно.

Один из простых методов повышения производительности — выполнение смещения с использованием покрывающего индекса, а не по полным строкам исходной таблицы. Затем можете соединить полученные результаты с полной строкой и получить дополнительные столбцы, которые вам нужны. Такой подход может оказаться намного эффективнее. Рассмотрим следующий запрос:

```
SELECT film_id, description FROM sakila.film ORDER BY title LIMIT 50, 5;
```

Если таблица очень большая, его лучше переписать в следующем виде:

```
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);
```

Это отложенное соединение работает, потому что позволяет серверу просматривать так мало данных, как только возможно в индексе, не обращаясь к самим строкам. А после того, как только нужные строки будут найдены, их соединяют с исходной таблицей, чтобы сделать выборку недостающих столбцов. Аналогичный метод применим к соединениям с ключевым словом `LIMIT`.

Иногда вы также можете преобразовать запрос с `LIMIT` в позиционный запрос, который сервер может выполнить как сканирование диапазона индекса. Например, если вы предварительно вычислите `position` и построите по нему индекс, то предыдущий запрос можно будет переписать следующим образом:

```
SELECT film_id, description FROM sakila.film
WHERE position BETWEEN 50 AND 54 ORDER BY position;
```

Аналогичная проблема возникает при ранжировании данных, причем обычно к этой задаче примешивается раздел `GROUP BY`. Вам почти наверняка потребуется предварительно вычислить и сохранить ранги.

Проблема в работе с ключевыми словами `LIMIT` и `OFFSET` на самом деле заключается в `OFFSET`, из-за которого сервер генерирует и отбрасывает строки. Если вы используете своего рода закладку для запоминания позиции последней выбранной вами строки, то сможете сгенерировать следующий набор строк, начав с этой позиции вместо применения `OFFSET`. Например, если вы хотите разбить на страницы записи об аренде, начиная с последнего заключенного арендного договора и двигаясь в обратном направлении, то можете полагаться на тот факт, что их первичные ключи всегда увеличиваются. Можно получить первый набор результатов следующим образом:

```
SELECT * FROM sakila.rental
ORDER BY rental_id DESC LIMIT 20;
```

Этот запрос возвращает арендные договоры с идентификаторами с 16049 по 16030. Следующий запрос может продолжить возвращать договоры с этой точки:

```
SELECT * FROM sakila.rental
WHERE rental_id < 16030
ORDER BY rental_id DESC LIMIT 20;
```

Преимущество этого метода заключается в том, что он одинаково эффективен независимо от того, в каком месте таблицы вы находитесь.

Альтернативой может стать предварительное вычисление итогов или соединение со вспомогательными таблицами, которые содержат только первичный ключ и столбцы, необходимые для выполнения `ORDER BY`.

Оптимизация `SQL_CALC_FOUND_ROWS`

Еще один широко распространенный метод оптимизации разбиения на страницы — добавление подсказки `SQL_CALC_FOUND_ROWS` в запрос с ключевым словом `LIMIT`. Используя этот прием, вы узнаете, сколько строк было бы возвращено сервером, если бы не было ключевого слова `LIMIT`. Может показаться, что здесь происходит какое-то волшебство, позволяющее серверу предсказать, сколько строк он мог бы найти. Но, к сожалению, на самом деле сервер этого не делает — он не может подсчитывать строки, которые не отбирал. Это ключевое слово просто указывает серверу сгенерировать и отбросить остальную часть результирующего набора вместо того, чтобы останавливаться, когда он выберет требуемое количество строк. Это очень затратно.

Лучше включить в состав элемента разбиения на страницы ссылку на следующую страницу. Предполагая, что на странице 20 результатов, мы отбираем с помощью ключевого слова `LIMIT 21` строку, а отображаем только 20. Если 21-я строка существует в результатах, значит, имеется следующая страница и вы можете отобразить ссылку «следующая».

Еще одна возможность состоит в том, чтобы выбрать и кэшировать гораздо больше строк, чем вам нужно, например 1000, а затем извлекать их из кэша для последующих страниц. Такая стратегия позволяет вашему приложению узнать, насколько велик размер полного результирующего набора. Если в нем меньше 1000 строк, приложению точно известно, сколько ссылок на страницы нужно формировать, а если больше, оно может просто вывести сообщение: «Найдено более 1000 результатов». Обе стратегии гораздо более эффективны, чем многократная генерация полного набора и отбрасывание его большей части.

Иногда можно также просто оценить полный размер результирующего набора, выполнив команду `EXPLAIN` и просмотрев столбец `rows` в результате (эй, даже Google не показывает точное количество результатов!). Даже если вы не можете использовать ни один из описанных ранее приемов, выполнить отдельный запрос `COUNT(*)` для определения количества строк может быть намного быстрее, чем `SQL_CALC_FOUND_ROWS`, если он может задействовать покрывающий индекс.

Оптимизация UNION

MySQL всегда выполняет запросы UNION, создавая временную таблицу и заполняя ее результатами UNION. MySQL может применять не так уж много оптимизаций к запросам UNION. Вероятно, вам придется помочь оптимизатору, вручную опустив вниз фразы WHERE, LIMIT, ORDER BY и другие условия, то есть при необходимости скопировав их из внешнего запроса в каждый SELECT, входящий в UNION.

Очень важно всегда использовать UNION ALL, если только вам не нужно, чтобы сервер удалял строки-дубликаты. Если вы не укажете ключевое слово ALL, MySQL будет создавать временную таблицу в режиме different, что означает: для соблюдения уникальности строки сравниваются целиком. Такая операция довольно затратна. Однако имейте в виду, что наличие ключевого слова ALL не отменяет необходимости во временной таблице. MySQL всегда помещает результаты во временную таблицу, а затем считывает их оттуда, даже если в этом нет особой необходимости, например, когда результаты могут быть возвращены непосредственно клиенту.

Резюме

Оптимизация запросов — это заключительная деталь взаимосвязанной головоломки создания высокопроизводительных приложений, состоящей из схемы, индекса и создания запросов. Чтобы писать хорошие запросы, вам нужно разбираться в схемах и индексировании и наоборот.

В конечном счете речь по-прежнему идет о времени отклика и необходимости понимания того, как выполняются запросы, чтобы вы могли рассуждать о том, на что затрачивается время. Добавление нескольких процессов, таких как синтаксический анализ и оптимизация, — это всего лишь следующий шаг в понимании того, как MySQL обращается к таблицам и индексам, которые мы обсуждали в предыдущей главе. Дополнительное измерение, появляющееся, когда вы начинаете изучать взаимодействие между запросами и индексами, — это то, как MySQL обращается к одной таблице или индексу на основе данных, которые она находит в другой таблице.

Оптимизация всегда требует трехстороннего подхода: перестать что-то делать, делать это реже и делать быстрее.

ГЛАВА 9

Репликация

Встроенные в MySQL средства репликации являются основой для построения крупных высокопроизводительных приложений поверх MySQL с использованием так называемой масштабируемой архитектуры. Репликация позволяет сконфигурировать один или несколько серверов как подчиненные другому серверу (такие серверы называют *репликами*). При этом данные реплики синхронизированы с данными главного сервера. Это не только полезно для создания высокопроизводительных приложений, но и является краеугольным камнем многих стратегий обеспечения высокой доступности, масштабируемости, аварийного восстановления, резервного копирования, анализа, хранения данных и многих других задач.

В этой главе мы сосредоточимся не столько на том, что представляет собой каждая функция, сколько на том, когда ее использовать. Официальная документация MySQL (<https://dev.mysql.com/doc>) исключительно подробно объясняет, что такое полусинхронная репликация, репликация с несколькими источниками и т. д., и вам следует обращаться к ней при настройке этих функций.

ПРИМЕЧАНИЕ ПО ТЕРМИНОЛОГИИ

Давние пользователи MySQL знакомы с терминами *master* и *slave*, относящимися к репликации. Однако со временем они были заменены на «источник» (*source*) и «реплика» (*replica*). В этой книге предпринята попытка следовать новой терминологии в соответствии с этим изменением. В некоторых старых версиях MySQL все еще содержатся замененные термины, поэтому при необходимости обращайтесь к руководству по MySQL.

Обзор репликации

Основная проблема, которую призвана решить репликация, заключается в поддержании синхронизации данных между экземплярами сервера базы данных, имеющих одинаковую топологию. Это производится путем записи событий, которые изменяют данные или структуру данных, в двоичный журнал на сер-

вере-источнике. Затем серверы-реплики могут считывать события из журнала источника и воспроизводить их. Это создает асинхронный процесс, при котором не гарантируется актуальность копии данных реплики в любой момент времени. Относительно величины отставания также не дается никаких гарантий. Если запросы сложны, то могут привести к отставанию реплики от источника на секунды, минуты или даже часы.

Репликация MySQL в основном обратно совместима с предыдущими версиями, то есть сервер с более поздней версией MySQL обычно может без проблем быть репликой сервера, на котором установлена более ранняя версия. Однако более старые версии сервера часто не могут выступать в роли реплики более новых версий — они не распознают новые возможности, синтаксические конструкции SQL, к тому же могут существовать различия в форматах файлов, используемых при репликации более новым сервером. Например, вы не можете выполнить репликацию с сервера-источника с версией MySQL 5.6 в реплику с MySQL версии 5.5. Перед обновлением с одной основной или дополнительной версии на другую, например с 5.6 до 5.7 или с 5.7 до 8.0, рекомендуется протестировать настройку схемы репликации. Обновления в рамках вспомогательных версий, например с 5.7.34 до 5.7.35, будут совместимы — прочитайте журнал изменений в примечании к выпуску, чтобы точно узнать, что конкретно изменилось при переходе от версии к версии.

Репликация довольно хорошо применима для масштабирования операций чтения, которые вы можете направить на реплику, но это не лучший способ масштабирования операций записи, если только не учесть это требование при проектировании системы. Подключение большого количества реплик к источнику просто приводит к многократному выполнению операций записи, по одному разу для каждой реплики. Система в целом ограничена количеством операций записи, которые может выполнить ее самое слабое звено.

Перечислим некоторые из наиболее распространенных сценариев применения репликации.

- *Распределение данных.* Репликация MySQL обычно не очень требовательна к пропускной способности сети, хотя, как вы увидите позже, строчная репликация может потребовать гораздо большей пропускной способности, чем более традиционная репликация на основе операторов. К тому же вы можете останавливать и запускать репликацию по желанию. Таким образом, это полезно для хранения копии ваших данных в территориально удаленном месте, например в другом центре обработки данных или в облачном регионе. Удаленная реплика может работать даже с непостоянным соединением (специально или по другим причинам). Однако, если вы хотите, чтобы ваши реплики имели незначительное отставание репликации, понадобится надежный канал связи с малым временем задержки.

- *Балансировка нагрузки.* С помощью репликации MySQL может помочь вам распределить запросы на чтение между несколькими серверами MySQL, что очень хорошо работает для приложений с интенсивным чтением. Вы можете выполнить базовую балансировку нагрузки, внося несколько простых изменений в код. Для небольших приложений можете применить упрощенные подходы, такие как жестко заданные имена хостов, или воспользоваться циклическим разрешением DNS-имен (когда с одним доменным именем связаны несколько IP-адресов). Можно применять и более сложные подходы. Стандартные технологии для балансировки нагрузки, такие как продукты для балансировки сетевой нагрузки, могут прекрасно работать для распределения операций чтения между несколькими серверами MySQL.
- *Резервное копирование.* Репликация хорошо помогает при резервном копировании. Однако реплика не является ни резервной копией, ни ее заменой.
- *Аналитика и отчетность.* Использование выделенной реплики для выполнения запросов формирования отчетов/аналитики (онлайн-аналитическая обработка, или OLAP) — хорошая стратегия для обеспечения изоляции этой нагрузки от того, что нужно вашему бизнесу для обслуживания запросов внешних клиентов. Репликация — это хороший способ обеспечить изоляцию.
- *Высокая доступность и отказоустойчивость.* Репликация может помочь избежать превращения MySQL в единственную точку отказа в вашем приложении. Хорошая система аварийного переключения, куда входят реплицированные серверы, способна существенно сократить время простоя при отказе.
- *Тестирование обновлений MySQL.* Обычной практикой является создание реплики с обновленной версией MySQL и ее использование для проверки правильности работы ваших запросов перед обновлением каждого экземпляра сервера.

Как работает репликация

Прежде чем углубиться в детали настройки репликации, рассмотрим, как на самом деле MySQL реплицирует данные. Здесь мы рассматриваем простейшую топологию репликации, включающую один источник и одну реплику.

На высоком уровне репликация представляет собой простой процесс, состоящий из трех шагов.

1. Источник записывает изменения своих данных в двоичный журнал. Эти записи называются событиями двоичного журнала.
2. Реплика копирует события двоичного журнала источника в собственный локальный журнал ретрансляции.

3. Реплика воспроизводит события, записанные в журнал ретрансляции, применяя изменения к собственным данным.

Более подробно иллюстрирует самую простую форму репликации рис. 9.1.

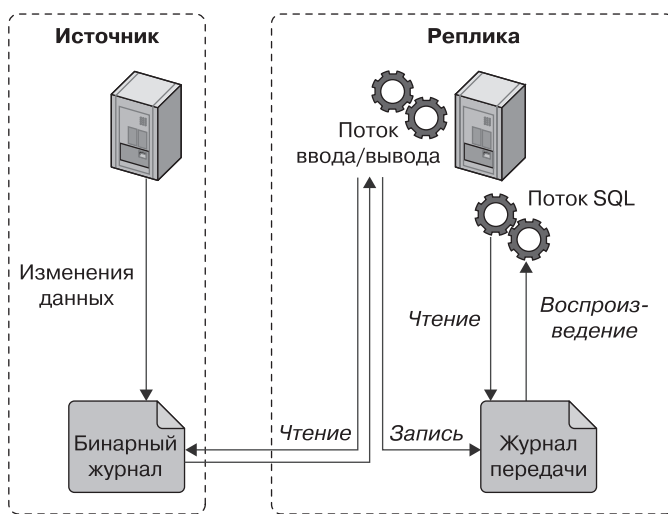


Рис. 9.1. Как работает репликация в MySQL

При этом архитектура репликации разделяет процессы выборки и воспроизведения событий на реплике, что позволяет им быть асинхронными, то есть поток ввода/вывода может работать независимо от потока SQL.

Взгляд на репликацию изнутри

Теперь, когда мы познакомили вас с основами репликации, углубимся в нее. Посмотрим, как на самом деле работает механизм репликации, познакомимся с его сильными и слабыми сторонами и рассмотрим некоторые дополнительные параметры конфигурации репликации.

Выбор формата репликации

MySQL предлагает три разных формата двоичных журналов для репликации: на основе *команд*, на основе строк и смешанный. Формат двоичных журналов управляется параметром конфигурации `binlog_format`, который определяет, как данные записываются в двоичный журнал.

Покомандная репликация (репликация на основе команд) работает путем протоколирования всех выполненных в источнике команд изменения данных. Когда реплика считывает событие из журнала ретрансляции и выполняет его, она на самом деле выполняет фактический SQL-запрос, который был ранее выполнен источником. Очевидное преимущество этого формата в том, что он простой и компактный. Запрос, который обновляет большие объемы данных, может занимать несколько десятков байт в двоичном журнале. Самым большим недостатком покомандной репликации является то, что она обычно имеет проблемы с недетерминированными запросами. Рассмотрим инструкцию, которая удаляет 100 строк таблицы из 1000 строк без предложения `ORDER BY`. Если строки в источнике и реплике упорядочены по-разному, вы можете удалить по 100 разных строк в каждой из них, что приведет к несоответствиям.

При построчной репликации (репликации на основе строк) в двоичный журнал записываются фактические изменения данных. Проще говоря, ее самое главное отличие от покомандной репликации заключается в том, что она обеспечивает детерминированность запросов. При использовании построчной репликации вы можете просмотреть двоичный журнал и точно увидеть, какие строки изменились и какими стали их новые значения. При использовании покомандной репликации SQL-команды интерпретируются во время выполнения и любые строки, выбранные сервером, изменяются. Недостаток метода построчной репликации заключается в том, что запись событий изменения данных каждой строки, затронутой запросом, может резко увеличить размер двоичного журнала.

В смешанном методе попытались объединить лучшее из обоих миров, используя формат покомандной репликации по умолчанию и переключаясь на построчную репликацию только тогда, когда это необходимо. Мы говорим «попытались», так как, несмотря на все усилия, у этого метода есть много условий¹, которые нужно выполнить, чтобы записать каждое из событий, что может вызвать появление непредсказуемых событий в двоичном журнале. Мы придерживаемся мнения, что двоичные данные журнала должны быть одним или другим, а не сочетанием того и другого.

Рекомендуем придерживаться построчной репликации, если у вас нет явной необходимости временно использовать покомандную репликацию. На практике в большинстве случаев построчная репликация обеспечивает самую безопасную репликацию ваших данных.

¹ Как и ожидалось, отсылаем вас к руководству, чтобы убедиться, что вы ознакомились с последними сведениями о том, как режим `MIXED` работает с различными типами операторов SQL.

Глобальные идентификаторы транзакций

До MySQL 5.6 реплика должна была отслеживать двоичный файл журнала и позицию в журнале, которую она читала при подключении к источнику. Например, реплика подключается к источнику и считывает данные из позиции 2749 в журнале `binlog.000002`. Поскольку реплика считывала события из этого двоичного журнала, она каждый раз перемещала позицию. А потом случилась катастрофа! Источник вышел из строя, и вам пришлось восстанавливать данные из резервной копии. Возник вопрос: как можно повторно подключить свою реплику, если бинарные журналы начинаются заново? Это был довольно сложный процесс чтения событий и определения того, к чему их привязывать. Если вы допустите ошибку и начнете со слишком ранней позиции в журнале, то можете продублировать события, а если со слишком поздней, то пропустите их. В любом случае неправильно присоединить реплику очень легко.

Для решения этой проблемы в MySQL был добавлен альтернативный метод отслеживания позиций репликации — глобальные идентификаторы транзакций (**GTID**). При использовании **GTID** каждой транзакции, которую совершает исходный сервер, присваивается уникальный идентификатор. Он представляет собой комбинацию `server_uuid`¹ и возрастающего номера транзакции. Когда транзакция записывается в двоичный журнал, вместе с ней записывается и **GTID**. Из обсуждения ранее в этой главе вы узнали, что реплика копирует двоичное событие журнала в свой локальный журнал ретрансляции и использует поток SQL-команд для применения изменений к локальной копии. Когда поток SQL-команд фиксирует транзакцию, он также записывает **GTID** как заверченный.

Рассмотрим пример, чтобы лучше проиллюстрировать этот процесс. Предположим, что наш сервер-источник только что настроен и на нем нет данных — даже не создана база данных. На этом сервере наш `server_uuid` был сгенерирован как `b9acac5a-7bbe-11eb-a043-42010af8001a`. Мы сделали то же самое со своей репликой и использовали соответствующие команды, чтобы дать указание реплике задействовать сервер-источник для репликации.

На сервере-источнике нужно создать новую базу данных:

```
CREATE DATABASE misc;
```

¹ Обратите внимание на то, что `server_uuid` отличается от `server_id` с таким же названием. Параметр `server_id` является определяемым пользователем значением, которое вы назначаете для своего сервера, тогда как `server_uuid` генерируется при первом запуске MySQL, если она не обнаруживает файл `auto.cnf`.

Это событие будет записано в двоичный журнал, чтобы реплика также могла создать базу данных. В двоичном журнале мы увидим одно событие, идентифицированное GTID:

```
b9acac5a-7bbe-11eb-a043-42010af8001a:1
```

Когда сервер-реплика применяет это событие, он запоминает завершение транзакции `b9acac5a-7bbe-11eb-a043-42010af8001a:1`.

В своем выдуманном примере предположим, что в этот момент на реплике мы останавливаем MySQL. Она закомитила одну транзакцию. Если наш источник продолжает принимать записи, список транзакций продолжит расти: 2, 3, 4, 5 и т. д. Когда мы запускаем резервное копирование реплики, она знает, что уже выполнила транзакцию 1 и может начать с обработки транзакции 2.

GTID решают одну из самых болезненных проблем запуска репликации MySQL — работу с файлами журналов и позициями. Настоятельно рекомендуем всегда включать GTID для своих баз данных, следуя инструкциям, приводимым в официальной документации MySQL.

Обеспечение безопасности при сбоях репликации

Хотя GTID помогает решить проблему с файлом журнала и положением, ряд других проблем также беспокоил администраторов MySQL. Позже в этой главе мы коснемся распространенных режимов отказа, однако перед этим рассмотрим несколько конфигураций, которые могут значительно улучшить работу с репликацией.

Чтобы свести к минимуму вероятность сбоя репликации, рекомендуем установить следующие параметры:

- `innodb_flush_log_at_trx_commit = 1`. Хотя, строго говоря, он и не является параметром репликации, но гарантирует, что при каждой транзакции журналы записываются на диск и синхронизируются. Этот параметр является полностью совместимым с ACID, обеспечивая максимальную защиту ваших данных — даже при репликации. Это связано с тем, что сначала фиксируются события двоичного журнала, а затем выполняется коммит транзакции и запись на диск. Установка этого значения равным 1 увеличит количество операций записи на диск, обеспечивая при этом долговременное хранение данных;
- `sync_binlog = 1`. Переменная определяет, как часто MySQL синхронизирует двоичные данные журнала с диском. Присвоение ей значения 1 означает синхронизацию перед каждой транзакцией. Это защищает от потери транз-

акций в случае сбоя сервера. Как и предыдущий параметр, этот увеличит количество операций записи на диск;

- `relay_log_info_repository = TABLE`. Репликация MySQL использовала файлы на диске для отслеживания позиции репликации. Это означало, что транзакции, завершённые репликацией, должны были синхронизироваться с диском в качестве второго шага. Если между фиксацией транзакции и синхронизацией произойдет сбой, файл на диске будет иметь неверный файл и неправильную позицию. Эта информация переместилась в таблицы InnoDB в самой MySQL, что позволяет репликации обновлять как транзакцию, так и информацию журнала ретрансляции в рамках одной и той же транзакции. Это создает атомарное действие и помогает в восстановлении после сбоя;
- `relay_log_recovery = ON`. Проще говоря, `relay_log_recovery` отбрасывает все локальные журналы ретрансляции при обнаружении сбоя и извлекает недостающие данные из источника. Это гарантирует, что любые поврежденные или неполные журналы ретрансляции на диске, которые могли возникнуть в результате сбоя, могут быть восстановлены. Этот параметр также устраняет необходимость использования `sync_relay_log`, поскольку в случае сбоя журналы ретрансляции удаляются. Нет необходимости выполнять лишние операции по их синхронизации с диском.

Отложенная репликация

При реализации некоторых сценариев может быть выгодно иметь в своей топологии отложенную реплику. Такую стратегию можно использовать для поддержания данных в оперативном режиме и в рабочем состоянии, но с постоянным отставанием от реального времени на несколько часов или дней. Это можно настроить с помощью оператора `CHANGE REPLICATION SOURCE TO` и параметра `SOURCE_DELAY`.

Представьте, что вы работаете с большим объемом данных и произошло случайное изменение — удалена таблица. Ее восстановление из резервной копии может занять несколько часов. С помощью реплики с задержкой по времени вы можете найти `GTID` оператора `DROP TABLE` и перехватить репликацию вплоть до момента, непосредственно предшествующего удалению этой таблицы. Часто это может помочь гораздо быстрее исправить ошибку.

Однако компромиссы здесь неизбежны. Хотя отложенная репликация может быть чрезвычайно полезна для смягчения последствий определенных сценариев потери данных, она усложняет многие другие операционные аспекты. Если вы решите, что нужно использовать отложенную репликацию, вам следует

также подумать о том, как правильно исключить эту отложенную реплику из числа кандидатов на узел-источник (если аварийное переключение записи автоматизировано, это еще более важно), как вы контролируете репликацию и как обрабатываете эту специальную реплику. Это лишь некоторые из дополнительных сложностей, которые необходимо учитывать при внедрении отложенных реплик.

Многопоточная репликация

Одна из исторических проблем с репликацией заключалась в том, что, хотя вы могли выполнять параллельную запись в свой источник, реплики были однопоточными. Современные версии MySQL предлагают многопоточную репликацию (рис. 9.2), при которой вы можете запускать несколько потоков применения SQL-команд для локальной реализации изменений из журнала ретрансляции.

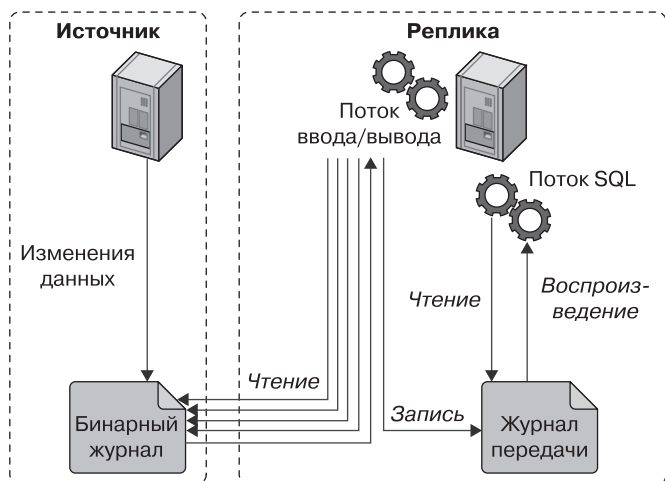


Рис. 9.2. Настройка многопоточной репликации

Существует два режима многопоточной репликации: `DATABASE` и `LOGICAL_CLOCK`. Параметр `DATABASE` использует несколько потоков для обновления разных баз данных, никакие два потока не будут обновлять одну и ту же базу данных одновременно. Этот метод хорошо работает, если вы распространяете свои данные по нескольким базам данных в MySQL и обновляете их последовательно и одновременно. Другой параметр, `LOGICAL_CLOCK`, позволяет выполнять параллельные обновления одной и той же базы данных, если они являются частью одного и того же группового коммита двоичного журнала.

ЧТО ТАКОЕ ГРУППОВОЙ КОММИТ ДВОИЧНОГО ЖУРНАЛА

Чтобы лучше объяснить это, используем прекрасную аналогию Моргана Токера — паром, пытающийся доставить пассажиров из пункта А в пункт Б.

В MySQL 5.0 паром заберет следующего пассажира в очереди из пункта А и перевезет его в пункт Б. Поездка между пунктами А и Б занимает около 10 мин в оба конца, поэтому вполне возможно, что за время движения парома прибудут несколько новых пассажиров.

Не имеет значения, когда паром прибудет обратно в пункт А, — он забирает только следующего пассажира в очереди.

В MySQL 5.6 паром забирает всех пассажиров в очереди в пункте А и затем переправляет их в пункт Б. Каждый раз, когда он возвращается в пункт А, чтобы забрать новых пассажиров, он собирает всех ожидающих и переправляет их в пункт Б.

Это значительно лучше производительность в реальных ситуациях, когда, как правило, многие пассажиры прибывают, ожидая возвращения парома в пункт А, а поездка между пунктами А и Б обычно занимает некоторое время. Это не поддается измерению в простейших тестах, выполняющихся в одном потоке.

MySQL 5.7 и более поздние версии ведут себя аналогично 5.6 в том смысле, что паром будет забирать из пункта А всех ожидающих пассажиров и доставлять их в пункт Б, но с одним заметным улучшением! Паром можно настроить так, чтобы, возвратившись в пункт А, чтобы забрать ожидающих пассажиров, он ждал чуть дольше, так как, скорее всего, прибудут новые пассажиры. Например, если вы знаете, что поездка между пунктами А и Б длится 10 мин, почему бы не подождать еще 30 с в пункте А перед отправлением? Это может помочь сэкономить на поездках туда и обратно и увеличить общее количество пассажиров, которых можно перевезти.

Переменными конфигурации, определяющими искусственную задержку, являются `binlog_group_commit_sync_delay` (задержка в микросекундах) и `binlog_group_commit_sync_no_delay_count` (количество транзакций, которых следует ожидать, прежде чем принять решение о прекращении ожидания).

В этом примере пассажиры, очевидно, являются аналогом транзакций, а паром — дорогостоящей операцией `fsync()`. Важно отметить, что работает только один паром (один набор упорядоченных двоичных журналов), поэтому возможность его настройки обеспечивает хороший уровень расширенной конфигурации.

В большинстве случаев вы можете просто включить эту функцию и получить немедленную выгоду, задав для параметра `replica_parallel_workers` ненулевое значение. Если работаете с одной базой данных, вам также потребуются изменить значение параметра `replica_parallel_type` на `LOGICAL_CLOCK`. Поскольку многопоточная репликация использует поток-координатор, для этого потока

возникнут некоторые накладные расходы, связанные с управлением состояниями всех других потоков. Кроме того, убедитесь, что ваши реплики работают с параметром `replica_preserve_commit_order`, чтобы коммит не по порядку не вызывал проблем. Подробное объяснение того, почему это важно, смотрите в разделе Gaps официальной документации (<https://oreil.ly/Tjb28>).

Есть два способа определить оптимальное значение параметра `replica_parallel_workers`. Не очень точный метод такой: остановить репликацию, а затем измерить, сколько времени потребуется, чтобы наверстать упущенное, используя разное количество потоков, пока не будет найдена оптимальная настройка. Однако это некорректный метод, поскольку предполагается, что при репликации отправляется постоянное количество операторов языка манипулирования данными (DML) и что все они работают относительно одинаково. На практике так будет вряд ли.

Более точный метод определения полученного уровня параллелизма заключается в анализе того, насколько загружен каждый из прикладных потоков для вашей рабочей нагрузки. Чтобы сделать это, требуется включить потребители и инструменты Performance Schema, разрешить сбор некоторой информации, а затем просмотреть результаты. Для начала нужно включить следующее¹:

```
UPDATE performance_schema.setup_consumers SET ENABLED = 'YES'
WHERE NAME LIKE 'events_transactions%';
```

```
UPDATE performance_schema.setup_instruments SET ENABLED = 'YES', TIMED = 'YES'
WHERE NAME = 'transaction';
```

И разрешить репликацию для обработки событий в течение определенного периода. В идеале вы должны рассматривать это во время самых тяжелых рабочих нагрузок записи или в любое время, когда видите увеличение отставания репликации:

```
mysql> USE performance_schema;
events_transactions_summary_by_thread_by_event_name.thread_id AS THREAD_ID,
events_transactions_summary_by_thread_by_event_name.count_star AS COUNT_STAR
FROM events_transactions_summary_by_thread_by_event_name
WHERE
events_transactions_summary_by_thread_by_event_name.thread_id IN (SELECT
replication_applier_status_by_worker.thread_id
FROM replication_applier_status_by_worker);
```

¹ Потребители и инструменты Performance Schema заставляют MySQL собирать дополнительные данные о своих внутренних компонентах, которые могут использовать дополнительный ЦП. Напоминаем, что всегда следует заранее проверять, как подобные изменения повлияют на рабочие нагрузки, в безопасной среде.

```

+-----+-----+
| THREAD_ID | COUNT_STAR |
+-----+-----+
| 1692957 | 23413 |
| 1692958 | 7150 |
| 1692959 | 1568 |
| 1692960 | 291 |
| 1692961 | 46 |
| 1692962 | 9 |
+-----+-----+
6 rows in set (0.00 sec)

```

Этот запрос поможет вам определить, сколько транзакций обрабатывается каждым потоком. Как видно из результатов примера рабочей нагрузки, оптимальная конфигурация включает три-четыре потока, а все, что сверх этого, будет использоваться очень редко.

Полусинхронная репликация

Когда вы включаете полусинхронную репликацию, каждая транзакция, которую коммитит ваш источник, должна быть подтверждена как полученная по крайней мере одной репликой¹. Уведомление подтверждает, что реплика получила его и успешно записала в собственный журнал ретрансляции (но не обязательно применила к локальным данным).

Поскольку каждая транзакция должна ожидать ответа от других узлов, эта функция увеличивает запаздывание выполнения коммита для каждой транзакции, которую выполняет ваш сервер. Это означает, что вам нужно учитывать возможные компромиссы.

Здесь следует отметить очень важную вещь: если ни одна реплика не подтвердит транзакцию в течение определенного времени, MySQL возвращается к стандартной асинхронной репликации. Это не приведет к сбою транзакции. Это действительно помогает проиллюстрировать, что полусинхронная репликация не инструмент для предотвращения потери данных, а скорее строительный блок для более широкого набора инструментов, который позволяет обеспечить более отказоустойчивое аварийное переключение на другой ресурс.

В связи с возвратом к асинхронному режиму мы изо всех сил пытались найти хороший сценарий использования, объясняющий, почему вы должны включить

¹ Требуемое количество реплик — это настраиваемый параметр (`rpl_semi_sync_source_wait_for_replica_count`). В более широких топологиях вы можете потребовать два или даже три подтверждения перед завершением исходной транзакции.

его. Логичным вариантом было бы подтверждение того, что в случае разрыва сетевого соединения изолированный источник все еще не записывает данные, будучи отделенным от своих реплик. К сожалению, этот источник просто вернется к асинхронному режиму и продолжит принимать записи. По данной причине мы рекомендуем не полагаться на это для обеспечения целостности данных.

Фильтры репликации

Параметры фильтрации позволяют вам реплицировать только часть данных, хранящихся на сервере, что, возможно, не оправдывает ваших ожиданий. Существует два типа фильтров репликации: те, которые применяются при записи событий в двоичный журнал на источнике, и те, которые фильтруют события, поступающие из журнала ретрансляции на реплике. И те и другие изображены на рис. 9.3.

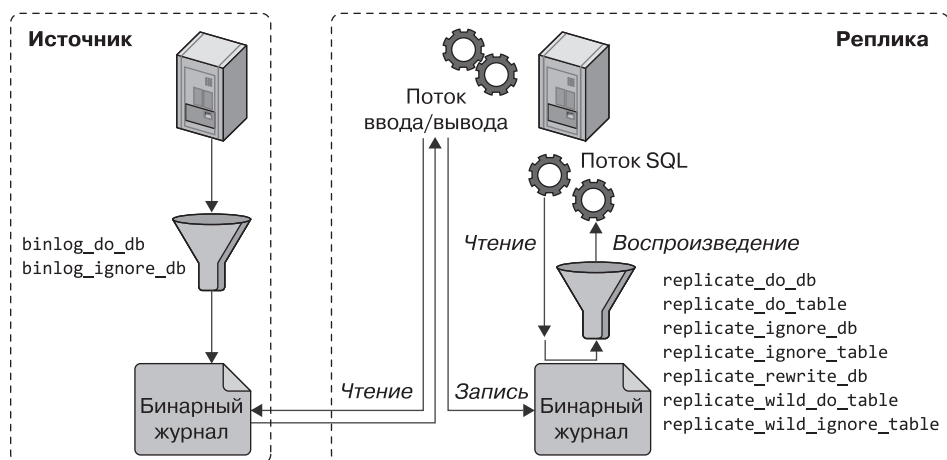


Рис. 9.3. Параметры фильтрации репликации

Параметры, управляющие фильтрацией двоичного журнала, — `binlog_do_db` и `binlog_ignore_db`. Но, как мы скоро поясним, их не надо включать, если, конечно, вы не хотите постоянно объяснять своему шефу, почему данные пропадают и их нельзя восстановить.

На реплике параметры `replicate_*` управляют фильтрацией событий, считываемых потоком SQL из журнала ретрансляции. Вы можете реплицировать или игнорировать одну или несколько баз данных, перезаписывать одну базу данных в другую базу данных, а также реплицировать или игнорировать определенные таблицы, задаваемые с помощью паттерна LIKE.

Самое важное, что нужно понять об этих параметрах, — то, что параметры `*_do_db` и `*_ignore_db` как в источнике, так и в реплике не работают так, как можно было бы ожидать. Естественно полагать, что фильтрация производится по имени базы данных объекта, но на самом деле анализируется имя текущей базы данных по умолчанию, то есть если выполнить на источнике такие команды:

```
USE test;  
DELETE FROM sakila.film;
```

Очень важно понимать, что параметры `*_do_db` и `*_ignore_db` будут фильтровать оператор `DELETE` в базе `test`, а не в базе `sakila`. Обычно это не то, что вам нужно, и может привести к повторению или игнорированию неправильных утверждений. Хотя у параметров `*_do_db` и `*_ignore_db` есть полезное применение, требуются они редко, так что будьте осторожны. Если вы используете эти параметры, то реплики очень легко могут оказаться рассогласованными или начать сбоить.

Параметры `binlog_do_db` и `binlog_ignore_db` не только могут нарушить репликацию, они также делают невозможным восстановление из резервной копии данных на определенный момент времени в прошлом. В большинстве ситуаций вы не должны их использовать.

В общем, фильтры репликации — это проблема, ожидающая своего решения. Предположим, что вы хотите предотвратить распространение изменений привилегий на реплики, что является довольно распространенной целью. (Такое желание может натолкнуть вас на мысль, что вы делаете что-то не так — возможно, есть и другие способы достичь цели.) 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, работая с фильтрами. Возможно, было бы лучше предотвратить репликацию конкретных команд, обычно с помощью `SET SQL_LOG_BIN=0`, хотя такая практика имеет собственные опасности. В общем случае к фильтрам репликации следует подходить с особой осторожностью и применять их только при острой необходимости, потому что они легко могут нарушить репликацию и вызвать проблемы, которые проявятся в самый неподходящий момент, например во время аварийного восстановления.

При этом в определенных ситуациях фильтры репликации могут оказаться полезными. Возможно, вы создали несколько баз данных, `users_1`, `users_2`, `users_3` и `users_4`, и теперь производительность на сервере сильно снижается. Восстановив резервную копию и подключив репликацию, вы можете подготовиться к переносу запросов для `users_3` и `users_4` на другой сервер. Это прекрасно работает, однако в новой базе данных все еще есть `users_1` и `users_2`. В какой-то момент вам придется удалить данные, которые могут повлиять на

производительность. Рассмотрим эту альтернативу. Вы восстанавливаете свою резервную копию, а затем удаляете `users_1`, `users_2`. Затем настраиваете правило репликации, чтобы игнорировать `users_1` и `users_2`, и завершаете настройку репликации. Теперь вы обрабатываете события только для `users_3` и `users_4` на новом сервере. Справившись с настройкой репликации, можете запустить производственный трафик.

Параметры фильтрации хорошо описаны в руководстве по MySQL, поэтому здесь мы не будем повторяться.

Отказоустойчивость репликации

В начале главы мы упомянули, что репликация, помимо прочего, является краеугольным камнем высокой доступности. Наличие в другом месте постоянно обновляемой копии ваших данных значительно упрощает восстановление после аварии в сравнении с использованием резервной копии. Более того, иногда вам просто нужно выполнить техническое обслуживание, включающее перезапуск MySQL.

В этом разделе мы хотим поговорить о правильных способах повышения реплики, когда она становится источником. При этом легко ошибиться, что способно привести к проблемам с данными и к длительному простоям. Мы хотим уточнить, что выражения «повышение реплики» и «переход на другой ресурс» — синонимы. Оба они означают понижение роли источника от записи и повышение роли реплики до роли источника.

Гораздо более подробное объяснение того, как этим управлять, имеется в официальной документации MySQL в разделе «Переключение источников во время отработки отказа», но, учитывая, насколько важно все сделать правильно, хотя бы немного затронем эту проблему.

Запланированные повышения

Наиболее распространенной причиной реплики является какое-либо событие обслуживания, включая исправление безопасности, обновления ядра и даже просто перезапуск MySQL, поскольку существует несколько параметров конфигурации, требующих перезапуска. Этот тип повышения называется контролируемым или запланированным повышением.

Чтобы успешно реализовать это повышение, выполните следующие шаги.

- Определите, какую реплику вы собираетесь повышать. Обычно это реплика, в которой, как вы уверены, есть все данные. Это ваш новый источник.

- Проверьте задержку, чтобы убедиться, что она находится в пределах нескольких секунд.
- Прекратите выполнять записи на текущем источнике, установив параметр `super_read_only`¹.
- Подождите, пока реплика не будет синхронизирована с источником. Сравните `GTID`, чтобы удостовериться в этом.
- Отключите `read_only` для нового источника.
- Переключите трафик приложения на новый источник.
- Перенастройте на новый источник все реплики, включая реплику с пониженным статусом (старый источник). Это тривиально с `GTID` и `AUTO_POSITION=1`.

Незапланированные повышения

В течение довольно длительного времени эксплуатации каждая система выходит из строя из-за сбоя либо программного, либо аппаратного обеспечения. Когда аварийный отказ происходит на сервере-источнике, где выполняется запись, это может серьезно повлиять на пользовательский интерфейс. Большинство приложений просто вернут сообщение об ошибке, оставляя пользователю возможность повторить попытку самостоятельно. Это тот случай, когда необходимо незапланированное повышение.

Поскольку у вас нет реального источника для проверки, реализуйте сокращенный вариант запланированного повышения, где вы сами выбираете, какая реплика основана на уже реплицированных данных.

- Определите, какую реплику вы собираетесь повышать. Обычно это реплика, в которой, как вы уверены, есть все данные. Это ваш новый источник.
- Отключите `read_only` для нового источника.
- Переключите трафик приложения на новый источник.
- Перенастройте на новый источник все реплики, включая реплику с пониженным статусом (старый источник), когда она вернется в эксплуатацию. Это тривиально с `GTID`.

Вы также должны убедиться, что, когда прежний источник снова подключается к сети, по умолчанию включен режим `super_read_only`. Это поможет предотвратить любые случайные записи.

¹ Параметр `super_read_only` неявно включает `read_only`. И наоборот, отключение `read_only` неявно отключает `super_read_only`. У вас нет причин включать или отключать обе переменные одновременно во время этого процесса.

Компромиссы повышения

Мы вынуждены отметить, что иногда первой реакцией на простои является аварийное переключение на другой ресурс. Поскольку определить, сколько данных может отсутствовать в целевом объекте, сложнее, иногда лучше отказаться от аварийного переключения.

Незапланированное повышение — это не очень распространенное мероприятие, то есть вы делаете это не очень часто. Когда вас попросят это сделать, вам может понадобиться просмотреть документацию, чтобы убедиться, что вы не пропустили ни одного шага. Вы также должны проверить другие реплики, чтобы убедиться, какая из них является вероятным кандидатом на повышение. Все это требует времени. В некоторых случаях может оказаться быстрее дождаться, пока ваш сервер или процесс MySQL вернется в оперативный режим. Преимущество этого заключается в том, что если вы выполнили действия по обеспечению соответствия требованиям ACID, описанные в главе 5, то не потеряли никаких данных и ваши реплики продолжают работу с того места, на котором остановились.

Топологии репликации

MySQL позволяет настроить репликацию практически для любой конфигурации источников и реплик. Возможны различные сложные топологии, но даже самые простые могут быть весьма гибкими. Одна и та же топология может иметь множество различных применений. Описания разнообразия способов использования репликации вполне может хватить на отдельную книгу.

Эта гибкость означает, что вы можете легко разработать топологию, которую невозможно будет обслуживать. Мы настоятельно рекомендуем максимально упростить топологию репликации, при этом удовлетворив свои потребности. С учетом сказанного советуем две возможные стратегии, которые должны охватывать практически все варианты использования. У вас могут быть весьма основательные причины для отклонения от них, но обязательно спросите себя, решаете ли вы правильные проблемы, когда переходите к более сложным топологиям репликации.

Активный/пассивный

В топологии «активный/пассивный» вы направляете все операции чтения и записи на один исходный сервер. Кроме того, поддерживаете небольшое количество пассивных реплик, которые активно не обслуживают трафик приложений. Основная причина выбора этой модели — то, что вас не волнует вопрос

об отставании репликации. Поскольку все операции чтения идут к источнику, вы предотвращаете любые проблемы чтения после записи, которые могут быть неприемлемыми для приложения. На рис. 9.4 показана эта топология с несколькими репликами.

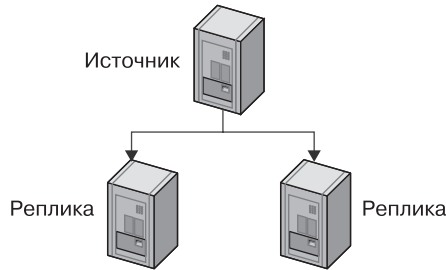


Рис. 9.4. Источник с несколькими репликами

Конфигурация

Мы предполагаем, что в этой топологии источник и реплики имеют идентичные конфигурации с точки зрения процессора, памяти и т. д. В течение довольно длительного периода эксплуатации системы вам потребуется выполнить аварийное переключение с текущего работающего источника на одну из реплик либо для обслуживания, обновления программного обеспечения или исправления, либо даже из-за сбоя оборудования. Имея на репликах ту же аппаратную и программную конфигурацию, вы гарантируете, что сможете поддерживать производительность и пропускную способность трафика, как до аварийного переключения.

Избыточность

В физической аппаратной среде вам действительно нужно резервирование $n + 2$ как минимум для трех серверов. В случае сбоя оборудования у вас все еще есть один дополнительный сервер для аварийного переключения. Можно также использовать одну из реплик в качестве сервера резервного копирования, если вам неудобно или вы не можете создавать резервные копии на своем источнике.

В облачной среде можете обойтись резервированием $n + 1$ для двух серверов, если данные довольно малы или вы можете легко скопировать их. В противном случае необходимо резервирование $n + 2$. Если вы ограничитесь резервированием $n + 1$, динамический характер предоставления облачных услуг может упростить управление. Для событий обслуживания, таких как установка исправлений, проще предоставить третью реплику по запросу, выполнить с ней все

необходимые действия (например, обновить ядро или применить обновление безопасности), а затем заменить другую реплику. Затем следует выполнить отработку отказа и повторить процесс на прежнем источнике. Цель состоит в том, чтобы реплика всегда была готова к аварийному переключению.

Во всех случаях вы можете разместить одну из своих реплик в географически удаленном месте, при этом необходимо обратить внимание на отставание репликации и убедиться, что она достаточна для вашего сценария использования. Реплики должны быть восстанавливаемыми, а любая потеря данных должна соответствовать установленным вами правилам. Мы поговорим об этом подробнее в разделе «Определение требований к восстановлению» в главе 10.

Предостережения

Выбирая эту модель, вы явно привязываете масштабирование чтения к пропускной способности одного сервера. Если достигнете предела масштабирования чтения, придется выйти за пределы этой топологии — скорее всего, перейти к топологии «активный/пул чтения» — или использовать сегментирование, чтобы уменьшить количество операций чтения в источнике.

Активный/пул чтения

В топологии «активный/ пул чтения» вы направляете все операции записи в источник. Операции чтения могут быть отправлены либо на исходный сервер, либо в пул чтения в зависимости от потребностей приложения. Пул чтения позволяет масштабировать операции чтения по горизонтали для приложений с интенсивным чтением. В какой-то момент горизонтальное масштабирование упадет из-за большого количества запросов репликации на источник.

На рис. 9.5 показана эта схема с одним источником и пулом реплик.

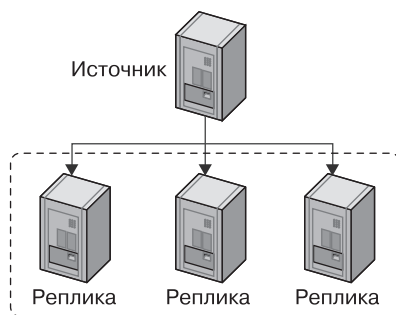


Рис. 9.5. Источник с пулом чтения

Конфигурация

В идеале вам нужна идентичная конфигурация между источником и хотя бы одной, а лучше двумя репликами в пуле чтения. Опять же в какой-то момент вы будете вынуждены переключиться на одну из этих реплик, и она должна иметь достаточную пропускную способность, чтобы обслуживать ваш трафик.

Если вы заметите, что нагрузка на пул со временем растет, можете оптимизировать затраты и использовать другую конфигурацию для некоторых реплик. Для этого попробуйте проанализировать возможность приоритизации трафика как способ его балансировки между репликами. Если у вас имеются 32 ядра процессора для целей аварийного переключения и восемь ядер для других реплик, попробуйте отправить в четыре раза больше трафика на 32-ядерный узел, чтобы обеспечить более эффективное использование.

Избыточность

Количество серверов в вашем пуле должно соответствовать ранее указанным требованиям, то есть по крайней мере один сервер может быть задействован для целей аварийного переключения. Кроме того, вам нужны достаточное количество узлов для обслуживания трафика чтения, а также небольшой буфер на случай сбоя узлов. При чтении наиболее вероятным индикатором использования будет загрузка процессора, поэтому ориентируйтесь на уровень загрузки 50–60 % на узел в пуле. По мере увеличения загрузки процессор тратит больше времени на переключение контекста между рабочими процессами, и задержка увеличивается. Постарайтесь найти правильный баланс между задержкой и использованием, соответствующий ожиданиям вашего приложения.

Предостережения

В тех случаях, когда вы используете пул чтения, приложение должно иметь некоторую толерантность к операциям чтения устаревших данных. Вы никогда не сможете гарантировать, что запись, которую выполнили в источнике, моментально была реплицирована в реплику. Вам также может понадобиться способ удаления из пула узлов, слишком сильно отстающих при репликации.

Размер пула чтения также может сильно влиять на объем задач администрирования, которые вам приходится выполнять, и на то, когда следует обратить внимание на автоматизацию. Пул из 16 узлов будет означать, что вам придется выполнять обновления ядра или исправления безопасности 16 раз. Автоматизация этой задачи для безопасного удаления узла из пула, выполнения исправлений, перезагрузки и повторного объединения уменьшит объем работы, которую вы будете выполнять вручную в будущем.

Нерекомендуемые топологии

Используя любую из двух рекомендаций, которые мы дали в этой главе, вы сохраняете свою топологию простой и понятной. Есть ряд других предложений из более ранних изданий этой книги, или, возможно, вы случайно слышали о том, как устроена топология другой компании. Мы называем некоторые из них нерекомендуемыми, потому что они сопряжены с большими риском и сложностью, чем нам хотелось бы.

Два источника в активно-активном режиме

Репликация с двумя источниками (известная также как двунаправленная репликация) включает в себя два сервера, каждый из которых настроен как источник и реплика другого — другими словами, пара совместных источников. На рис. 9.6 показана эта топология.

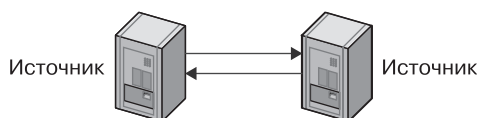


Рис. 9.6. Два источника в активно-активном режиме

На первый взгляд это ничем не отличается от активного/пассивного режима с двумя серверами, за исключением того, что репликация настроена и в обратном направлении. Реальная опасность заключается в том, что вы явно отправляете трафик записи на обе стороны, следовательно, на две активные части.

Топологию «актив/актив» очень сложно выполнить правильно. Некоторые стратегии предполагают выбор стороны для отправки на основе четного/нечетного хеширования. Это гарантирует согласованность операций чтения после записи для одной и той же строки, но запросы, которые включают строки, являющиеся каноническими на другой стороне, могут быть несогласованными. Проще говоря, чтение строк с идентификаторами 1, 3 и 5 с одной стороны всегда будет согласованным. А как насчет запроса, который считывает ID 1–6? Куда вы отправляете этот запрос? Что делать, если на другой стороне существует обновление, которое не отражается на этой стороне из-за отставания репликации?

Вам также необходимо тщательно сбалансировать пропускную способность. В сценарии совместного источника каждый сервер является репликой другого сервера и наиболее вероятным ресурсом для целей аварийного переключения. Вы должны спланировать свою пропускную способность таким образом, чтобы гарантировать, что при перемещении трафика с одной стороны на другую не исчерпаете ресурсы процессора. Вы также терпите неудачу и вводите со-

вершенно другой рабочий набор данных. Буферный пул InnoDB теперь должен обновиться, удаляя записи, чтобы освободить место для нового «горячего» набора данных.

Учтите наш совет и держитесь подальше от этого варианта. Может показаться, что вы получаете пользу от пассивного сервера, заставляя его обрабатывать трафик вместо того, чтобы простаивать. В итоге вы внесете несогласованность данных в приложение и всегда будете на грани того, что ресурсов для аварийного переключения окажется недостаточно. Теряя свою стратегию аварийного переключения, вы теряете устойчивость.

Два источника в активно-пассивном режиме

Существует разновидность топологии с двумя источниками, которая позволяет избежать всех подводных камней, которые мы только что обсуждали. Основное ее отличие состоит в том, что один из серверов является пассивным сервером только для чтения, как показано на рис. 9.7.

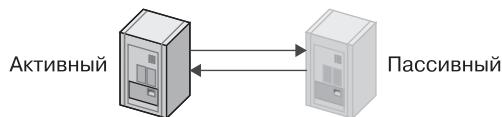


Рис. 9.7. Два источника в активно-пассивном режиме

На первый взгляд в этой топологии действительно нет ничего плохого. Единственным ее отличием от рекомендованной топологии «активный/пассивный» — то, что репликация предварительно настроена на другой сервер. Это работает только в конфигурации с двумя серверами. Если вы запускаете более двух серверов, нужно решить, какой узел лучше всего подходит для аварийного переключения. Предварительная настройка репликации напрямую привязывает вас к одному из них и не обеспечивает гибкости в ситуации сбоя.

Мы утверждаем, что настройка репликации — это простой и автоматизированный шаг в рамках процесса восстановления после отказа, о котором мы говорили ранее. Это ненужная конфигурация, которая только вызывает путаницу.

Два источника с репликами

Усложняя ситуацию еще больше, мы можем добавить одну или несколько реплик к каждому источнику, как показано на рис. 9.8.

Это устраняет большинство проблем с двойным источником в режиме «активный/активный», наиболее важной из которых является то, как вы маршрутизируете трафик. Это устраняет проблемы, связанные с планированием пропускной

способности и обновлением пула буферов при восстановлении после отказа. В отработке отказа у вас есть дополнительные шаги, призванные указать одному из источников на недавно повышенную реплику, связанную с ним.

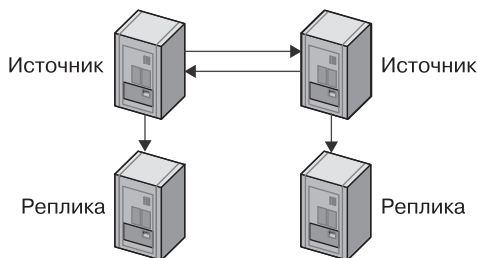


Рис. 9.8. Топология с двумя источниками и репликами

Мы против этого варианта — в основном из-за проблем с доступом к данным. Совместные источники определенно ведут к неприятностям.

Кольцевая репликация

Кольцевая репликация имеет три и более источника, где каждый сервер является репликой предшествующего ему сервера в кольце и источником для последующего сервера, как показано на рис. 9.9. Эта топология также называется циклической репликацией.

Если какой-либо сервер в этой топологии отключается, она нарушается и обновления перестают распространяться по кольцу. Существуют варианты присоединенных реплик, где каждый источник, обозначенный на рис. 9.9, имеет выделенную реплику для замены. Это по-прежнему означает, что кольцо разорвано до тех пор, пока вы не переместите реплику в прежнее положение.

Эта топология довольно сложна и не имеет преимуществ.

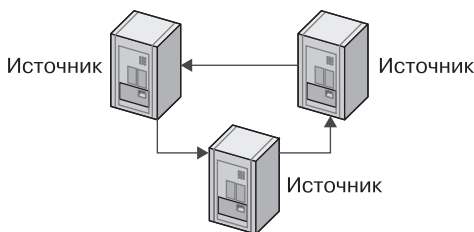


Рис. 9.9. Топология кольцевой репликации

Репликация с несколькими источниками

Несмотря на то что важно сохранять простоту топологии репликации, могут возникнуть ситуации, когда вам потребуется использовать более продвинутые функции для обработки исключительных случаев. Предположим, вы создали совершенно новый сайт для загрузки и просмотра видео, который сейчас становится популярным. Одним из ваших ранних проектных решений было разделение данных о видео и данных о пользователях на два отдельных кластера базы данных. По мере роста трафика вы обнаружите, что хотите объединить их данные в запросах. Этого можно достичь с помощью репликации с несколькими источниками, чтобы объединить оба набора данных в реплику, как показано на рис. 9.10.

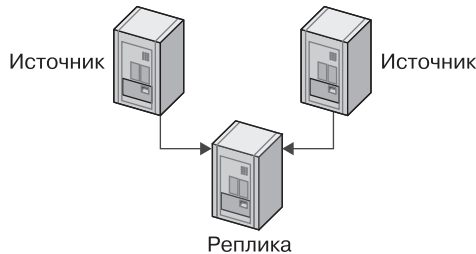


Рис. 9.10. Репликация с несколькими источниками

Эта функциональность основана на концепции, называемой каналами репликации. В рассматриваемом примере вам понадобится третий кластер для MySQL. В новом кластере будут созданы два канала репликации: один для видеоданных и один для пользовательских данных. После того как загрузили и реплицировали данные, вы можете получить очень короткое время простоя, в течение которого замораживаете запись в оба источника и заставляете свой код переключиться на чтение и запись в новую объединенную базу данных. Вуаля, теперь вы объединили две базы данных в одну.

Прежде чем мы двинемся дальше, следует узнать об одном важном ограничении: вы не можете настроить реплику для многократного использования репликации с несколькими источниками для одного и того же источника.

Эта топология предназначена в основном для особых условий применения. Мы не рекомендуем только строить вокруг этой концепции постоянную топологию. Ее временное использование для слияния данных по-прежнему является приемлемым сценарием, но в итоге стоит вернуться к одной из наших двух рекомендаций.

Администрирование и обслуживание репликации

При небольшом объеме данных и постоянной рабочей нагрузке на запись маловероятно, что вам очень часто придется следить за отставанием репликации или, что еще хуже, ее прерыванием. Однако размер большинства баз данных со временем увеличивается, а вместе с этим возникает необходимость в обслуживании.

Мониторинг репликации

При использовании репликации сложность мониторинга MySQL повышается. Хотя на самом деле репликация происходит как на источнике, так и на реплике, большая часть работы выполняется на реплике, и именно здесь чаще возникают наиболее распространенные проблемы. Все реплики работают? Были ли какие-то ошибки на репликах? Насколько отстала самая медленная реплика? По умолчанию MySQL предоставляет большую часть информации, необходимой для того, чтобы ответить на эти вопросы, но автоматизация процесса мониторинга и обеспечение надежности репликации возлагаются на вас.

При настройке мониторинга репликации есть несколько моментов, которые мы считаем наиболее важными.

- *Для репликации требуется дисковое пространство как на источнике, так и на реплике.* Как вы видели на рис. 9.1, при репликации используются как двоичные журналы на источнике, так и журналы ретрансляции на репликах. Если на исходном диске нет свободного места, транзакции не смогут завершиться и начнется тайм-аут. Если то же самое происходит с репликой, MySQL ведет себя немного более корректно, приостанавливая репликацию и ожидая появления свободного места на диске. Вам нужно будет контролировать доступное дисковое пространство на них обоих, чтобы обеспечить непрерывную работу репликации.
- *Репликацию необходимо контролировать на предмет состояния и ошибок.* Хотя репликация давно используется и является очень надежной функцией, внешние факторы, такие как проблемы с сетью, несогласованность и повреждение данных, могут привести к ее сбою. Из-за этих проблем необходимо отслеживать, выполняются ли потоки репликации, а если нет, выяснять, в чем заключается последняя ошибка, чтобы определить, каким должен быть следующий шаг. Подробнее об устранении конкретных проблем мы расскажем в разделе «Проблемы с репликацией и их решения» далее.
- *Отложенная репликация должна быть задержана так, как предполагалось.* Поскольку ранее мы упоминали отложенную репликацию, рекомендуется настроить мониторинг, чтобы убедиться, что отложенные реплики действительно

задерживаются на нужное время. Слишком большое отставание реплики может привести к тому, что ее использование станет требовать больше времени. Если задержка слишком мала или, что еще хуже, ее вообще нет, отложенная копия может оказаться бесполезной, когда она вам понадобится.

Измерение отставания репликации

Одна из наиболее распространенных вещей, которую вам нужно будет отслеживать, — это то, насколько реплика отстает от источника. Хотя столбец `Seconds_behind_source` в `SHOW REPLICA STATUS` теоретически показывает отставание реплики, на самом деле оно не всегда точно по целому ряду причин.

- Реплика вычисляет значение `Seconds_behind_source` сравнением текущей метки времени сервера с меткой времени, записанной в событии двоичного журнала, поэтому реплика даже не может сообщить о своем отставании, если только она не обрабатывает запрос.
- Реплика обычно сообщает значение `NULL`, если потоки репликации не запущены.
- Некоторые ошибки (например, несоответствие настроек параметра `max_allowed_packet` источника и реплики или нестабильная сеть) могут нарушить репликацию и/или остановить потоки репликации, но столбец `Seconds_behind_source` будет содержать `0`, а не укажет на ошибку.
- Реплика иногда не может рассчитать задержку, даже если процессы репликации запущены. Если это произойдет, реплика может сообщить либо значение `0`, либо `NULL`.
- Очень длительная транзакция может привести к большим вариациям сообщаемой задержки. Например, если у вас есть транзакция, которая обновляет данные, остается открытой в течение часа, а затем коммитится, обновление войдет в двоичный журнал через час после того, как оно действительно произошло. Когда реплика обработает запрос, она временно сообщит, что отстает от источника на час, а затем вернется к нулевому отставанию в секундах.

Решение этих проблем состоит в том, чтобы игнорировать `Seconds_behind_source` и отслеживать отставание реплики с помощью чего-то, что можно наблюдать и измерять напрямую. Лучшим решением является запись `heartbeat` (пульса), представляющая собой отметку времени, которую вы обновляете один раз в секунду в источнике. Чтобы вычислить задержку, можете просто вычесть `heartbeat` из текущей метки времени на реплике. Этот метод невосприимчив ко всем проблемам, о которых мы только что упомянули, и у него есть дополнительное преимущество, заключающееся в создании удобной метки времени, которая показывает, в какой момент времени данные реплики актуальны.

Скрипт `pt-heartbeat`, включенный в `Percona Toolkit`, является наиболее популярной реализацией пульса репликации.

`heartbeat` имеет и другие преимущества. Записи пульса репликации в двоичном журнале полезны для многих целей, например для аварийного восстановления в трудноразрешимых сценариях.

Ни один из показателей задержки, которые мы только что упомянули, не дает представления о том, сколько времени потребуется реплике, чтобы фактически догнать источник. Это зависит от многих факторов, таких как пропускная способность реплики и количество запросов на запись, которые обрабатывает источник. Дополнительную информацию по этой теме вы найдете в подразделе «Слишком большое отставание репликации» раздела «Проблемы с репликацией и их решения».

Как определить, согласованы ли реплики с источником

В идеальном мире реплика всегда являлась бы точной копией своего источника за вычетом отставания репликации. Но в реальном мире из-за ошибок репликации данные на реплике и источнике могут рассинхронизироваться. Некоторыми возможными причинами являются:

- случайная запись в реплику;
- использование репликации с двумя источниками, когда оба источника выполняют запись;
- недетерминированные запросы и репликация на основе команд;
- аварийное завершение работы MySQL, когда вы работаете в режиме, не гарантирующем долговременное хранение данных (см. главу 5 о конфигурации долговечности);
- ошибки в MySQL.

Предлагаются следующие правила для обеспечения согласованности реплики с источником.

- *Всегда запускайте свои реплики с включенным параметром `super_read_only`.* Применение `read_only` не позволяет пользователям без привилегий `SUPER` выполнять запись, но это не мешает вашим администраторам баз данных выполнять команды `DELETE` или `ALTER`, не осознавая, что они находятся на реплике. Параметр `super_read_only` разрешает репликацию только для записи и является самым безопасным способом запуска реплик.
- *Используйте репликацию на основе строк или детерминированные операторы.* Несмотря на ситуации, когда требуется гораздо больше места на диске,

репликация на основе строк является наиболее согласованным способом репликации данных. Это связано с тем, что он включает точное изменение данных строки для каждой записи.

При репликации на основе команд примите во внимание следующее:

```
DELETE FROM users WHERE last_login_date <= NOW() LIMIT 10;
```

Что происходит, когда в этой таблице будет 1000 пользователей, соответствующих предложению `WHERE`? MySQL будет применять естественный порядок в таблице, чтобы удалить только первые десять строк. Естественный порядок таблицы может различаться в репликах, поэтому может быть затронут другой набор из десяти строк. Выполняемые в будущем команды изменяют или удаляют строки на основе `last_login_date`, которых может и не быть. Это может привести к несогласованности данных источника и реплики.

Лучший способ решить эту проблему — использовать `ORDER BY`, чтобы сделать порядок строк детерминированным:

```
DELETE FROM users WHERE last_login_date <= NOW() ORDER BY user_id  
LIMIT 10;
```

С помощью этой команды, пока данные согласованы между источником и репликой, будут удалены одни и те же десять строк.

- *В топологии репликации не пытайтесь одновременно выполнять запись на несколько серверов.* Это включает в себя использование источников с записью на обеих сторонах или кольцевую репликацию. Наиболее практичной топологией репликации является использование одного источника, выполняющего все операции записи, и одной или нескольких реплик, при необходимости выполняющих операции чтения.

Наконец, мы настоятельно рекомендуем при столкновении с какими-либо ошибками репликации для восстановления реплики задействовать стратегии, описанные в официальной документации MySQL.

Проблемы с репликацией и их решения

Простота реализации репликации MySQL, позволяющая так легко настраивать механизм, одновременно означает, что существует множество способов остановить, запутать или иным образом вывести ее из строя. Ранее в этой главе мы говорили об отказоустойчивой репликации и правилах, помогающих синхронизировать источник и реплики. В этом разделе обсуждаются распространенные проблемы — то, как они проявляются и как их можно решить или даже предотвратить.

Повреждение двоичных журналов в источнике

Если двоичный журнал поврежден в источнике, у вас не будет другого выхода, кроме как перестроить свои реплики. Пропуск поврежденной записи приведет к пропуску транзакций, которые больше не будут обрабатываться вашими репликами.

Неуникальные идентификаторы серверов

Это одна из самых трудноуловимых ошибок, с которыми вы можете столкнуться при репликации. Если вы случайно настроите две реплики с одним и тем же идентификатором сервера, может показаться, что они работают нормально, если следить невнимательно. Но если вы внимательно просмотрите их журналы ошибок или понаблюдаете за источником с помощью такого инструмента, как `innotop`, то заметите кое-что странное.

В источнике в каждый момент времени вы увидите только одну из двух подключенных реплик. (Обычно все реплики подключены одновременно и непрерывно занимаются репликацией.) В журнале ошибок на реплике вы увидите множество сообщений об ошибках отключения и повторного подключения, но не найдете упоминаний о неправильно сконфигурированном идентификаторе сервера.

В зависимости от версии MySQL реплики могут реплицировать данные правильно, но медленно или вовсе неправильно — любая конкретная реплика 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 без всяких сообщений. Возможны даже сбой или повреждение данных на источнике из-за повышенной нагрузки от реплик, борющихся между собой. И если борьба реплик становится особенно ожесточенной, то за очень короткое время журналы ошибок могут вырасти до необъятных размеров.

Единственное решение этой проблемы — быть очень внимательными при настройке реплик. Возможно, вам будет полезно вести список стандартных сопоставлений идентификаторов, присвоенных репликам, чтобы не потерять информацию о том, какой идентификатор назначен каждой реплике. Если все реплики находятся в пределах одной подсети, то в качестве идентификатора можно выбрать последний октет IP-адреса.

Неопределенные идентификаторы серверов

Если вы не укажете идентификатор сервера, MySQL настроит репликацию с опцией `CHANGE REPLICATION SOURCE TO`, но не позволит вам запустить реплику:

```
mysql> START REPLICA;  
ERROR 1200 (HY000): The server is not configured as replica;  
fix in config file  
or with CHANGE REPLICATION SOURCE TO
```

Эта ошибка вызывает особенно сильное недоумение, если вы только что выполнили команду `CHANGE REPLICATION SOURCE TO` и проверили свои настройки с помощью команды `SHOW REPLICA STATUS`. Команда `SELECT @@server_id` вернет какое-то значение, но это всего лишь значение по умолчанию. Вы должны задать идентификатор явно.

Отсутствующие временные таблицы

Временные таблицы очень удобны для некоторых задач, но, к сожалению, они несовместимы с репликацией на основе команд. В случае сбоя реплики или ее останова все временные таблицы, которые использовались в потоке репликации, внезапно исчезают. После перезапуска реплики любые последующие команды, которые ссылаются на отсутствующие временные таблицы, завершатся ошибкой.

Наилучшим подходом является применение репликации на основе строк. Следом за ним идет последовательное присвоение имен временным таблицам (например, с префиксом `temporary_`) и использование правил репликации, позволяющих полностью пропустить их репликацию.

Репликация не всех обновлений

Если вы неправильно применяете команду `SET SQL_LOG_BIN=0` или не понимаете правил фильтрации репликации, то ваша реплика может не выполнить обновления, которые произошли на источнике. Иногда именно это вам нужно для целей архивирования, но обычно все происходит непреднамеренно и имеет плачевные последствия.

Предположим, что у вас имеется правило `replicate_do_db`, которое разрешает реплицировать только базу данных `sakila` на одну из ваших реплик. Если будут выполнены следующие команды на источнике:

```
mysql> USE test;  
mysql> UPDATE sakila.actor ...
```

данные реплики рассинхронизируются с данными на источнике. Другие команды могут даже вызвать ошибку репликации из-за нереплицируемых зависимостей.

Слишком большое отставание репликации

Отставание репликации — распространенная проблема. В любом случае рекомендуется проектировать приложения таким образом, чтобы они сохраняли работоспособность при небольшом отставании реплик. Вот несколько распространенных подходов к уменьшению отставания репликации.

- *Многопоточная репликация.* Убедитесь, что вы используете многопоточную репликацию и рассмотрели настройку различных параметров в соответствии с руководством для того, чтобы получить от нее максимальную эффективность.
- *Использование сегментирования.* Хотя это кажется неожиданным, но применение методов сегментирования (sharding) для распределения записей по нескольким серверам оказывается очень эффективной стратегией. Для MySQL существует давнее эмпирическое правило: масштабирование чтения с помощью реплик, масштабирование записи с помощью сегментирования.
- *Временное снижение долговечности.* Пуристы с этим не согласятся, но возможны случаи, когда вы исчерпали все возможности настройки и подстройки и сегментирование оказывается нежизнеспособным вариантом либо из-за необходимости приложить значительные усилия, либо из-за проблем с дизайном. Если отставание репликации в основном связано с ограничениями операций записи, можете временно установить `sync_binlog=0` и `innodb_flush_log_at_trx_commit=0`, чтобы увеличить скорость репликации.

Выбрав этот последний путь, вы должны быть очень и очень осторожны. Делайте это только на своей реплике, и если она тоже является местом, где вы создаете резервные копии, изменение перечисленных параметров может сделать невозможным восстановление из резервной копии¹. Кроме того, если реплика выйдет из строя при временном снижении долговечности, вам, вероятно, придется восстанавливать ее из источника. Наконец, если вы сделаете это вручную, очень легко забыть снова настроить долговечность. Убедитесь, что у вас есть хороший мониторинг или какой-то скрипт для восстановления долговечности.

Одной из возможных стратегий может быть просмотр значения `Seconds_behind_source`, полученного с помощью команды `SHOW REPLICA STATUS`, и, когда оно превышает определенное значение, запуск действия, которое должно:

- убедиться, что сервер является репликой, недоступной для записи, возможно проверив, включен ли параметр `super_read_only`;
- изменить настройки `sync_binlog` и `innodb_flush_log_at_trx_commit`, чтобы уменьшить количество операций записи;

¹ Обычно это верно в тех случаях, когда вы можете использовать моментальные снимки LVM или облачный подход к резервному копированию моментальных снимков диска.

- периодически проверять значение `Seconds_behind_source` с помощью команды `SHOW REPLICA STATUS`;
- когда значение ниже приемлемого порога, вернуть настройки к их постоянному состоянию.

Чрезмерно большие пакеты от источника

Еще одна трудноотслеживаемая проблема репликации может возникнуть, когда параметры `max_allowed_packet` на источнике и реплике не совпадают. В этом случае источник может поместить в журнал пакет, который реплика считает слишком большим, и, когда реплика получит это двоичное событие журнала, у нее могут возникнуть разного рода сбои. Это может, в частности, вызвать бесконечный цикл ошибок и повторных попыток чтения или повредить журнал ретрансляции.

Отсутствие места на диске

Репликация действительно может заполнить ваши диски двоичными журналами, журналами ретрансляции или временными файлами, особенно если вы выполняете много команд `LOAD DATA INFILE` на источнике, а на реплике включен режим `log_replica_updates`. Чем сильнее отстает реплика, тем больше места на диске она может использовать для журналов ретрансляции, которые уже были прочитаны с источника, но еще не обработаны. Вы можете предотвратить эти ошибки, отслеживая свободное место на дисках и устанавливая параметр конфигурации `relay_log_space`.

Ограничения репликации

Репликация MySQL может допустить сбой или привести к рассинхронизации (сообщения об ошибке может и не появиться) просто из-за присущих ей ограничений. Существует довольно много функций SQL и приемов программирования, которые невозможно надежно реплицировать (многие из них мы упоминали в этой главе). Трудно гарантировать, что ничего из их списка не попадет в код промышленной системы, особенно если приложение большое или им занимается значительный коллектив разработчиков.

Немало неприятностей причиняют и ошибки в коде сервера. Мы не хотим показаться критиканами, но исторически многие основные версии сервера MySQL содержали ошибки в коде репликации, особенно в первых выпусках. А появление новых возможностей, в частности хранимых процедур, приводило к новым ошибкам.

Для большинства пользователей это не повод избегать новых функций. Это просто повод для тщательного тестирования, особенно при переходе на новую версию приложения или MySQL. Мониторинг тоже важен — необходимо знать, когда что-то вызывает проблему.

Репликация MySQL устроена сложно, и чем сложнее ваше приложение, тем тщательнее должно быть тестирование. Однако, если понять, как пользоваться репликацией, она будет работать хорошо.

Резюме

Репликация — это швейцарский армейский нож среди встроенных возможностей MySQL, она значительно расширяет диапазон функциональных возможностей и увеличивает полезность MySQL. Возможно, это одна из ключевых причин такого быстрого роста популярности MySQL.

Хотя репликация имеет много ограничений и оговорок, оказывается, что большинство из них не слишком важны или их легко избежать. Многие из недостатков — это просто особое поведение расширенных возможностей, которые основная масса людей не будет применять, но для меньшинства пользователей они нужны и очень полезны.

Когда дело доходит до репликации, вашим девизом должен быть K.I.S.S. (Keep It Short and Simple — «Пусть будет коротко и ясно»). Не делайте ничего необычного, например не используйте кольца репликации или фильтры репликации, если не уверены, что это действительно вам необходимо. Задействуйте репликацию лишь для зеркалирования полной копии ваших данных, включая все привилегии. Поддержание реплик полностью идентичными источнику во всех отношениях поможет вам избежать многих проблем.

Резервное копирование и восстановление

Если вы не запланировали резервное копирование заранее, позже можете обнаружить, что исключили некоторые из лучших вариантов. Например, вы могли сконфигурировать сервер, а затем осознали, что надо бы воспользоваться LVM для создания мгновенных снимков файловой системы, но было уже слишком поздно. Вы также можете выяснить некоторые важные последствия для производительности при настройке систем для резервного копирования. 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.

В этой главе мы не будем рассматривать все компоненты хорошо спроектированного решения для резервного копирования и восстановления — только те части, которые имеют отношение к MySQL. Вот несколько тем, которые мы решили не включать в нее, но которые вы должны ввести в свою общую стратегию резервного копирования и восстановления.

- Безопасность (доступ к резервному копированию, привилегии для восстановления данных и необходимость шифрования файлов).
- Где хранить резервные копии, в том числе как далеко они должны находиться от источника (на другом диске, другом сервере или вне площадки), и как перемещать данные из источника в пункт назначения.
- Политика хранения, аудит, требования законодательства и связанные с этим темы.
- Системы хранения и носители, сжатие и инкрементное копирование.
- Форматы хранения данных.
- Мониторинг и отчетность по резервным копиям.
- Возможности резервного копирования, встроенные в уровни хранения или специальные устройства, такие как готовые аппаратные файловые серверы.

Прежде всего уточним некоторые ключевые термины. Во-первых, вы часто будете слышать о так называемых горячих, теплых и холодных резервных копиях. Обычно эти термины используются для обозначения влияния резервного копирования: например, горячее резервное копирование означает, что сервер не нужно останавливать. Проблема в том, что не все их одинаково понимают. В названиях некоторых инструментов даже присутствует слово *hot* («горячий»), хотя они определенно не выполняют то, что мы считаем горячим резервным копированием. Постараемся избегать этих терминов, а вместо этого будем говорить о том, насколько конкретный метод или инструмент прерывает работу вашего сервера.

Два других сбивающих с толку слова — это «возвращать» (*restore*) и «восстанавливать» (*recover*). В этой главе им придается вполне конкретный смысл. Возврат означает извлечение данных из резервной копии и либо загрузку их в MySQL, либо размещение файлов там, где MySQL ожидает их найти. Восстановление обычно означает весь процесс приведения системы или ее части в рабочее состояние после того, как что-то пошло не так. Сюда входят не только возврат данных из резервных копий, но и прочие шаги, необходимые для возобновления функционирования сервера в полном объеме, в частности перезапуск MySQL, изменение конфигурации, прогрев кэшей сервера и т. д.

Для многих восстановление означает просто исправление поврежденных в результате сбоя таблиц. Но это совсем не то же самое, что восстановление всего сервера. Процедура восстановления после сбоя подсистемы хранения должна привести в соответствие его файлы данных и журналов. Она также должна гарантировать, что файлы данных содержат только модификации, произведенные закоммиченными транзакциями, и повторить те транзакции из файлов журналов, которые еще не были применены к файлам данных. Это может быть частью общего процесса восстановления или даже резервного копирования. Однако это не то же самое, что восстановление, которое вам может понадобиться, например, после случайного выполнения команды `DROP TABLE`. Действия, которые вы предпринимаете для восстановления, могут сильно различаться в зависимости от проблемы, которая решается при восстановлении. Наконец, существуют два основных типа резервных копий: сырые и логические. Сырые резервные копии, иногда называемые физическими¹ резервными копиями, относятся к копиям файлов из файловой системы. Логические резервные копии относятся к операциям SQL, необходимым для восстановления данных.

¹ Сырые резервные копии также могут не интуитивно называться физическими резервными копиями, поскольку предполагается, что вы перемещаете физические файлы в место назначения для резервного копирования. Мы говорим «не интуитивно», потому что сам файл вообще не является физическим!

Зачем нужно резервное копирование

Приведем несколько причин необходимости резервного копирования.

- *Аварийное восстановление.* Это то, что требуется сделать, когда аппаратное обеспечение выходит из строя, неприятная ошибка повреждает данные или сервер и его данные становятся недоступными или непригодными для использования по какой-либо другой причине. Вы должны быть готовы ко всему, от случайного подключения к неправильному серверу, выполняющему ALTER TABLE, до сгоревшего здания, действий злоумышленника или ошибки MySQL. И хотя вероятность того, что произойдет какое-либо конкретное стихийное бедствие, не очень высока, эти катаклизмы имеют тенденцию происходить одновременно.
- *Изменение решения.* Вы бы удивились, узнав, как часто люди намеренно удаляют данные, а затем хотят их восстановить.
- *Аудит.* Иногда требуется знать, как выглядели ваши данные или схема в какой-то момент времени в прошлом. Например, вы можете быть вовлечены в судебный процесс или обнаружить ошибку в приложении и вам нужно понять, что было написано в этом коде раньше (иногда того, что код находится в системе контроля версий, недостаточно).
- *Тестирование.* Один из самых простых способов тестирования — периодическое обновление тестового сервера с использованием последних производственных данных. Если вы делаете резервные копии, это просто — нужно лишь восстановить резервную копию на тестовом сервере.

Проверьте свои предположения. Как думаете, ваш хостинг-провайдер поддерживает резервное копирование сервера MySQL, предоставляемого вместе с вашей учетной записью? Ответ может вас удивить. Многие хостинг-провайдеры вообще не делают резервных копий серверов MySQL, а другие просто копируют файлы во время работы сервера, что, вероятно, создает поврежденную резервную копию, которая может оказаться бесполезной.

Определение требований к восстановлению

Пока все идет хорошо, вам не придется задумываться о восстановлении. Но, когда такой момент настанет, даже самая лучшая в мире система резервного копирования не спасет, если вы никогда не проверяли восстановление резервной копии. Вам понадобится отличная система восстановления.

Проблема в том, что проще организовать бесперебойную работу систем резервного копирования, чем создать хорошие процедуры и инструменты восстановления. И вот почему.

- Резервное копирование происходит раньше. Если вы не создали резервной копии, то восстанавливать будет не с чего, поэтому при построении системы ваше внимание, естественно, будет сосредоточено на резервных копиях.
- Резервное копирование автоматизировано с помощью скриптов и заданий. Поэтому вы думаете главным образом о том, как автоматизировать и настроить именно процесс резервного копирования, часто даже не отдавая себе в этом отчета. Пять минут в день на улучшение процедуры копирования могут показаться неважными, но уделяете ли вы ежедневно такое же внимание восстановлению?
- Снятие резервной копии обычно выполняется без всякого стресса, но восстановление, как правило, происходит в кризисной ситуации.
- Часто в дело вступают соображения безопасности. Если резервное копирование производится за пределами рабочей площадки, то вы, скорее всего, шифруете данные резервной копии или принимаете другие меры для их защиты. Очень легко рассуждать об ущербе, который нанесет компрометация данных, и совсем упустить из виду то, что произойдет, когда никто не сможет прочесть зашифрованный том, чтобы восстановить с него ваши 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. Но он может быть недоступен, когда произойдет бедствие. Вам необходимо обучить нескольких человек и позаботиться о том, чтобы кто-нибудь из них постоянно присутствовал на работе, чтобы не поручать неквалифицированному сотруднику восстанавливать данные.

Существует два важных требования, которые могут оказаться полезными при планировании стратегии резервного копирования и восстановления. Это *целевая точка восстановления* (recovery point objective, RPO) и *целевое время восстановления* (recovery time objective, RTO). Можно заметить, что это звучит очень похоже на SLO, которые мы обсуждали в главе 2. Они определяют, сколько данных вы готовы потерять и как долго готовы ждать их восстановления. При определении RPO и RTO попытайтесь ответить на следующие типы вопросов.

- Сколько данных вы можете потерять без серьезных последствий? Потребуется ли вам восстановление на определенный момент времени в прошлом, или допустима потеря всей работы, выполненной с момента последнего

регулярного резервного копирования? Существуют ли какие-то законодательные требования?

- Насколько быстрым должно быть восстановление? Какое время простоя допустимо? С чем будут готовы смириться приложение и пользователи (например, с частичной недоступностью) и как вы собираетесь обеспечить возможность продолжения функционирования в таких условиях?
- Что вам необходимо восстанавливать? Обычно требования заключаются в восстановлении всего сервера, отдельной базы данных, одной таблицы либо только конкретных транзакций или команд.

Рекомендуется задокументировать ответы на эти вопросы, всю вашу политику резервного копирования, а также процедуры резервного копирования.

МИФ О РЕЗЕРВНОМ КОПИРОВАНИИ № 1. ИСПОЛЬЗУЮ РЕПЛИКАЦИЮ КАК РЕЗЕРВНОЕ КОПИРОВАНИЕ

Эта ошибка встречалась нам довольно часто. Реплика не может заменить резервную копию. И RAID-массив тоже. Чтобы понять почему, задумайтесь над таким вопросом: помогут ли они восстановить все ваши данные, если вы случайно выполните команду `DROP DATABASE` для промышленной базы данных? Ни RAID-массив, ни репликация не проходят даже этот простой тест. Их нельзя относить к технологиям резервного копирования. Они даже не могут заменить резервную копию! Ничто, кроме самого резервного копирования, не способно реализовать функции резервного копирования.

Проектирование решения для резервного копирования MySQL

Резервное копирование MySQL сложнее, чем кажется. Прежде всего, резервная копия — это просто копия данных, но потребности вашего приложения, архитектура подсистемы хранения MySQL и конфигурация системы могут затруднить ее создание.

Прежде чем подробно рассматривать все существующие варианты, хотим дать некоторые рекомендации.

- Для больших баз данных физическое резервное копирование просто необходимо: логическое копирование выполняется очень медленно, потребляет много ресурсов, а восстановление из логической резервной копии занимает слишком много времени. Резервное копирование на основе снимков с использованием Percona XtraBackup и MySQL Enterprise Backup — лучшие

варианты. Для небольших баз данных логическое резервное копирование может хорошо работать.

- Сохраняйте несколько поколений резервных копий.
- Периодически создавайте логические резервные копии (возможно, из физических резервных копий).
- Храните двоичные журналы для восстановления на определенный момент времени. Установите довольно большое значение параметра `expire_logs_days`, достаточное для восстановления как минимум из двух поколений физических резервных копий. Должна существовать возможность создать реплику и запустить ее из работающего источника без применения к ней двоичных журналов. Создавайте резервные копии двоичных журналов, не зависящие от заданного срока действия, и храните их в резервной копии достаточно долго, чтобы можно было выполнить восстановление по крайней мере из самой последней логической резервной копии.
- Ведите мониторинг процесса резервного копирования и самих резервных копий независимо от инструментов резервного копирования. Необходимо независимое подтверждение их пригодности.
- Периодически проверяйте процедуру резервного копирования и восстановления, выполнив процесс восстановления с начала до конца. Измеряйте ресурсы, необходимые для восстановления (процессор, объем диска, физическое время, пропускная способность сети и т. д.).
- Не упускайте из виду безопасность. Если кто-то скомпрометирует ваш сервер, сможет ли он получить доступ и к серверу резервного копирования, и наоборот?

Знание RPO и RTO станет определяющим при выработке вами стратегии резервного копирования. Нужна ли вам возможность восстановления баз на определенный момент времени, или достаточно восстановить данные с копии, снятой прошлой ночью, смирившись с потерей всей проделанной с тех пор работы? Если необходимо восстановление на определенный момент, вы, вероятно, можете выполнять регулярное резервное копирование и включить режим записи в двоичный журнал, чтобы иметь возможность восстановить эту резервную копию, а затем воспроизвести двоичный журнал до нужного момента времени.

Как правило, чем больше данных вы можете позволить себе потерять, тем проще процедура резервного копирования. Если у вас очень жесткие требования, то гораздо сложнее гарантировать, что получится восстановить всю информацию. Существуют также различные варианты восстановления на определенный момент времени. Мягкие требования восстановления на определенный момент времени означают, что вы хотели бы иметь возможность воссоздать свои данные,

чтобы они были довольно близки к тому состоянию, в котором были к моменту сбоя. Жесткое требование подразумевает необходимость восстановить все зафиксированные транзакции, даже если произошло что-то ужасное, например, сервер в буквальном смысле сгорел. Для этого применяются специальные методы, такие как хранение двоичного журнала в отдельном SAN-томе или использование репликации диска с распределенным реплицированным блочным устройством (DRBD).

Оперативное или автономное резервное копирование?

Остановить сервер MySQL на время резервного копирования, если это допустимо, — это самый простой, безопасный и в целом лучший способ получить непротиворечивую копию данных с минимальным риском искажения или несогласованности. Если вы выключите MySQL, то сможете скопировать данные, не заботясь о таких неприятностях, как грязные буферы в буферном пуле InnoDB или в других кэшах. Вам не нужно беспокоиться о том, что данные будут модифицироваться в процессе резервного копирования, а поскольку нет никакой нагрузки на сервер со стороны приложения, можно сделать резервную копию быстрее.

Однако перевод сервера в автономный режим более затратен, чем может показаться. В результате вам почти наверняка потребуется спроектировать создание резервных копий так, чтобы они не требовали отключения рабочего сервера. Однако в зависимости от требований к согласованности создание резервной копии во время работы сервера может привести к значительным прерываниям в работе.

Перечислим факторы, связанные с производительностью, которые следует учитывать при планировании резервного копирования.

- *Время резервного копирования.* Сколько времени занимает создание резервной копии и ее копирование в пункт назначения?
- *Резервная загрузка.* Как сильно копирование резервной копии в место назначения влияет на производительность сервера?
- *Время восстановления.* Сколько времени занимает копирование образа резервной копии из хранилища на сервер MySQL, повторное использование двоичных журналов и т. д.?

Самое главное — найти компромисс между временем резервного копирования и временем восстановления. Как правило, один показатель можно улучшить за счет другого: например, вы можете повысить приоритет резервного копирования за счет дальнейшего снижения производительности сервера.

Можно спроектировать создание резервной копии так, чтобы использовались преимущества паттернов загрузки. Например, если сервер загружен только на 50 % в течение 8 ч в ночное время, можете попытаться спроектировать резервное копирование так, чтобы загрузка сервера составляла менее 50 % и весь процесс выполнялся в течение 8 ч. Сделать это можно разными способами: например, применять утилиты `ionice` и `nice`, чтобы отдать предпочтение операциям копирования или сжатия, использовать разные уровни сжатия или сжимать данные на сервере резервного копирования вместо сервера MySQL. Можно также использовать утилиты `lzo` или `pigz` для более быстрого сжатия. Вы можете взять функции `O_DIRECT` или `fsync()`, чтобы не задействовать кэш операционной системы для операций копирования, чтобы они не загрязняли кэши сервера. Такие инструменты, как Percona XtraBackup и MySQL Enterprise Backup, имеют параметры пропуска тактов, и вы можете применить утилиту `pv` с параметром `--rate-limit`, чтобы ограничить пропускную способность написанных вами скриптов.

Логическое и физическое резервное копирование

Как упоминалось ранее, существует два основных способа резервного копирования данных MySQL: с помощью логического резервного копирования (также называемого дампом) и копирования исходных файлов. В логической резервной копии данные представлены в формате, который MySQL может интерпретировать либо как SQL-команды, либо как текст с разделителями¹. Исходные файлы — это просто файлы в том виде, в каком они находятся на диске.

У каждого способа резервного копирования данных имеются свои преимущества и недостатки.

Логические резервные копии

Логические резервные копии имеют следующие преимущества.

- Это обычные файлы, которые можно обрабатывать и просматривать с помощью редакторов и инструментов командной строки, таких как `grep` и `sed`. Это может быть очень полезно при восстановлении данных или когда вы просто хотите просмотреть их, не восстанавливая.

¹ Логические резервные копии, созданные `mysqldump`, — это не всегда текстовые файлы. SQL-дамп может содержать текст в различных кодировках и даже двоичные данные, которые по определению не соответствуют печатаемым символам. Да и строки могут быть 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`.

- Из них легко восстанавливать данные. Вы можете просто передать файл по конвейеру на вход программы `mysql` или воспользоваться программой `mysqlimport`.
- Можно выполнять резервное копирование и восстановление по сети, то есть не на той же машине, где работает сервер MySQL.
- Они могут работать с облачными системами MySQL, где у вас нет доступа к базовой файловой системе.
- Процедуру можно очень гибко настраивать, потому что `mysqldump` — инструмент, который большинство людей предпочитают использовать для снятия логических копий, — может принимать множество параметров, например раздел `WHERE`, позволяющий 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 нивелируются¹.
- Они могут помочь избежать повреждения данных. Если ваши диски сбоят, то при копировании физических файлов вы получите сообщение об ошибке и/или сделаете частичную или поврежденную резервную копию. И если специально не проверите ее по окончании резервного копирования, то узнаете об этом только тогда, когда не сможете воспользоваться ею для восстановления. Если данные, хранящиеся в памяти MySQL, не повреждены, то иногда вы можете получить заслуживающую доверия логическую резервную копию, когда не удастся получить хорошую физическую копию файла.

Есть у логических резервных копий и недостатки.

- Для их создания требуется, чтобы сервер работал, поэтому процессор оказывается сильнее загруженным.
- В некоторых случаях логические резервные копии могут быть больше, чем исходные физические файлы². Представление данных в формате ASCII не всегда так же эффективно, как внутреннее представление в подсистеме хранения. Например, для хранения целого числа требуется 4 байта, но при его записи в виде ASCII-текста может потребоваться до 12 символов. Зачастую логические копии хорошо поддаются сжатию и можно получить меньшую

¹ Имейте в виду, что, хотя данные, которые выгружаются, не зависят от механизма, функции механизма хранения могут быть несовместимы. Например, вы не можете сделать дампы базы данных InnoDB с определенными отношениями внешних ключей и ожидать, что внешние ключи будут работать в движке, который их не реализует.

² По нашему опыту, логические резервные копии обычно меньше необработанных, но не всегда.

резервную копию, но при этом используется дополнительное процессорное время, что увеличивает время восстановления. (Если применяется много индексов, то логические резервные копии обычно меньше, чем исходные резервные копии.)

- Копирование и восстановление данных не всегда гарантируют их получение в исходном виде. Утрата точности в представлении чисел с плавающей точкой, ошибки и т. п. могут вызывать проблемы, хоть это и случается редко.
- Для восстановления данных из логической резервной копии MySQL необходимо загружать и интерпретировать команды, преобразовывать их в формат хранения и перестраивать индексы. Все это происходит очень медленно.

Самый большой недостаток — затраты на выгрузку данных из MySQL и обратную их загрузку с помощью команд SQL. Если вы используете логическое резервное копирование, важно проверить время, необходимое для восстановления данных.

Физические резервные копии

Физические резервные копии имеют следующие преимущества.

- Для получения физической копии следует просто скопировать нужные файлы в другое место. Никакой дополнительной работы для создания физической резервной копии не требуется.
- Физические резервные копии легко переносятся между платформами, операционными системами и версиями MySQL. (Логические дампы тоже. Мы указываем на это, чтобы исключить любые проблемы, которые у вас могут возникнуть.)
- Восстановление физических резервных копий может происходить быстрее, потому что серверу MySQL не нужно выполнять какие-либо SQL-команды или создавать индексы. Если у вас есть таблицы InnoDB, которые не помещаются полностью в память сервера, восстановление данных из физических файлов можно выполнить намного быстрее — на порядок или даже больше. На самом деле одной из самых неприятных вещей в логических резервных копиях является непредсказуемое время восстановления.

Вот некоторые недостатки физических резервных копий.

- Физические файлы InnoDB, как правило, намного больше, чем соответствующие логические резервные копии. Табличное пространство InnoDB обычно имеет много неиспользуемого пространства. К тому же какое-то пространство

задействуется для целей, не связанных с хранением табличных данных (буфер вставки, сегмент отката и т. д.).

- Физические резервные копии не всегда переносимы между платформами, операционными системами и версиями MySQL. В частности, препятствием могут стать чувствительность к регистру имени файла и формат чисел с плавающей точкой — это места, где вы можете столкнуться с проблемами. Возможно, вы не сможете переместить файлы в систему с другим форматом с плавающей точкой (впрочем, в большинстве современных процессоров используется формат чисел с плавающей точкой IEEE).

Работать с физическими резервными копиями, как правило, проще и намного эффективнее¹. Однако вы не должны целиком полагаться на физические резервные копии при необходимости долгосрочного хранения или соответствия требованиям законодательства, время от времени нужно делать и логические резервные копии.

Не считайте резервную копию (особенно физическую) пригодной для использования, пока не протестируете ее. Для InnoDB это означает, что нужно запустить экземпляр MySQL, дать InnoDB завершить процедуру восстановления, а затем выполнить команду `CHECK TABLES`. Вы можете пропустить этот шаг или просто проверить файлы с помощью утилиты `innochecksum`, но мы не рекомендуем так делать.

Стоит комбинировать оба подхода: сделайте физические копии, а затем запустите экземпляр сервера MySQL с полученными данными и утилиту `mysqlcheck`. Затем хотя бы периодически выполняйте дампы данных с помощью утилиты `mysqldump` для получения логической резервной копии. В результате вы получите преимущества обоих подходов, не создавая на промышленном сервере излишней нагрузки по формированию дампа. Это особенно удобно, если у вас есть возможность делать мгновенные снимки файловой системы: можно сделать снимок, скопировать на другой сервер и развернуть его, а затем проверить физические файлы и выполнить логическое резервное копирование.

Что нужно копировать

Ваши требования к восстановлению будут определять, что нужно включать в резервную копию. Самая простая стратегия — просто скопировать данные и определить таблицы, однако это лишь необходимый минимум. Как правило, для полного восстановления промышленного сервера вам требуется гораздо

¹ Стоит отметить, что для необработанных резервных копий больше риск возникновения ошибок: трудно превзойти простоту `mysqldump`.

больше. Перечислим кое-что из того, что вы могли бы включить в свои резервные копии MySQL.

- *Неочевидные данные.* Не забывайте о данных, которые не бросаются в глаза, например о двоичных журналах и журналах транзакций InnoDB. В идеале вы должны сделать резервную копию всего каталога данных для MySQL.
- *Код.* В современном сервере MySQL может содержаться большой объем программного кода, такого как триггеры и хранимые процедуры. Если вы выполняете резервное копирование базы данных `mysql`, то большая часть этого кода войдет в резервную копию. Однако тогда будет сложно полностью восстановить единственную базу из всего множества, поскольку часть данных в этой базе данных, к примеру хранимые процедуры, на самом деле содержатся в базе `mysql`.
- *Конфигурация сервера.* Если вам потребуется восстановить данные после настоящей аварии — скажем, вы строите сервер с нуля в новом центре обработки данных после землетрясения, — полезно будет включить конфигурационные файлы сервера в состав резервной копии.
- *Отдельные файлы операционной системы.* Как и в случае с конфигурацией сервера, очень важно создать резервную копию всех внешних конфигурационных файлов, необходимых для работы операционной системы сервера. На Unix-сервере это могут быть ваши таблицы заданий `cron`, конфигурации пользователей и групп, административные сценарии и правила `sudo`.

Подобные рекомендации во многих сценариях быстро превращаются в совет: «Копируйте все». Однако, если информации очень много, это может оказаться слишком дорогим удовольствием и, возможно, придется более разумно подходить к резервному копированию. В частности, вы можете захотеть создать резервную копию разных данных в различных резервных копиях. Например, можно создавать по отдельности резервные копии данных, двоичных журналов, конфигурационных файлов операционной системы и сервера.

Инкрементное и дифференциальное резервное копирование

При наличии большого объема данных общепринятой стратегией работы является регулярное выполнение инкрементного или дифференциального копирования. Разница между ними может немного сбивать с толку, поэтому уточним термины: *дифференциальная резервная копия* — это резервная копия всего, что изменилось с момента создания последней полной резервной копии, тогда как *инкрементная резервная копия* содержит все, что изменилось с момента выполнения последней резервной копии любого типа.

Предположим, что вы делаете полное резервное копирование каждое воскресенье. В понедельник создаете дифференциальную резервную копию всего, что изменилось с воскресенья. Во вторник у вас есть два варианта: можно создать резервную копию всего, что изменилось с воскресенья (дифференциальная), или резервную копию только тех данных, которые изменились с момента резервного копирования в понедельник (инкрементная).

Как дифференциальные, так и инкрементные резервные копии — это частичные резервные копии: как правило, они не содержат полного набора данных, поскольку некоторые данные почти наверняка не изменились. Частичные резервные копии зачастую стоит создавать при желании сэкономить накладные расходы на сервере и уменьшить время резервного копирования и пространство для резервного копирования. Однако некоторые частичные резервные копии на самом деле не уменьшают нагрузку на сервер. Percona XtraBackup и MySQL Enterprise Backup, например, по-прежнему сканируют каждый блок данных на сервере, поэтому они не очень уменьшают накладные расходы, хотя и экономят немного времени работы, много процессорного времени для сжатия и, конечно же, дисковое пространство¹.

Вы можете использовать продвинутые современные методы резервного копирования, но чем сложнее ваше решение, тем более рискованным оно может быть. Остерегайтесь скрытых опасностей, таких как создание множества поколений резервных копий, которые тесно связаны друг с другом, потому что, если одно поколение содержит ошибку, оно может сделать недействительными все остальные. Поделимся некоторыми продвинутыми идеями резервного копирования.

- Используйте функции инкрементного резервного копирования Percona XtraBackup или MySQL Enterprise Backup.
- Делайте резервную копию своих двоичных журналов. Вы также можете использовать `FLUSH LOGS`, чтобы начинать новый двоичный журнал после каждого резервного копирования, а затем создавать резервные копии только новых двоичных журналов.
- Если у вас есть справочные таблицы, содержащие такие данные, как списки названий месяцев на разных языках или аббревиатуры для штатов или регионов, имеет смысл поместить их в отдельную базу данных, чтобы не приходилось каждый раз их копировать. Еще лучше было бы поместить их в код, а не в базу данных.

¹ Функция инкрементного резервного копирования для Percona XtraBackup находится в разработке. Она сможет создавать резервные копии блоков, которые были изменены, без необходимости сканирования каждого блока.

- Не создавайте резервные копии неизменных строк. Если таблица предназначена только для добавления данных командой `INSERT`, как, например таблица, которая регистрирует обращения к веб-странице, вы можете добавить в нее столбец `TIMESTAMP` и включать в резервную копию только строки, вставленные с момента последнего резервного копирования. Лучше всего это работает в сочетании с `mysqldump`.
- Не создавайте резервные копии некоторых данных. Иногда это вполне оправданно. Например, если у вас есть хранилище данных, которое создается из других данных и, строго говоря, является избыточным, то вы можете просто создать резервную копию данных, использованных для создания хранилища, а само его не копировать. Это может быть здоровой идеей, даже если воссоздавать хранилище из исходных данных очень долго. Отказ от копирования всех данных может со временем принести намного бóльшую экономию, чем удалось бы получить при наличии полной копии. Вы также можете отказаться от резервного копирования некоторых временных данных, таких как таблицы, содержащие данные о сеансах веб-сайта.
- Создавайте резервные копии всего, чего только можно, но позаботьтесь, чтобы они копировались туда, где есть возможность устранения избыточности данных, например в файловую систему ZFS.

К недостаткам инкрементного резервного копирования относятся повышенная сложность восстановления, увеличенный риск и более длительное время восстановления. Если есть возможность делать полные резервные копии, то рекомендуем ею воспользоваться, чтобы упростить работу.

В любом случае время от времени вам определенно нужно делать полные резервные копии, советуем делать это по крайней мере еженедельно. Вряд ли стоит рассчитывать на то, что можно будет восстановиться, имея только инкрементные резервные копии за месяц. Даже неделя — это трудоемко и рискованно.

Репликация

Самым большим преимуществом резервного копирования на реплике является то, что оно не прерывает работу источника и не оказывает на него дополнительной нагрузки. Это само по себе серьезная причина для организации сервера-реплики, даже если он вам не нужен с целью балансировки нагрузки или обеспечения высокой доступности. Если ваш бюджет ограничен, вы всегда можете использовать сервер резервного копирования для других целей, например для генерации отчетов, при условии, что не будете производить на нем никаких операций записи и, таким образом, не измените данные, для которых пытаетесь создать резервную копию. Реплика не обязательно должна быть целиком выделена только для резервного копирования, важно лишь, чтобы она успевала

догнать источник к моменту снятия следующей резервной копии, если другие ее роли время от времени заставляют отставать в репликации.

Когда вы делаете резервную копию на реплике, очень разумно использовать `GTID`, как упоминалось в главе 9. Это позволяет избежать необходимости сохранять всю информацию о процессах репликации, такую как позицию реплики относительно источника. Это полезно для клонирования новых реплик, повторного применения двоичных журналов к источнику с целью восстановления на определенный момент времени, повышения реплики до уровня источника и т. д. Кроме того, убедитесь, что в момент останова реплики нет открытых временных таблиц, потому что они могут помешать вам возобновить репликацию.

Как упоминалось в подразделе «Отложенная репликация» в главе 9, преднамеренное отставание репликации на одной из ваших реплик может быть очень полезно для восстановления после некоторых аварийных сценариев. Предположим, вы производите репликацию с паузой 1 ч. Если на источнике была выполнена нежелательная команда, у вас есть час, чтобы заметить это и остановить реплику, прежде чем она воспроизведет это событие из своего журнала ретрансляции. Затем вы можете повысить реплику до источника и выполнить несколько событий из журнала, пропустив неверные операторы. Это может оказаться намного быстрее, чем применять технику восстановления на конкретный момент, которую мы обсудим чуть позже.



Данные на реплике и источнике могут различаться. Часто думают, что реплики являются точными копиями своего источника, но наш опыт показывает, что расхождения в данных в репликах — обычное явление и MySQL не имеет возможности обнаружить эту проблему. Единственный способ найти расхождения — применить такой инструмент, как `pt-table-checksum` Percona Toolkit. Лучший способ предотвратить такие расхождения — использовать флаг `super_read_only`, чтобы гарантировать, что только репликация может записывать в реплики.

Наличие реплицированной копии данных может помочь вам застраховаться от таких проблем, как выход из строя диска на источнике, но никаких гарантий нет. Репликация — это не резервное копирование.

Управление двоичными журналами и их резервное копирование

Двоичные журналы вашего сервера — одна из самых важных вещей, которые следует включать в резервную копию. Они совершенно необходимы для восстановления на определенный момент времени, и поскольку их размер обычно меньше, чем сами данные, то копировать их можно чаще. Если у вас есть резервная копия данных на какое-то время и все двоичные журналы, накопившиеся

с этого момента, вы можете воспроизвести двоичные журналы и накатить изменения, произошедшие с момента создания последней полной резервной копии.

MySQL также использует двоичный журнал для репликации. Это означает, что ваша стратегия резервного копирования и восстановления тесно связана с конфигурацией репликации. Рекомендуется почаще создавать резервные копии двоичных журналов. Если потеря данных более чем за 30 мин недопустима, создавайте их резервные копии по крайней мере каждые 30 мин.

Вам нужно выбрать политику истечения срока действия журнала, чтобы MySQL не заполнял диск двоичными журналами. Размер журналов зависит от вашей рабочей нагрузки и формата ведения журнала (ведение журнала на основе строк приводит к увеличению размера записей в журнале). Мы рекомендуем хранить журналы до тех пор, пока они будут полезны, если это возможно. Их хранение полезно для настройки реплик, анализа рабочей нагрузки вашего сервера, аудита и восстановления на определенный момент времени из последней полной резервной копии. Учитывайте все эти потребности, когда будете решать, как долго хранить журналы.

Обычная настройка заключается в использовании переменной `binlog_expire_logs_seconds`, чтобы сообщить MySQL о необходимости очистки журналов через некоторое время. Не следует удалять эти файлы вручную.

Параметр `binlog_expire_logs_seconds` вступает в силу при запуске сервера или когда MySQL выполняет ротацию двоичного журнала, поэтому, если ваш двоичный журнал никогда не заполняется и не ротируется, сервер не будет удалять старые записи. Он решает, какие файлы следует удалить, просматривая время их модификации, а не их содержимое.

Инструменты резервного копирования и восстановления

Сейчас существует множество хороших и не очень хороших инструментов резервного копирования. Изготавливать физические резервные копии мы рекомендуем с помощью Percona XtraBackup. Это утилита для горячего резервного копирования баз данных MySQL с открытым исходным кодом, широко используемая и хорошо документированная. Для логических резервных копий мы предпочитаем утилиту `mysqldumper`. Хотя `mysqldump` поставляется с MySQL из коробки, его однопоточная природа может привести к очень длительному времени резервного копирования и восстановления. `mysqldumper` имеет встроенную многопоточность (параллелизм), которая может значительно ускорить получение логической резервной копии.

MySQL Enterprise Backup

Этот инструмент является частью подписки MySQL Enterprise от Oracle. Его использование не требует остановки MySQL, установки блокировок или прерывания нормальной работы базы данных, хотя и вызовет некоторую дополнительную нагрузку на ваш сервер. Он поддерживает такие возможности, как изготовление сжатых резервных копий, инкрементное резервное копирование и потоковое резервное копирование на другой сервер. Это официальный инструмент резервного копирования для MySQL.

Percona XtraBackup

Инструмент Percona XtraBackup во многом похож на MySQL Enterprise Backup, но у него открытый исходный код и он бесплатный. Он поддерживает такие функции, как потоковые, инкрементные, сжатые и многопоточные (параллельные) операции резервного копирования. В дополнение к основному инструменту резервного копирования здесь есть также множество специальных функций для снижения влияния резервного копирования на сильно загруженные системы.

Percona XtraBackup работает, отслеживая файлы журналов InnoDB в фоновом потоке, а затем копируя файлы данных InnoDB. Это слегка запутанный процесс со специальными проверками, обеспечивающими согласованность копирования данных. Когда все файлы данных скопированы, поток копирования журнала также завершается. В результате получается копия всех данных, но в разные моменты времени. Теперь журналы можно применить к файлам данных с использованием процедур аварийного восстановления InnoDB, чтобы привести все файлы данных в согласованное состояние. Это называется процессом подготовки. После подготовки резервная копия полностью согласована и содержит все закоммиченные транзакции по состоянию на момент завершения копирования файлов. Все эти процессы происходят вне MySQL, поэтому серверу не нужно каким-либо образом подключаться к MySQL или обращаться к ней.

mydumper

Несколько нынешних и бывших инженеров, работавших над MySQL, создали **mydumper** в качестве замены **mysqldump**, основываясь на своем многолетнем опыте. Это многопоточный (параллельный) набор инструментов для резервного копирования и восстановления для MySQL с множеством приятных функций. Многие, по-видимому, посчитают скорость многопоточного резервного копирования и восстановления самым привлекательным достоинством этого инструмента.

mysqldump

Большинство людей используют программы, которые поставляются с MySQL, поэтому, несмотря на свои недостатки, наиболее распространенным вариантом для создания логических резервных копий данных и схем является `mysqldump`. Обратитесь к официальному руководству для получения подробной информации о том, как применять этот инструмент.

Резервное копирование данных

Как и в большинстве случаев, существуют хорошие и плохие способы резервного копирования, причем самые очевидные из них не обязательно лучшие. Хитрость заключается в том, чтобы по максимуму задействовать производительность сети, диска и процессора и тем самым сделать резервные копии как можно быстрее. Вам придется поэкспериментировать, для того чтобы найти золотую середину.

Логические SQL-дампы

Логические SQL-дампы — это то, с чем знакомо большинство людей, потому что утилита `mysqldump` создает их по умолчанию. Например, при выгрузке дампа небольшой таблицы с параметрами по умолчанию получится такой результат (мы кое-что опустили):

```
$ mysqldump test t1
-- [Version and host comments]
/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
-- [More version-specific comments to save options for restore]
--
-- Table structure for table `t1`
--

DROP TABLE IF EXISTS `t1`;
/*!40101 SET @saved_cs_client = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `t1` (
  `a` int NOT NULL,
  PRIMARY KEY (`a`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
/*!40101 SET character_set_client = @saved_cs_client */;
--
-- Dumping data for table `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 */;
-- [More option restoration]
```

Файл дампа содержит как описание структуры таблицы, так и данные, причем и то и другое представлено в виде корректных 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. Далее вы можете увидеть описание структуры таблицы, а затем ее данные. Наконец, сценарий сбрасывает параметры, которые были изменены в начале дампа.

Выходные данные дампа можно использовать для операции восстановления. Это удобно, но подразумеваемые по умолчанию параметры `mysqldump` не подходят для создания очень больших резервных копий.

Утилита `mysqldump` не единственный инструмент, который может создавать логические резервные копии. Вы также можете создать их, например, с помощью утилиты `mydumper` или `phpMyAdmin`. На самом деле мы хотели бы отметить здесь не столько проблемы, свойственные тому или иному конкретному инструменту, сколько недостатки создания монолитных логических резервных копий как таковых.

Вот основные проблемные зоны.

- *Схема и данные хранятся вместе.* Хотя это и удобно, когда нужно восстановить все из одного файла, но усложняет задачу, если вам требуется восстановить только одну таблицу или только данные. Эту трудность можно устранить, выгрузив дампы дважды — один раз для данных, один раз 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 огромный файл.* Большинство текстовых редакторов не могут редактировать большие файлы или файлы с очень длинными строками. Хотя в некоторых случаях для извлечения нужных данных можно воспользоваться потоковыми редакторами, такими как `sed` или `grep`, но предпочтительнее, чтобы файлы были небольшими.

- *Создание логической резервной копии весьма затратно.* Существуют более эффективные способы извлечения данных из MySQL, помимо передачи их по протоколу взаимодействия между клиентом и сервером в виде результирующего набора.

Как видите, логическое резервное копирование может быть трудно заставить работать в вашей среде. Если вам необходимо использовать логическое резервное копирование, настоятельно рекомендуем обратить внимание на `mysdumper`, чтобы избежать однопоточной работы и уделить время измерениям воздействия на вашу базу данных при выполнении резервного копирования.

Снимки файловой системы

Снимки файловой системы — прекрасный способ выполнить оперативное резервное копирование. Файловые системы с поддержкой моментальных снимков способны в любой момент времени мгновенно создавать согласованный образ своего 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, такие как устройства хранения NetApp. Некоторые варианты удаленно подключенных дисков, предоставляемые облачными провайдерами, также предлагают моментальные снимки диска.

Не путайте снимок с резервной копией. Создание моментального снимка — это просто способ уменьшить время, в течение которого должны удерживаться блокировки; после того как они сняты, необходимо скопировать файлы в резервную копию. На самом деле вы можете делать моментальные снимки в InnoDB, даже не получая блокировки. Мы покажем вам два способа использования LVM для резервного копирования базы, состоящей только из таблиц InnoDB: с минимальной блокировкой и вообще без блокировки.

МИФ О РЕЗЕРВНОМ КОПИРОВАНИИ № 2. «МОЙ СНИМОК — ЭТО МОЯ РЕЗЕРВНАЯ КОПИЯ»

Моментальный снимок, будь то снимок LVM, моментальные снимки ZFS или SAN, не является настоящей резервной копией, так как не содержит полной копии ваших данных. Поскольку моментальные снимки поддерживают копирование при записи, они содержат только различия между фактической копией данных и данными на то время, когда был создан моментальный снимок. Если немодифицированный блок оказывается поврежденным в фактической копии данных, то хорошая копия этого блока, которую можно использовать для восстановления, отсутствует и каждый моментальный снимок видит тот же поврежденный блок, что и существующий том. Применяйте моментальные снимки, чтобы зафиксировать свои данные во время резервного копирования, но не полагайтесь на сам снимок как на резервную копию.

Моментальный снимок может быть отличным способом создания резервной копии для определенных целей. Один из примеров — резервное копирование в случае возникновения проблем во время обновления. Вы можете сделать моментальный снимок, обновить систему и, если возникнут проблемы, просто вернуться к этому снимку. То же самое можно сделать для любой неопределенной и рискованной операции, например для изменения огромной таблицы (это займет неизвестное количество времени).

Как работают снимки LVM

В LVM для создания снимка применяется технология копирования при записи, а это означает, что логическая копия всего тома снимается мгновенно. Чем-то это немного похоже на технологию MVCC в базах данных, за исключением того, что сохраняется только одна старая версия данных.

Заметьте, мы не сказали «физическая копия». Логическая копия на первый взгляд содержит те же данные, что и том, с которого был сделан моментальный снимок, но изначально она не содержит никаких данных. Вместо того чтобы копировать данные в моментальный снимок, LVM просто запоминает время создания моментального снимка, а затем считывает данные из их исходного местоположения, когда вы запрашиваете данные из моментального снимка. Таким образом, операция первоначального копирования производится мгновенно независимо от размера тома, для которого создается моментальный снимок.

Когда что-то изменяет данные в исходном томе, LVM копирует прежнее содержимое блоков в область, зарезервированную для моментального снимка, прежде чем записывать в них какие-либо модификации. LVM не хранит несколько старых версий данных, поэтому последующие записи в блоки, измененные в исходном томе, не требуют дополнительной работы с моментальным снимком. Другими словами, только первая запись в каждый блок вызывает его копирование в зарезервированную область.

Теперь, когда вы запрашиваете эти блоки в моментальном снимке, LVM считывает данные из скопированных блоков, а не из исходного тома. Это позволяет вам продолжать видеть одни и те же данные в моментальном снимке, не блокируя ничего в исходном томе. Эта схема показана на рис. 10.1.

Снимок создает в каталоге `/dev` новое логическое устройство, и вы можете монтировать его так же, как и любое другое.

Теоретически с помощью этого метода вы можете сделать моментальный снимок огромного тома, заняв для этого очень мало физического пространства. Однако вам необходимо зарезервировать достаточно места для хранения всех блоков, которые будут обновлены в исходном томе, пока моментальный снимок

остается открытым. Если вы не зарезервируете достаточно места для копирования при записи, для моментального снимка закончится место и устройство станет недоступным. Эффект подобен отключению внешнего диска: любая операция чтения с устройства, в том числе резервное копирование, завершается с ошибкой ввода/вывода.

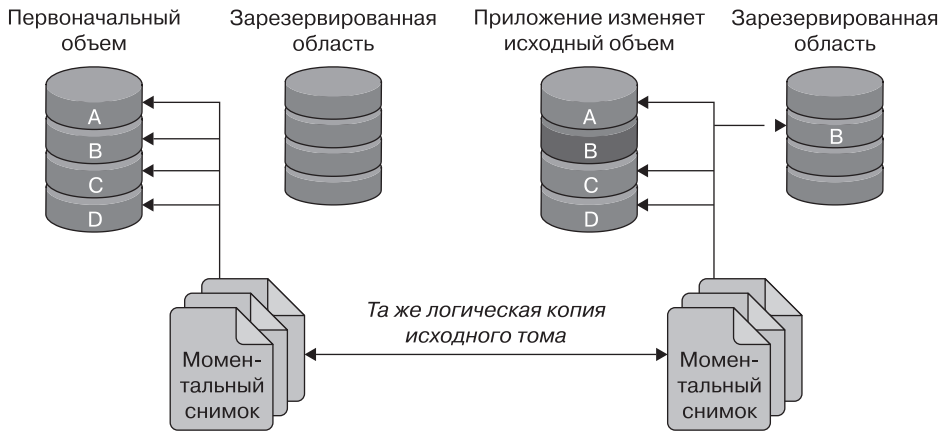


Рис. 10.1. Как технология копирования при записи уменьшает размер моментального снимка

Необходимые условия и конфигурация

Создание снимка — почти тривиальная задача, но вам нужно убедиться, что система сконфигурирована таким образом, чтобы можно было получить согласованный дубликат всех файлов, для которых планируете создать резервную копию, в один и тот же момент времени. Вначале убедитесь, что система удовлетворяет следующим условиям.

- Все файлы InnoDB (файлы табличного пространства InnoDB и журналы транзакций InnoDB) должны находиться в одном логическом томе (разделе). Вам необходима абсолютная согласованность на определенный момент времени, а LVM не умеет одномоментно делать согласованные снимки нескольких томов. (Это ограничение LVM, в некоторых других системах такой проблемы нет.)
- Если вам необходимо включить в резервную копию также определения таблиц, то каталог данных MySQL должен находиться в том же логическом томе. Если вы используете другой метод для резервного копирования определений таблиц, например хранение схемы в системе управления версиями, вам, возможно, не нужно беспокоиться об этом.

- Для создания моментального снимка в группе томов должно быть достаточно свободного места. Сколько его нужно, будет зависеть от рабочей нагрузки. Поэтому при конфигурировании системы оставьте нераспределенное пространство, чтобы позже у вас было место для моментальных снимков.

В LVM существует понятие группы томов, в которую входит один или несколько логических томов. Вы можете посмотреть, какие группы томов имеются в вашей системе, следующим образом:

```
$ vgs
VG #PV #LV #SN Attr VSize VFree
vg 1 4 0 wz--n- 534.18G 249.18G
```

В этих выходных данных показана группа томов, состоящая из четырех логических томов, которые находятся в одном физическом томе, в котором свободно примерно 250 Гбайт. При необходимости можно получить более подробную информацию с помощью утилиты `vgdisplay`. Теперь посмотрим на сами логические тома в системе:

```
$ 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
```

Вывод показывает, что размер тома `mysql` составляет 225 Гбайт. Данное устройство называется `/dev/vg/mysql`. Это просто имя, хотя оно выглядит как путь к файловой системе. Еще сильнее запутывает ситуацию то, что существует символическая ссылка с этого имени на узел реального устройства `/dev/mapper/vg-mysql`, которую вы можете увидеть с помощью команд `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
```

Вооружившись этой информацией, можете приступить к созданию снимка файловой системы.

Создание, монтирование и удаление моментального снимка LVM

Можно создать снимок с помощью одной команды. Вам нужно лишь решить, куда его поместить и сколько места выделить для копирования при записи. Не бойтесь задействовать больше места, чем, по вашему мнению, может реально

потребуется. LVM не использует указанное вами пространство немедленно, а зарезервирует его для применения в будущем, поэтому резервирование очень большого пространства не принесет никакого вреда, если только вам одновременно не нужно оставлять место для других снимков.

Создадим снимок просто для практики. Предоставим ему 16 Гбайт пространства для копирования при записи и назовем его `backup_mysql`:

```
$ lvcreate --size 16G --snapshot --name backup_mysql /dev/vg/mysql
Logical volume "backup_mysql" created
```



Мы намеренно назвали том `backup_mysql` вместо `mysql_backup`, чтобы не возникало неоднозначности при автоматическом завершении команды после нажатия клавиши табуляции. Это помогает избежать вероятности случайного удаления группы томов `mysql` при автодополнении имени файла нажатием кнопки табуляции.

Теперь посмотрим на состояние только что созданного тома:

```
$ 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 на то, что атрибуты моментального снимка отличаются от атрибутов исходного устройства и что на дисплее отображается небольшая дополнительная информация — его происхождение и то, сколько из выделенных 16 Гбайт в настоящее время используется для копирования при записи. Рекомендуется следить за этим при создании резервной копии, чтобы видеть, не переполняется ли устройство и не собирается ли оно выйти из строя. Вы можете вести мониторинг состояния своего устройства в интерактивном режиме или с помощью системы мониторинга, такой как Nagios:

```
$ watch 'lvs | grep backup'
```

Как видно из представленной ранее выдачи команды `mount`, том `mysql` содержит файловую систему. Это означает, что она же будет и в томе моментального снимка и вы можете монтировать и использовать его так же, как и любую другую файловую систему:

```
$ mkdir /tmp/backup
$ mount /dev/mapper/vg-backup_mysql /tmp/backup
$ ls -l /tmp/backup
```

```
total 188880
-rw-r-----. 1 mysql mysql 56 Jul 30 22:16 auto.cnf
-rw-r-----. 1 mysql mysql 475 Jul 30 22:31 binlog.000001
-rw-r-----. 1 mysql mysql 156 Jul 30 22:31 binlog.000002
-rw-r-----. 1 mysql mysql 32 Jul 30 22:31 binlog.index
-rw-----. 1 mysql mysql 1676 Jul 30 22:16 ca-key.pem
-rw-r--r--. 1 mysql mysql 1120 Jul 30 22:16 ca.pem
-rw-r--r--. 1 mysql mysql 1120 Jul 30 22:16 client-cert.pem
-rw-----. 1 mysql mysql 1676 Jul 30 22:16 client-key.pem
... omitted ...
```

Поскольку мы лишь практикуемся, сейчас размонтируем и удалим снимок с помощью команды `lvremove`:

```
$ umount /tmp/backup
$ rmdir /tmp/backup
$ lvremove --force /dev/vg/backup_mysql
Logical volume "backup_mysql" successfully removed
```

Применение снимков LVM для резервного копирования InnoDB без блокировок

Если вы используете MySQL 8+ только с таблицами InnoDB, задействуя GTID и режим полной ACID-совместимости, создание резервной копии становится невероятно простым. Во время работы MySQL сделайте моментальный снимок, смонтируйте снимок, а затем скопируйте файлы в место хранения резервной копии. Нет необходимости блокировать какие-то файлы, перехватывать какие-либо выходные данные или выполнять какие-либо специальные действия. При восстановлении файлов из одной из этих резервных копий будет выполнено аварийное восстановление InnoDB, а настройки GTID уже будут знать, какие транзакции обработаны.

Планирование резервного копирования с помощью LVM

Самое важное, что нужно спланировать, — выделение места, которого будет достаточно для моментального снимка. Мы рекомендуем использовать следующий подход.

- Помните, что LVM должен скопировать каждый измененный блок в снимок только один раз. Когда MySQL записывает блок в исходный том, LVM копирует блок в моментальный снимок, а затем вносит пометку о скопированном блоке в свою таблицу исключений. При последующих изменениях того же блока он уже не копируется.
- Если в базе используются только таблицы типа InnoDB, то примите во внимание, как InnoDB сохраняет данные. Поскольку любой элемент данных

записывается дважды, по крайней мере половина операций ввода/вывода InnoDB попадает в буфер двойной записи, файлы журналов и другие сравнительно компактные области на диске. Тем самым одни и те же блоки перезаписываются снова и снова, поэтому в первой модификации они будут влиять на моментальный снимок, но после этого перестанут вызывать запись в моментальный снимок.

- Затем оцените, какая часть операций ввода/вывода будет сопряжена с записью в блоки, которые еще не были скопированы в моментальный снимок, по сравнению с повторной модификацией уже измененных данных. Не бойтесь зависеть оценку.
- Используйте утилиты `vmstat` или `iostat` для сбора статистики о количестве блоков, записываемых в секунду вашим сервером.
- Измерьте (или оцените), сколько времени потребуется для копирования вашей резервной копии в другое место, другими словами, как долго снимок LVM должен оставаться открытым.

Предположим, вы подсчитали, что половина операций записи будет выполняться в область, зарезервированную для копирования при записи моментального снимка, и что сервер сохраняет 10 Мбайт в секунду. Если копирование моментального снимка на другой сервер занимает час (3600 с), то потребуется $1 / 2 \times 10 \times 3600$ Мбайт, или 18 Гбайт, места для моментального снимка. На всякий случай добавьте дополнительное пространство.

Иногда бывает легко подсчитать, сколько данных изменится, пока вы держите снимок открытым.

Другие варианты использования и альтернативы

Можно задействовать моментальные снимки не только для резервного копирования. Например, как упоминалось ранее, они могут быть полезным способом создания контрольной точки непосредственно перед выполнением потенциально опасного действия. Некоторые системы позволяют вернуть данные из моментального снимка к исходному состоянию. Это позволяет легко возвратиться к моменту, когда сделан снимок.

Снимки файловой системы не единственный способ получить мгновенную копию данных. Другим вариантом является расщепление RAID-массива: если имеется реализованный программно зеркалированный RAID-массив из трех дисков, вы можете исключить один диск из зеркала и смонтировать его отдельно. Тогда не будет никаких затрат на копирование при записи, и в случае необходимости очень легко сделать такой вид моментального снимка главной

копией. Однако после добавления этого диска обратно в RAID-массив его необходимо будет повторно синхронизировать. К сожалению, бесплатного сыра не бывает.

Percona XtraBackup

Инструмент Percona XtraBackup — одно из самых популярных решений для резервного копирования MySQL, и на то есть веские причины. Он обладает широкими возможностями настройки, включая способы резервного копирования сжатых и зашифрованных файлов.

Как работает XtraBackup

InnoDB — отказоустойчивая подсистема хранения данных. Если в MySQL происходит сбой, она использует режим восстановления после сбоя, основанный на журналах повторного выполнения (**redo logs**), чтобы корректно вернуть ваши данные в рабочее состояние. Инструмент Percona XtraBackup основан на этой схеме. Когда вы создаете резервную копию с помощью Percona XtraBackup, он записывает порядковый номер журнала (**log sequence number, LSN**) и применяет его для выполнения аварийного восстановления резервных копий файлов. Он также включает блокировку в определенных точках, чтобы гарантировать, что данные о репликации согласуются с исходными данными. Для получения более подробного объяснения обратитесь к документации XtraBackup. Вот пример процесса XtraBackup:

```
$ xtrabackup --backup --target-dir=/backups/
```

```
xtrabackup version 8.0.25-17 based on MySQL server 8.0.25 Linux (x86_64)
(revision id: d27028b)
Using server version 8.0.25-15
210821 17:01:40 Executing LOCK TABLES FOR BACKUP...
```

В этот момент мы видим, что XtraBackup установил работающую версию MySQL. Это помогает ему определить, какими возможностями он обладает и как ему следует выполнять резервное копирование файлов. В нашем случае доступна команда **LOCK TABLES FOR BACKUP**, с помощью которой XtraBackup получает блокировку таблиц:

```
210821 17:01:41 [01] Copying ./ibdata1 to /backups/ibdata1
210821 17:01:41 [01] ...done
210821 17:01:41 [01] Copying ./sys/sys_config.ibd to /backups/sys/sys_config.ibd
210821 17:01:41 [01] ...done
210821 17:01:41 [01] Copying ./test/t1.ibd to /backups/test/t1.ibd
210821 17:01:41 [01] ...done
```

```

210821 17:01:41 [01] Copying ./foo/t1.ibd to /backups/foo/t1.ibd
210821 17:01:41 [01] ...done
210821 17:01:41 [01] Copying ./sakila/actor.ibd to /backups/sakila/actor.ibd
210821 17:01:41 [01] ...done

```

Теперь XtraBackup копирует файлы из источника в место назначения:

```

210821 17:01:42 Finished backing up non-InnoDB tables and files
210821 17:01:42 Executing FLUSH NO_WRITE_TO_BINLOG BINARY LOGS
210821 17:01:42 Selecting LSN and binary log position from p_s.log_status
210821 17:01:42 [00] Copying /var/lib/mysql/binlog.40 to /backups/binlog.04
up to position 156
210821 17:01:42 [00] ...done
210821 17:01:42 [00] Writing /backups/binlog.index
210821 17:01:42 [00] ...done
210821 17:01:42 [00] Writing /backups/xtrabackup_binlog_info
210821 17:01:42 [00] ...done

```

После завершения копирования файлов он собирает информацию о репликации:

```

210821 17:01:42 Executing FLUSH NO_WRITE_TO_BINLOG ENGINE LOGS...
xtrabackup: The latest check point (for incremental): '35005805'
xtrabackup: Stopping log copying thread at LSN 35005815.
210821 17:01:42 >> log scanned up to (35005825)
Starting to parse redo log at lsn = 35005460
210821 17:01:43 Executing UNLOCK TABLES
210821 17:01:43 All tables unlocked

```

Теперь XtraBackup определил последнюю контрольную точку для InnoDB. Это поможет ему применить записи, которые сделаны во время резервного копирования. Он освобождает блокировку таблиц, полученную предыдущей командой LOCK TABLES FOR BACKUP, с помощью команды UNLOCK TABLES:

```

210821 17:01:43 [00] Copying ib_buffer_pool to /backups/ib_buffer_pool
210821 17:01:43 [00] ...done
210821 17:01:43 Backup created in directory '/backups/'
MySQL binlog position: filename 'binlog.000004', position '156'
210821 17:01:43 [00] Writing /backups/backup-my.cnf
210821 17:01:43 [00] ...done
210821 17:01:43 [00] Writing /backups/xtrabackup_info
210821 17:01:43 [00] ...done
xtrabackup: Transaction log of lsn (35005795) to (35005835) was copied.
210821 17:01:44 completed OK!

```

Последними шагами являются запись LSN, копирование дампа буферного пула и запись окончательных файлов. Один из них является копией файла `my.cnf`, а файл `xtrabackup_info` содержит метаданные о резервной копии, такие как UUID MySQL, версии сервера и XtraBackup.

Примеры использования

Мы опишем в общих чертах несколько основных рецептов применения XtraBackup, но предварим их несколькими замечаниями.

- Ваша установка MySQL должна быть защищена паролем. Убедитесь, что используете параметры `--user` и `--password`, чтобы указать учетную запись с достаточными правами для создания резервных копий.
- Выходные данные XtraBackup очень многословны. Мы сократили вывод, чтобы выделить наиболее важные части каждого сценария применения.
- Как обычно, рекомендуем ознакомиться с официальным руководством для Percona XtraBackup, прежде чем запускать здесь какие-либо команды, поскольку синтаксис и параметры могут измениться. Несмотря на то что нам неизвестно о каких-либо потерях данных, связанных с этим инструментом, вам следует протестировать его на непроизводственной резервной копии, прежде чем пробовать работать с критически важными данными.

Базовое резервное копирование в каталог. Первый метод, который мы хотим показать, — это то, как можно использовать XtraBackup для создания полной резервной копии ваших данных в другом каталоге. В зависимости от того, что вы будете делать с данными впоследствии, это может быть другой диск, каталог на том же диске или смонтированный общий файловый ресурс на большом сервере резервного копирования. Имейте в виду, что для выполнения такого резервного копирования потребуется соответствующее пространство для копирования файлов.

Вот самый простой вариант использования XtraBackup, в котором указывается режим (резервное копирование) и место для резервного копирования файлов (целевой каталог):

```
$ xtrabackup --backup --target-dir=/backups/
```

После выполнения вывод будет выглядеть примерно так, как описано в пункте «Как работает XtraBackup» ранее. В случае успеха каталог `/backups` будет содержать полную копию ваших данных.

Потоковое резервное копирование. Копирование всех файлов в новый каталог может быть не самым идеальным вариантом использования. Иногда проще хранить несколько резервных копий в одном каталоге. В этом случае может быть полезна опция потокового резервного копирования. Потоковая передача позволяет записать резервную копию в виде одного файла:

```
$ xtrabackup --backup --stream=xbstream > /backups/backup.xbstream
```

В этом примере мы по-прежнему указываем режим резервного копирования и удаляем параметр `target-dir`, поскольку вывод будет осуществляться в `STDOUT`. Затем перенаправим его в файл.

Обратите внимание: вы можете использовать также команду `Bash shell` с параметром `date`, чтобы включить временную метку в имя выходного файла, например:

```
$ xtrabackup --backup --stream=xbstream > /backups/backup-$(date +%F).xbstream
```

Весь процесс резервного копирования будет выполняться, как и раньше, с использованием `STDOUT` в качестве места назначения. Содержимое будет записано в файл `xbstream` в `/backups`.

Резервное копирование со сжатием. Как мы отмечали ранее, потребуется достаточно места для создания полной копии ваших файлов данных или единственного файла `xbstream`. Один из распространенных способов уменьшения требований к пространству — использование функции сжатия в XtraBackup:

```
$ xtrabackup --backup --compress --stream=xbstream > /backups/backup  
compressed.xbstream
```

Вы заметите, что вместо отображения *Streaming* для каждой таблицы теперь выводится *Compressing and streaming*. В ходе тестирования мы загрузили базу данных Sakila Sample Database и наблюдали, как несжатый файл `xbstream` размером 94 Мбайт превратился в сжатый файл размером 6,5 Мбайт.

Резервное копирование с шифрованием. Последний пример, который мы хотим рассмотреть, — это использование шифрования в рамках стратегии резервного копирования. Шифрование потребует больше ресурсов процессора, а резервное копирование займет больше времени, однако это может оказаться приемлемым компромиссом, учитывая то, что при создании резервной копии легко получить много данных в одном файле. Мы снова используем режим резервного копирования и потоковую передачу, но применим шифрование с помощью ключа и параметра `encrypt-key-file`, чтобы указать файл, в котором находится ключ шифрования:

```
$ xtrabackup --backup --encrypt=AES256 --encrypt-key-  
file=/safe/key/location/encrypt.key --stream=xbstream > /backups/backup  
encrypted.xbstream
```

Вывод снова изменился, отобразив *Encrypting and streaming* для каждого файла.

Обратите внимание на то, что вы можете также использовать параметр `--encrypt-key` и указать его в командной строке. Мы не рекомендуем этого делать, поскольку

ку ключ будет доступен в списке процессов или как часть файловой системы `/proc` в Linux.

Другие важные флаги. Один из аспектов, на который вы должны обратить внимание, — это время, необходимое для завершения резервного копирования. Помочь в этом могут параметры `--parallel` и `compress-threads`. Их использование увеличит загрузку процессора, но должно сократить общее время, необходимое для резервного копирования. Шифрование имеет аналогичные параметры распараллеливания.

Если у вас много баз данных и таблиц, воспользуйтесь параметром `--rsync`, чтобы оптимизировать процесс копирования файлов.

Восстановление из резервной копии

Способ восстановления данных зависит от того, как вы создали их резервную копию. Возможно, вам потребуется выполнить все перечисленные шаги или некоторые из них.

1. Остановите сервер MySQL.
2. Запишите куда-нибудь данные о конфигурации сервера и правах доступа к файлам.
3. Скопируйте данные из резервной копии в каталог данных MySQL.
4. Внесите изменения в конфигурационные файлы.
5. Измените права доступа к файлам.
6. Перезапустите сервер в режиме ограниченного доступа и подождите, пока он не перейдет в состояние готовности.
7. Загрузите файлы логической резервной копии.
8. Проверьте и воспроизведите двоичные журналы.
9. Убедитесь, что все восстановлено.
10. Перезапустите сервер в режиме полного доступа.

В следующих разделах по мере необходимости мы продемонстрируем, как выполняется каждый из этих шагов. А также добавим примечания, относящиеся к определенным методам или инструментам резервного копирования, в посвященные им разделы.



Если есть вероятность, что вам еще понадобятся текущие версии ваших файлов, не заменяйте их файлами из резервной копии. Например, если резервная копия содержит двоичные журналы и их необходимо воспроизвести для восстановления на определенный момент времени, не перезаписывайте текущие двоичные журналы устаревшими версиями из резервной копии. При необходимости переименуйте их или переместите в другое место.

Во время восстановления часто бывает важно, чтобы к MySQL мог обращаться лишь процесс восстановления и больше никто. Чтобы гарантировать, что сервер будет недоступен для существующих приложений, запускаем его с флагами `--skip-networking` и `--socket=/tmp/mysql_recover.sock`, пока мы все не проверим и не вернем его в оперативный режим. Это особенно важно для логических резервных копий, которые загружаются по частям.

Восстановление логических резервных копий

Если вы восстанавливаете из логической резервной копии, а не из физических файлов, то для загрузки данных в таблицы вам понадобится сам сервер MySQL. Копирования файлов на уровне операционной системы недостаточно.

Однако, прежде чем загружать этот файл дампа, найдите время, чтобы посмотреть на его размер и подумать, сколько времени он будет обрабатываться и не надо ли что-то сделать перед началом процедуры, например уведомить своих пользователей или деактивировать часть приложения. Полезно на этот период отключить запись в двоичный журнал, если только вам не нужно реплицировать восстанавливаемые данные на реплику: загружать огромный дамп серверу и так довольно сложно, а запись его в двоичный журнал увеличивает накладные расходы (возможно, без всякой необходимости). Кроме того, в некоторых подсистемах хранения загрузка очень больших файлов требует особых предосторожностей. Например, не рекомендуется загружать 100 Гбайт данных в таблицу InnoDB одной транзакцией, поскольку в результате образуется огромный сегмент отката. Для загрузки рекомендуется разбить файл на фрагменты и коммитить транзакцию после каждого из них.

Существует две разновидности восстановления, соответствующие двум видам логических резервных копий, которые вы можете сделать.

Если у вас есть SQL-дамп, файл будет содержать исполняемые команды SQL. Вам останется только запустить его. Предположим, что вы создали резервную копию демонстрационной базы данных Sakila Sample Database и схемы в одном

файле. Далее приведена команда, которую обычно можно использовать для ее восстановления:

```
$ mysql < sakila-backup.sql
```

Можете также загрузить файл из клиента командной строки `mysql` с помощью команды `SOURCE`. Хотя это, в общем-то, лишь другой способ сделать то же самое, выполнить некоторые вещи при этом оказывается проще. Например, если вы обладаете административными правами в MySQL, то можете отключить запись в двоичный журнал команд, которые будете выполнять из текущего соединения, а затем загрузить файл, не перезапуская сервер MySQL:

```
SET SQL_LOG_BIN = 0;  
SOURCE sakila-backup.sql;  
SET SQL_LOG_BIN = 1;
```

Но при использовании команды `SOURCE` имейте в виду, что ошибка не прерывает выполнения пакета команд, как происходит по умолчанию в случае чтения из стандартного ввода `mysql`.

Если резервная копия была сжата, не нужно перед загрузкой отдельно распаковать ее. Разархивирование и загрузку можно объединить в одной операции. Это будет намного быстрее:

```
$ gunzip -c sakila-backup.sql.gz | mysql
```

Что делать, если вы хотите восстановить данные только для одной таблицы, например `actor`? Если данные не содержат символов перехода на новую строку и схема таблицы уже имеется, 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 и у вас есть вся база данных в одном файле, придется отредактировать этот файл. Именно поэтому многие предпочитают выгружать каждую таблицу в отдельный файл. Большинство редакторов не могут работать с огромными файлами, особенно если они сжаты. Кроме того, вам не требуется редактировать файл, достаточно лишь извлечь из него соответствующие строки, поэтому, вероятно, придется выполнить некоторую работу с помощью утилит командной строки. Довольно просто воспользоваться инструментом `grep` для выборки только команд `INSERT`, относящихся к данной таблице, как мы делали в предыдущих командах, но

извлечь команду `CREATE TABLE` сложнее. Далее приведен скрипт `sed`, который извлекает именно то, что необходимо:

```
$ sed -e '/./{H;$!d;}' -e 'x;/CREATE TABLE `actor`/!d;q' sakila-backup.sql
```

Мы признаем, что это выглядит довольно загадочно. Если для восстановления данных вам приходится идти на подобные трюки, значит, процедура резервного копирования плохо спроектирована. Когда все распланировано заранее, можно избежать ситуаций, когда вы в панике пытаетесь разобраться, как работает `sed`. Просто создайте резервную копию каждой таблицы в отдельном файле или, что еще лучше, создайте резервную копию данных и схемы отдельно.

Восстановление физических файлов из моментального снимка

Процедура восстановления данных из физических файлов, как правило, довольно прямолинейна, иначе говоря, вариантов здесь не так уж много. Это может быть и хорошо и плохо в зависимости от ваших требований к восстановлению. Обычно все сводится к простому копированию файлов в нужное место.

Если восстанавливается традиционно сконфигурированная база с подсистемой InnoDB, когда все таблицы хранятся в одном табличном пространстве, вам нужно остановить MySQL, скопировать или переместить файлы на место, а затем перезапустить сервер. Необходимо также убедиться, что файлы журнала транзакций InnoDB соответствуют файлам, содержащим табличное пространство. Если эти файлы не соответствуют друг другу, например, если вы заменили файлы табличного пространства, но забыли файлы журналов транзакций, то InnoDB откажется запускаться. Это одна из причин, по которой крайне важно создавать резервные копии журнала транзакций вместе с файлами данных.

Если вы используете возможность размещать по одной таблице в каждом файле (режим `innodb_file_per_table`), то InnoDB сохранит данные и индексы для каждой таблицы в файле с расширением `.ibd`. Резервное копирование и восстановление отдельных таблиц в этом случае может выполняться простым копированием этих файлов, делать это можно на работающем сервере, но процесс не очень прост. Отдельные файлы не являются независимыми от InnoDB в целом. Каждый файл `.ibd` содержит внутреннюю информацию, которая сообщает InnoDB, как файл связан с основным (общим) табличным пространством. Восстанавливая такой файл, вам необходимо попросить InnoDB импортировать его.

На эту процедуру наложено множество ограничений, о которых вы можете прочитать в разделе руководства по MySQL, посвященном использованию

табличных пространств с отдельным хранением таблиц. Самое серьезное ограничение заключается в том, что вы можете восстановить таблицу только на тот сервер, с которого сделали ее резервную копию. Резервное копирование и восстановление таблиц в этой конфигурации возможно, но это несколько сложнее, чем может показаться.

Наличие таких сложностей означает, что восстановление физических файлов может быть очень трудоемким процессом, в ходе которого можно легко ошибиться. Эмпирическое правило заключается в том, что чем сложнее и труднее процедура восстановления, тем важнее застраховаться от неприятностей, создавая также логические резервные копии. Всегда полезно иметь логическую резервную копию на случай, если что-то пойдет не так и вы не сможете заставить MySQL использовать физические резервные копии.

Восстановление с помощью Percona XtraBackup

В пункте «Как работает XtraBackup» мы упомянули, что для создания безопасных резервных копий используется процесс аварийного восстановления InnoDB. Это означает, что для применения файлов, резервные копии которых созданы с помощью XtraBackup, необходимо выполнить дополнительные действия.

Если вы использовали потоковое резервное копирование, то необходимо сначала распаковать файл `xbstream`. Для извлечения `xbstream` можно применять команду `xbstream`:

```
$ xbstream -x < backup.xbstream
```

Она извлечет все файлы в ваше нынешнее местоположение. Либо используйте параметр `-C` для предварительного перехода в определенный каталог. Если вы применяли сжатие или шифрование, можете задействовать аналогичные параметры для обратного процесса. Для сжатого файла возьмите параметр `-decompress`, а для шифрования — параметр `--decrypt`, указав при этом местоположение файла ключа `--encrypt-keyfile`:

```
$ xbstream -x --decompress < backup-compressed.xbstream
```

```
$ xbstream -x --decrypt --encrypt-key-file=/safe/key/location/encrypt.key  
  < backup-encrypted.xbstream
```

После завершения работы следующим шагом будет подготовка файлов. Подготовка — это процесс, который фактически выполняет действия по восстановлению после сбоя и гарантирует, что будут восстановлены все данные:

```
$ xtrabackup --prepare --target-dir=/restore
```



Если вы не используете потоковый режим, можете выполнить этап подготовки после создания резервной копии. Это приведет к созданию подготовленной резервной копии и уменьшит объем работы, которую необходимо выполнить, когда придет время восстановления.

После успешного завершения вы готовы использовать эти файлы для запуска MySQL:

```
$ xtrabackup --move-back --target-dir=/restore
```



Вы можете использовать флаги `--copy-back` или `--move-back` с XtraBackup, чтобы правильно скопировать или переместить файлы на место.

XtraBackup автоматически обнаружит переменную `data-dir` из вашей установки MySQL и переместит файлы в нужное место.

Запуск MySQL после восстановления физических файлов

Перед тем как запускать восстановленный сервер MySQL, нужно сделать несколько шагов.

Первое и самое важное, о чем легко забыть, — проверить конфигурацию своего сервера и убедиться, что для восстановленных файлов заданы правильный владелец и права доступа. Это нужно сделать прежде, чем пытаться запустить сервер MySQL. Если эти атрибуты хоть чем-то отличаются от необходимых, MySQL может не запуститься. Атрибуты варьируются от системы к системе, поэтому проверьте свои заметки, чтобы точно знать, как они должны быть установлены. Обычно для файлов и каталогов указываются владелец и группа `mysql`, причем владельцу и группе должны быть разрешены чтение и запись, а всем остальным запрещен любой доступ.

Мы также рекомендуем следить за журналом ошибок MySQL во время запуска сервера. В UNIX-подобных системах вы можете просмотреть файл, выполнив команду:

```
$ tail -f /var/log/mysql/mysql.err
```

Точное местоположение журнала ошибок может различаться в зависимости от системы. Начав мониторинг этого файла, можете запустить сервер MySQL

и наблюдать за тем, что будет появляться в журнале ошибок. Если после запуска MySQL ошибок не возникнет, значит, сервер восстановился нормально и может работать.

Наблюдение за журналом ошибок еще более важно в новых версиях MySQL. Даже если кажется, что сервер запускается без проблем, вы должны выполнить команду `SHOW TABLE STATUS` в каждой базе данных, а затем снова проверить журнал ошибок.

Резюме

Все знают, что им нужны резервные копии, но не все понимают, что это должны быть восстанавливаемые резервные копии. Существует множество способов создания резервных копий, которые противоречат вашим требованиям к восстановлению. Чтобы избежать этой проблемы, предлагаем определить и задокументировать целевую точку восстановления (RPO) и целевое время восстановления (RTO) и использовать эти требования как ориентиры при выборе системы резервного копирования.

Важно регулярно тестировать восстановление, чтобы убедиться в том, что оно работает. Очень просто настроить утилиту `mysqldump` и запускать ее каждую ночь, не понимая, что со временем данные выросли так, что для повторного импорта могут потребоваться несколько дней или даже недель. Хуже всего обнаружить, сколько времени займет восстановление, тогда, когда оно действительно необходимо. Резервное копирование, которое выполняется за несколько часов, для восстановления может потребовать в буквальном смысле нескольких недель в зависимости от используемого оборудования, схемы, индексов и данных.

Не попадайтесь в ловушку, думая, что реплика — это резервная копия. Это менее разрушающий источник для резервного копирования, но не резервная копия. То же самое относится к томам RAID, SAN и моментальным снимкам файловой системы. Убедитесь, что ваши резервные копии могут пройти тест `DROP TABLE` (или тест «меня взломали»), а также тест на утрату центра обработки данных. И если вы берете резервные копии из реплики, убедитесь, что реплики непротиворечивы, перестроив их из вашего источника и с этого момента применяя `super_read_only`.

Безусловно, два наших любимых способа резервного копирования — использование `Percona XtraBackup` для создания физических резервных копий и `mydumper` — для логических резервных копий. Оба метода позволяют получать

неразрушающие двоичные (физические) резервные копии данных, которые затем можно верифицировать, запустив экземпляр `mysqld` и проверив таблицы. Иногда вы даже можете убить одним выстрелом двух зайцев — тестировать восстановление каждый день, восстанавливая резервную копию на сервере разработки или промежуточном сервере. Можете также сделать из этого экземпляра дамп данных, чтобы получить логическую резервную копию. Мы считаем необходимым наряду с этим создавать резервные копии двоичных журналов и хранить достаточное количество поколений резервных копий и двоичных журналов, благодаря которым можно выполнить восстановление или настроить новую реплику, даже если самая последняя резервная копия непригодна для использования.

Масштабирование MySQL

Использование MySQL в личном проекте или даже в молодой компании значительно отличается от ее применения в бизнесе с устоявшимся рынком и быстрым ростом. В условиях быстро расширяющегося бизнеса трафик может расти на несколько порядков из года в год, среда становится все более сложной, а сопутствующие потребности в данных быстро увеличиваются. Масштабирование MySQL сильно отличается от масштабирования других типов серверов в основном из-за характера данных с сохранением состояния. Сравните это с веб-сервером, где широко распространенная модель добавления дополнительных ресурсов за балансировщиком нагрузки — это, как правило, все, что нужно сделать.

В этой главе мы объясним, что означает масштабирование, и проведем вас по разным направлениям, где оно может понадобиться. Мы расскажем, почему масштабирование операций чтения имеет большое значение, и покажем, как выполнить его безопасно, используя такие стратегии, как постановка в очередь, чтобы сделать масштабирование операций записи более предсказуемым. Наконец, поговорим о сегментировании наборов данных для масштабирования операций записи с помощью таких инструментов, как ProxySQL и Vitess. К концу этой главы вы должны будете в состоянии определить, какой шаблон использовать в своей системе и как масштабировать операции чтения и записи.

Что такое масштабируемость

Масштабируемость — это способность системы поддерживать растущий трафик. Критерии того, хорошо или плохо масштабируется система, — это стоимость и простота. Если повышение масштабируемости вашей системы обходится чрезмерно дорого или это очень сложно, вы, скорее всего, потратите значительно больше усилий на исправление ситуации, когда столкнетесь с ограничениями.

Мощность — это родственная категория. Мощность системы — это объем работы, которую она может выполнить за определенный промежуток времени¹. Однако мощность должна отвечать требованиям системы. Максимальная пропускная способность системы не то же самое, что ее мощность. Большинство эталонных тестов измеряют максимальную пропускную способность системы, но в реальности ее так не нагружают. Если вы это сделаете, производительность ухудшится, а время отклика станет неприемлемо большим и непостоянным. Мы определяем фактическую емкость системы как пропускную способность, которой она может достичь, обеспечивая при этом приемлемую производительность.

Мощность и масштабируемость не связаны с производительностью. Проведем аналогию с автомобилями на шоссе.

- Система представляет собой шоссе со всеми полосами движения и автомобилями на нем.
- Производительность — это скорость движения автомобилей.
- Пропускная способность — это количество полос движения, умноженное на максимальную безопасную скорость.
- Масштабируемость — это степень, до которой вы можете добавить больше автомобилей и полос движения без замедления движения.

В этой аналогии масштабируемость зависит от таких факторов, как удобство развязок, количество поломок машин и дорожно-транспортных происшествий, от того, ездят ли автомобили с разной скоростью или часто меняют полосу движения, но в общем случае масштабируемость не зависит от мощности двигателей автомобилей. Это не значит, что производительность не имеет значения, поскольку это не так. Мы просто хотим сказать, что системы могут быть масштабируемыми, даже если они не отличаются высокой производительностью.

В целом масштабируемость — это возможность увеличить производительность за счет добавления ресурсов.

Даже если ваша архитектура MySQL масштабируема, приложение может таковым не быть. Если по какой-либо причине увеличить мощность сложно, то приложение в целом не масштабируется. Ранее мы определяли мощность с точки зрения пропускной способности, но стоит посмотреть на нее с высоты птичьего полета. С этой точки зрения мощность просто означает способность обрабатывать нагрузку, и ее также полезно рассматривать с разных точек зрения.

¹ В физических науках работа в единицу времени называется мощностью, но в вычислительной технике «мощность» настолько перегруженный термин, что он двусмыслен и мы его избегаем. А точным определением мощности (capacity) является максимальная выходная мощность системы.

- *Объем данных.* Огромный объем данных, которые может накапливать ваше приложение, — одна из наиболее распространенных проблем масштабирования. Это особенно актуально для многих современных веб-приложений, которые никогда не удаляют данные. Сайты социальных сетей, например, обычно не удаляют старые сообщения или комментарии.
- *Количество пользователей.* Даже если каждый пользователь порождает небольшой объем данных, при увеличении их числа данные суммируются. Помимо этого, общий объем данных может расти несоизмеримо быстрее, чем количество пользователей. Кроме того, обычно чем больше пользователей, тем больше количество транзакций, а зависимость между количеством транзакций и количеством пользователей тоже может быть нелинейной. Наконец, с увеличением числа пользователей и объема данных сложность запросов чаще всего возрастает, особенно если запросы зависят от количества связей между пользователями. (Количество связей ограничено сверху величиной $(N \times (N - 1)) / 2$, где N — количество пользователей.)
- *Активность пользователей.* Не все пользователи одинаково активны, и их активность распределена неравномерно. Если ваши пользователи вдруг станут более активными — например, из-за появления новых возможностей, которые им понравятся, то нагрузка на приложение может значительно увеличиться. Активность пользователей зависит не только от количества просмотров страниц. То же количество просмотров страниц может потребовать больше работы, если часть сайта, для создания которой требуется много труда, становится более популярной. Кроме того, некоторые пользователи гораздо более активны, чем другие: у них может быть намного больше друзей, сообщений или фотографий, чем у среднего пользователя.
- *Размер взаимосвязанных наборов данных.* Если между пользователями существуют связи, то, вероятно, приложению может потребоваться выполнять запросы и вычисления для целых групп связанных пользователей. Это сложнее, чем работать с индивидуальными пользователями и их данными. Сайты социальных сетей часто сталкиваются с непростыми проблемами из-за особо популярных групп или пользователей, у которых много друзей.

Проблемы масштабирования могут проявляться во многих формах. В следующем разделе поговорим о том, как определить узкое место и что с ним можно сделать.

Рабочие нагрузки, связанные с чтением и записью

Одна из первых вещей, которую вы должны изучить, задумываясь о масштабировании архитектуры вашей базы данных, — масштабируете ли вы рабочую нагрузку, связанную с чтением или записью. Рабочая нагрузка, связанная с чтением, — это нагрузка, при которой объем трафика чтения (SELECT) превышает

возможности вашего сервера. Рабочая нагрузка, связанная с записью, превышает возможности сервера по обслуживанию DML (INSERT, UPDATE, DELETE). Понимание того, с чем вы сталкиваетесь, включает в себя знание вашей рабочей нагрузки.

Знание рабочей нагрузки

Рабочая нагрузка базы данных включает в себя множество вещей. Во-первых, это производительность, или, как говорилось ранее, мера работы за определенное время. Для баз данных это обычно сводится к количеству запросов в секунду. Одним из определений рабочей нагрузки может быть количество запросов в секунду, которое способна выполнить система. Однако не стройте иллюзий: 1000 запросов в секунду при загрузке процессора на 20 % не всегда означает, что можно добавить еще 4000 запросов в секунду. Не все запросы одинаковы.

Запросы бывают разных форм: чтение, запись, поиск по первичному ключу, подзапросы, соединения, массовые вставки и т. д. Каждый из них имеет стоимость. Она измеряется процессорным временем или задержкой. Когда запрос дольше ожидает возврата информации на диске, это время добавляется к стоимости выполнения запроса¹. Важно понимать, каковы возможности ваших ресурсов. Сколько у вас процессоров, каковы ограничения по скорости чтения и записи, пропускной способности диска, а также какова пропускная способность сети? Каждый из них по-своему влияет на задержку, напрямую связанную с рабочей нагрузкой. *Рабочая нагрузка* — это сочетание всех типов запросов и их задержек. Справедливее было бы сказать, что если мы обрабатываем 1000 запросов в секунду при 20%-ной загрузке процессора, то можем добавить еще 4000 запросов в секунду, если их задержки будут одинаковыми². Если же добавим еще 4000 запросов и столкнемся с узким местом дискового ввода/вывода, задержка всех операций чтения возрастет.

Если единственными показателями, к которым у вас есть доступ, являются основные системные показатели, такие как процессор, память и диск, может быть почти невозможно понять, какие из них вы используете. Вам нужно определить, каково соотношение производительности чтения и производительности записи. Мы представили пример этого в подразделе «Анализ производительности чтения и записи» в главе 3. Используя его, вы можете определить задержку операций чтения и записи. Если же проследите динамику этих показателей с течением времени, то сможете увидеть, увеличиваются ли задержки чтения или записи и, следовательно, где могут появиться ограничения.

¹ Для простоты в этом объяснении мы решили игнорировать сложности многопроцессорности и переключения контекста.

² Это все еще не совсем точно, потому что по мере приближения загрузки процессора к 100 % увеличивается задержка, и вы не сможете добавить еще 4000 запросов.

Рабочие нагрузки, связанные с чтением

Предположим, приступив к разработке своего продукта, вы пошли по кратчайшему пути использования одного узла источника для всего трафика базы данных. Добавление узлов приложения может масштабировать количество клиентов, обслуживающих запросы, но в конечном итоге будет ограничено способностью хоста базы данных с одним источником отвечать на запросы на чтение. Основным показателем здесь является загрузка процессора. Высокая загрузка означает, что сервер тратит все свое время на обработку запросов. Чем выше загрузка процессора, тем больше задержек вы увидите в запросах. Однако это не единственный показатель. Вы также можете увидеть высокие показатели операций ввода/вывода в секунду или пропускной способности при чтении с диска, указывающие на то, что вы очень часто обращаетесь к диску или считываете с диска большое количество строк.

Вначале можно улучшить этот показатель добавлением индексов, оптимизацией запросов и кэшированием данных, которые возможно кэшировать. Исчерпав эти возможности улучшения, вы останетесь с рабочей нагрузкой, связанной с чтением, и именно тогда наступит время масштабирования трафика чтения с использованием реплик. Далее в этой главе мы обсудим, как масштабировать операции чтения с помощью пулов реплик чтения, как проверить работоспособность этих пулов и каких подводных камней следует избегать при использовании этой архитектуры.

Рабочие нагрузки, связанные с записью

Вы можете столкнуться и с нагрузкой, связанной с записью. Вот несколько примеров нагрузки на базу данных, связанной с записью.

- Возможно, количество подписчиков растет в геометрической прогрессии.
- Сейчас пик сезона электронной коммерции, и продажи растут вместе с количеством заказов, которые необходимо отслеживать.
- Сейчас сезон выборов, и у вас много информации о предвыборной кампании.

Все это сценарии использования, приводящие к экспоненциальному увеличению числа операций записи в базу данных, которые теперь необходимо масштабировать. Опять же база данных с одним источником, даже если вы можете какое-то время масштабировать ее по вертикали, далеко не уйдет. Когда узким местом является объем записи, вы должны начать думать о способах разделения данных, чтобы можно было параллельно принимать записи на отдельных подмножествах. Позже в этой главе мы поговорим о том, как использовать сегменты для масштабирования записи.

На этом этапе логично спросить: «А что, если я наблюдаю оба типа роста?» Важно внимательно изучить свою схему и определить, есть ли подмножество таблиц, для которых операции чтения увеличиваются быстрее по сравнению с другим подмножеством таблиц, для которых растет потребность в записи. Пытаясь масштабировать кластер базы данных для обоих типов роста одновременно, вы можете получить много неприятностей. Рекомендуем разделять таблицы на разные функциональные кластеры для независимого масштабирования операций чтения и записи. Это необходимое условие для более эффективного масштабирования трафика чтения с помощью пулов чтения.

Теперь, когда вы определили, есть ли у вас нагрузка, связанная с чтением или записью, обсудим, как можно эффективно управлять функциональным разделением данных.

Функциональное сегментирование

Разделение данных на основе их функций в бизнесе — это сложная контекстно зависимая задача, требующая глубокого понимания того, что представляют собой данные. Она идет рука об руку с популярными парадигмами архитектуры программного обеспечения, такими как сервис-ориентированная архитектура (SOA) и микросервисы. Не все функциональные подходы одинаковы, и если в гиперболизированном примере вы поместите каждую таблицу в свою собственную функциональную базу данных, то определенно можете ухудшить ситуацию из-за чрезмерно большой фрагментации.

Как вы подходите к разделению своей большой монолитной/смешанной базы данных проблем на разумный набор небольших кластеров, которые помогут бизнесу масштабироваться? Вот несколько рекомендаций, которых стоит придерживаться.

- Не разделяйте базу данных на основе структуры инженерной команды — в какой-то момент она обязательно изменится.
- Разделяйте таблицы в зависимости от бизнес-функций. Таблицы, обеспечивающие регистрацию учетных записей, могут быть отделены от таблиц, содержащих настройки существующих клиентов, а таблицы, обеспечивающие новую функцию, должны запускаться в их собственной базе данных.
- Не уклоняйтесь от решения проблем, когда в данных смешиваются отдельные бизнес-задачи и вам нужно выступать не только за разделение данных, но и за рефакторинг приложений и внедрение доступа к API через эти границы. Распространенным примером, который мы видели, является смешивание идентификации клиента с выставлением счетов клиентам.

Вполне нормально, что сначала будут таблицы, которые явно имеют собственную бизнес-функцию и шаблон доступа и, следовательно, легко поддаются разделению на отдельные кластеры, но по мере продвижения вперед это разделение будет становиться все более тонким.

Теперь, когда мы продуманно разделили данные на основе бизнес-функций, поговорим о том, как масштабировать нагрузки, связанные с чтением, с помощью пулов чтения реплик.

Масштабирование чтения с помощью пулов чтения

Реплики в кластере могут служить для нескольких целей. Прежде всего они являются кандидатами на замещение операций записи, плановое или внеплановое, когда текущий источник по какой-либо причине должен быть выведен из эксплуатации. Но, поскольку эти реплики также постоянно выполняют обновления, чтобы соответствовать данным в источнике, вы можете использовать их и для обслуживания запросов на чтение.

На рис. 11.1 мы начинаем с визуального представления того, как выглядит новая установка с пулами реплик чтения.

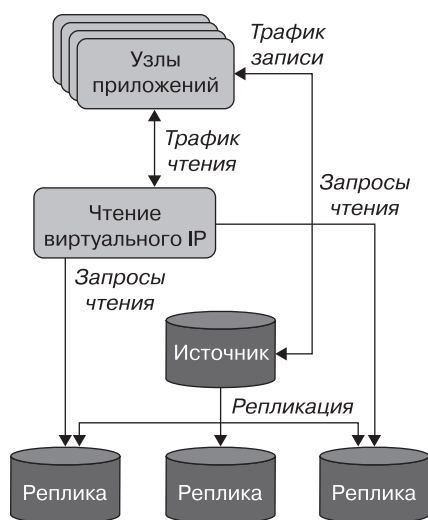


Рис. 11.1. Узлы приложений, использующие виртуальный IP-адрес для получения доступа к репликам для чтения

Для простоты предположим, что узлы приложений по-прежнему выполняют запросы на запись, напрямую подключаясь к исходной базе данных. Позже мы рассмотрим, как подключение к узлу-источнику может улучшить масштабирование. Однако обратите внимание на то, что те же узлы приложений подключаются к виртуальному IP-адресу, который действует как промежуточный слой между ними и репликами чтения. Это пул чтения реплик, и именно так вы распределяете растущую нагрузку чтения на несколько узлов. Вы также можете заметить, что не все реплики включены в пул. Это распространенный способ предотвратить влияние различных рабочих нагрузок чтения друг на друга. Если у вас есть процессы создания отчетов или процесс резервного копирования имеет тенденцию потреблять все ресурсы дискового ввода/вывода и вызывать задержку репликации, можете оставить один или несколько узлов реплики для выполнения этих задач и исключить их из пула чтения, обслуживающего клиентов. В качестве альтернативы можете дополнить проверку работоспособности балансировщика нагрузки проверкой репликации, которая автоматически удаляет отстающий резервный узел из пула и повторно вводит его, когда он догонит пул. Гибкость превращения реплик чтения во взаимозаменяемые ресурсы значительно возрастает, когда приложение обращается к одной точке для чтения, и вы можете управлять этими ресурсами плавно, не влияя на клиентов.

Теперь, когда более одного узла базы данных обслуживает запросы на чтение, для обеспечения бесперебойной работы в производственной среде необходимо учесть несколько моментов.

- Как направить трафик ко всем репликам чтения?
- Как равномерно распределить нагрузку?
- Как выполнить проверку работоспособности и удалить неработоспособные или отстающие реплики, чтобы избежать предоставления устаревших данных?
- Как избежать случайного удаления всех узлов, что может нанести еще больший ущерб трафику приложений?
- Как вручную удалить сервер для профилактического обслуживания?
- Как добавить новые серверы в балансировщик нагрузки?
- Какие автоматизированные проверки помогут избежать добавления нового узла в балансировщик нагрузки до того, как он будет готов?
- Достаточно ли конкретно ваше определение готовности к появлению нового узла?

Очень распространенным способом управления пулами чтения является использование балансировщика нагрузки для запуска виртуального IP-адреса, который действует как посредник для всего трафика, предназначенного для перехода к репликам чтения. Эти технологии включают HAProxy, аппаратный балансировщик нагрузки, если вы используете собственный хостинг, или сетевой балансировщик нагрузки, если работаете в публичной облачной среде. В случае использования HAProxy все хосты приложений будут подключаться к этому одному интерфейсу (фронтенду), а HAProxy позаботится о направлении запросов к одной из реплик чтения, определенных в бэкенде. Вот пример конфигурационного файла HAProxy, который определяет виртуальный IP-адрес интерфейса и в бэкенде сопоставляет его с несколькими репликами чтения в качестве внутреннего пула:

```
global
    log 127.0.0.1 local0 notice
    user haproxy
    group haproxy

defaults
    log global
    retries 2
    timeout connect 3000
    timeout server 5000
    timeout client 5000

listen mysql-readpool
    bind 127.0.0.1:3306
    mode tcp
    option mysql-check user haproxy_check
    balance leastconn
    server mysql-1 10.0.0.1:3306 check
    server mysql-2 10.0.0.2:3306 check
```

Как правило, для автоматического заполнения такого файла используется управление конфигурацией. В этой конфигурации есть несколько моментов, на которые стоит обратить внимание. Балансировка между узлами пула с помощью `leastconn` — рекомендуемый способ в MySQL. Случайная балансировка, такая как `roundrobin` во время повышенной нагрузки, не поможет вам использовать узлы, которые не перегружены. Убедитесь, что на ваших экземплярах MySQL создан соответствующий пользователь базы данных для выполнения этой проверки работоспособности, иначе все узлы будут помечены как неработоспособные.

Инструменты, облегчающие сегментирование, такие как Vitess и ProxySQL, могут действовать и как балансировщик нагрузки. Мы рассмотрим их в конце главы.

Управление конфигурацией пулов чтения

Теперь, когда у вас есть шлюз (gate) между узлами приложения и репликами, вам нужен удобный способ управления узлами, включенными или не включенными в этот пул чтения, с помощью выбранного балансировщика нагрузки. Вы не хотите, чтобы эта конфигурация управлялась вручную. Вы уже находитесь на пути масштабирования до большого количества экземпляров базы данных, а ручное управление файлами конфигурации приведет к ошибкам, замедлению времени отклика и сбоям узлов, и это просто не масштабируется.

Обнаружение служб — хороший вариант для автоматического обнаружения узлов, которые могут быть в этом списке. Это может означать развертывание решения для обнаружения служб как части технического стека или использование опции управляемого обнаружения сервисов у вашего облачного провайдера, если таковая имеется. Здесь важно быть очень внимательными и четко определить критерии, по которым копия чтения может быть включена в пул чтения. В идеале вы исключаете узел-источник и, возможно, одну или несколько реплик, предназначенных для создания отчетов. Но, может быть, вам нужно что-то еще более сложное, где реплики дополнительно сегментированы для обслуживания различных нагрузок чтения приложений? Мы рекомендуем использовать как минимум три узла на пул реплик, обслуживающих определенные цели, в дополнение к серверу резервного копирования/отчетности и узлу-источнику.

Независимо от того, запускаете ли вы собственный сервис обнаружения¹ или используете что-то предлагаемое облачным провайдером, вы должны знать о гарантиях этого сервиса. Вот несколько моментов, которые следует учитывать независимо от того, будете ли вы использовать сервис обнаружения или работать над ним вместе с командой.

- Как скоро он сможет обнаружить сбой узла?
- Как быстро распространяются эти данные?
- Как будет обновляться конфигурация балансировщика нагрузки при сбое экземпляра базы данных?
- Происходит ли смена элементов базы данных в фоновом режиме, или для этого потребуется разорвать существующие соединения?
- Что произойдет, если сама служба обнаружения сервисов выйдет из строя? Помешает ли это любым новым подключениям к базе данных или повлияет только на внесение изменений в состав балансировщика нагрузки? Сможете ли вы в этот момент внести изменения вручную?

¹ Мы наиболее часто используем и рекомендуем вам Consul от Hashicorp (<https://www.consul.io/>).

С гибкостью приходит сложность, и вы должны сбалансировать эти два фактора для достижения оптимальных результатов в производстве, когда случаются сбои. Ваша задача состоит в том, чтобы всегда привязывать свои решения к тому, какие SLI и SLO преследуются, а не в достижении мифической цели — 100%-ной безотказной работы.

Теперь, когда вы знаете, как заполнять конфигурации и обновлять их по мере того, как узлы приходят и уходят, пришло время поговорить о том, как выполнять проверки работоспособности для членов пула чтения реплик.

Проверки работоспособности пулов чтения

На этом этапе вам необходимо определить приемлемые критерии, по которым реплика чтения считается здоровой и готовой принимать трафик чтения от приложения. Эти критерии могут быть простыми, такими как «Процесс базы данных запущен и работает, порт отвечает», но могут стать и более сложными, например «База данных запущена, задержка репликации должна быть не более 30 с, запросы на чтение должны выполняться с задержкой не более 100 мс».



Проверьте состояние переменных `read_only` и `super_read_only`, чтобы убедиться, что все элементы в пуле чтения вашего балансировщика нагрузки действительно являются репликами.

Решение о том, как далеко следует заходить в проверке работоспособности, должно обсуждаться с командами разработчиков приложений, чтобы все понимали и были едины в том, какого поведения они ожидают при чтении из базы данных. Вот несколько вопросов, которые можно задать команде и которые помогут направить процесс принятия решений.

- Насколько допустимо устаревание данных? Если возвращенные данные устарели на несколько минут, на что это повлияет?
- Какова максимально допустимая задержка запроса для приложения?
- Какая имеется логика повторных попыток для запросов чтения и, если она существует, является ли она экспоненциальной?
- У нас уже есть SLO для приложения? Распространяются ли эти SLO на время ожидания запросов или только на время безотказной работы?
- Как поведет себя система при отсутствии запрошенных данных? Приемлема ли такая деградация? Если да, то на какой срок?

Во многих случаях вам будет достаточно использовать только проверку порта, чтобы убедиться, что процесс MySQL работает и может принимать соединения.

Это означает, что, пока база данных работает, она будет частью этого пула и будет обслуживать запросы.

Однако иногда вам может понадобиться что-то более сложное, потому что задействованный набор данных довольно критичен и вы не хотите обслуживать его при задержке репликации более чем на несколько секунд или если репликация вообще не выполняется. Для таких сценариев можно по-прежнему использовать пул чтения, но дополнить проверку работоспособности проверкой HTTP. Это работает следующим образом: выбранный вами балансировщик нагрузки выполняет команду (обычно скрипт) и на основе кода ответа определяет, исправен узел или нет. Например, в HAProxy серверная часть будет содержать такую строку кода:

```
option httpchk GET /check-lag
```

Она означает, что для каждого узла в пуле чтения балансировщик нагрузки будет вызывать путь `/check-lag` с помощью вызова `GET` и проверять код ответа. Этот путь запускает сценарий, который определяет, насколько допустимо отставание. Сценарий сравнивает существующий статус задержки с этим порогом, и в зависимости от этого балансировщик нагрузки считает реплику либо здоровой, либо нет.



Несмотря на то что проверки работоспособности являются мощным инструментом, будьте осторожны при использовании инструментов со сложной логикой (таких как описанная ранее проверка отставания) и убедитесь, что у вас есть план действий на случай, если все реплики в пуле не прошли проверку работоспособности. Вы можете иметь статический резервный пул, который возвращает все узлы в случае определенных глобальных сбоев (например, при задержке всего кластера), чтобы избежать случайного прерывания всех запросов на чтение. Для получения более подробной информации о том, как одна компания внедрила это, смотрите пост в блоге GitHub: <https://oreil.ly/zyjA4>.

Выбор алгоритма балансировки нагрузки

Существует множество различных алгоритмов для определения того, какой сервер должен получить следующее соединение. Каждый поставщик использует свою терминологию, но следующий список может дать представление о том, что существует.

- *Случайный*. Балансировщик нагрузки направляет каждый запрос на сервер, выбранный случайным образом из пула доступных серверов.
- *Round-robin*. Балансировщик нагрузки отправляет запросы на серверы в повторяющейся последовательности: А, В, С, А, В, С и т. д.

- *Наименьшее количество соединений.* Следующее соединение отправляется на сервер с наименьшим количеством активных подключений.
- *Самый быстрый ответ.* Сервер, который быстрее всех обрабатывал запросы, получает следующее соединение. Это может хорошо работать, когда пул содержит смесь быстрых и медленных машин, однако очень сложно для SQL, когда сложность запросов сильно варьируется. Даже один и тот же запрос может выполняться совершенно по-разному в разных обстоятельствах, например, когда он обслуживается из кэша запросов или когда кэши сервера уже содержат необходимые данные.
- *Хешированный.* Балансировщик нагрузки хеширует IP-адрес источника соединения, который сопоставляет его с одним из серверов в пуле. Каждый раз, когда запрос на соединение поступает с одного и того же IP-адреса, балансировщик нагрузки отправляет его на один и тот же сервер. Привязки меняются только тогда, когда изменяется количество машин в пуле.
- *Взвешенный.* Балансировщик нагрузки может комбинировать и добавлять вес нескольким другим алгоритмам. Например, у вас могут быть машины с одним и двумя процессорами. Двухпроцессорные машины примерно в два раза мощнее, поэтому можете указать балансировщику нагрузки отправлять им в среднем в два раза больше запросов.

Лучший алгоритм для MySQL зависит от рабочей нагрузки. Например, алгоритм наименьшего количества подключений может привести к переполнению новых серверов, если вы добавите их в пул доступных серверов до того, как их кэш прогреется.

Вам нужно будет поэкспериментировать, чтобы найти наилучшую производительность для своей рабочей нагрузки. Обязательно учитывайте то, что происходит в чрезвычайных обстоятельствах, а также в повседневной работе. Именно в чрезвычайных обстоятельствах — например, во время высокой нагрузки запросов, когда вы вносите изменения в схему или когда необычно большое количество серверов работает в автономном режиме — вы меньше всего можете допустить, чтобы что-то пошло не так.

Мы описали здесь только алгоритмы мгновенного предоставления соединения, которые не ставят запросы на подключение в очередь. Иногда алгоритмы, использующие очередь, могут оказаться более эффективными. Например, алгоритм может поддерживать заданный параллелизм на сервере базы данных, например разрешать не более N активных транзакций одновременно. Если активных транзакций слишком много, алгоритм может поставить новый запрос в очередь и обслужить его с первого сервера, который станет доступным в соответствии с заданными критериями. Некоторые пулы соединений поддерживают алгоритмы очередей.

Теперь, когда мы рассмотрели, как масштабировать нагрузку на чтение и проверять ее работоспособность, пришло время обсудить масштабирование операций записи. Прежде чем искать, как масштабировать непосредственно запись, можете рассмотреть места, где постановка в очередь может сделать рост трафика записи более управляемым. Обсудим, как постановка в очередь может помочь масштабировать производительность записи.

Очередь

Масштабирование прикладного уровня становится намного более сложным при масштабировании транзакций записи с хранилищем данных, которое в соответствии со своим строением отдает предпочтение согласованности, а не доступности. Большее количество узлов приложений, записывающих данные в один узел-источник, приведет к тому, что система базы данных станет более восприимчивой к тайм-аутам блокировок, взаимоблокировкам и неудачным попыткам записи, которые придется повторять. Все это в конечном счете приведет к ошибкам, с которыми сталкиваются пользователи, или к неприемлемым задержкам.

Прежде чем приступить к сегментированию данных, которое мы обсудим далее, вам следует изучить горячие точки записи ваших данных и подумать, действительно ли все операции записи необходимы для активного сохранения в базе данных. Можно ли поставить часть из них в очередь и записать в БД в удобное время?

Допустим, у вас есть база данных, в которой хранятся большие массивы исторических данных о клиентах. Клиенты время от времени отправляют API-запросы для получения этих данных, но вам также необходимо поддерживать API для удаления этих данных. Вы можете с уверенностью обслуживать вызовы API на чтение с растущего числа реплик, но как быть с удалением? HTTP RFC допускает код ответа **202 Accepted**. Вы можете вернуть этот код, поместить запрос в очередь (например, Apache Kafka или Amazon Simple Queue Service) и обрабатывать запросы в темпе, который не приведет к перегрузке базы данных непосредственно вызовами запросов на удаление.

Очевидно, это не то же самое, что код ответа 200, означающий, что запрос был выполнен мгновенно. Чтобы сделать требования к API правдоподобными и реализуемыми, решающее значение имеет обычная практика их обсуждения с командой разработчиков. Разница между кодами ответов 200 и 202 сводится к работе по проектированию сегментирования этих данных для поддержки большого количества параллельных операций записи.

Один из важных вариантов проектирования, который необходимо реализовать, если вы применяете постановку в очередь к нагрузке записи, заключается в том, чтобы заранее определить желаемый временной интервал, в течение которого эти вызовы должны быть выполнены после постановки в очередь. Мониторинг увеличения времени, которое запрос проводит в очереди, покажет, когда эта стратегия исчерпает себя и вам действительно нужно будет начать разделять этот набор данных для поддержки большей параллельности нагрузки записи. Вы можете сделать это, используя шардирование (sharding), которое мы обсудим далее.

Масштабирование операций записи с помощью шардирования

Если не можете справиться с ростом трафика записи с помощью оптимизированных запросов и постановки операций записи в очередь, то еще одной альтернативой является шардирование.

Сегментирование — это разделение данных на меньшие блоки (shards), обеспечивающие возможность выполнять одновременно больше операций записи на большем количестве узлов-источников. Существует два различных вида сегментирования, или разделения: функциональное разделение и шардирование данных.

Функциональное разделение, или разделение обязанностей, означает выделение разных узлов для разных задач. Примером этого может быть размещение записей о пользователях в одном кластере и выставление их счетов в другом кластере. Такой подход позволяет каждому кластеру масштабироваться независимо друг от друга. Резкий рост числа регистраций пользователей может создать нагрузку на кластер пользователей. При задействовании отдельных систем ваш биллинговый кластер не так сильно загружен, что позволяет выставять счета клиентам. И наоборот, если платежный цикл приходится на первое число месяца, вы можете запустить его, зная, что это не повлияет на регистрацию пользователей.

Шардирование данных — наиболее распространенный и успешный подход к масштабированию очень больших современных приложений MySQL. Вы сегментируете данные, разбивая их на более мелкие части или сегменты и сохраняя их на разных узлах.

Большинство приложений сегментируют только те данные, которые нуждаются в этом, — как правило, те части набора данных, которые будут очень большими. Предположим, вы создаете сервис ведения блога. Если вы ожидаете 10 млн пользователей, то сегментировать регистрационную информацию пользователя

может и не понадобиться, потому что все пользователи (или их активное подмножество) смогут полностью разместиться в памяти. Но если ожидаете 500 млн пользователей, вам, вероятно, следует сегментировать эти данные. Пользовательский контент, такой как сообщения и комментарии, почти наверняка потребует сегментирования в любом случае, поскольку эти записи намного объемнее и их гораздо больше.

Большие приложения могут иметь несколько логических наборов данных, которые вы можете сегментировать по-разному. Их можно хранить в разных наборах серверов, но это не обязательно. Вы также можете сегментировать одни и те же данные несколькими способами в зависимости от того, как получаете к ним доступ.

Будьте осторожны, планируя сегментировать только то, что нужно. Эта концепция должна включать не только быстро растущие данные, но и те, которые логически связаны с ними и будут регулярно запрашиваться в одно и то же время. Если выполняете сегментацию на основе поля `user_id`, но есть набор других, меньших таблиц, которые соединяются по тому же `user_id` в большинстве запросов, имеет смысл сегментировать все эти таблицы вместе, чтобы можно было обрабатывать большинство запросов приложения по одному сегменту за раз и избегать перекрестных соединений между базами данных.

Выбор схемы сегментирования

Самая важная и сложная задача, возникающая при шардировании, — это поиск и извлечение данных. Способ поиска данных зависит от того, как они шардированы. Существует множество способов сделать это — одни лучше, другие хуже.

Наша цель состоит в том, чтобы самые важные и часто встречающиеся запросы затрагивали как можно меньшее число шардов (помните, что один из принципов масштабируемости состоит в уменьшении перекрестных коммуникаций между узлами). Наиболее важной частью этого процесса является выбор ключа (или ключей) сегментирования для ваших данных. Ключ сегментирования определяет, в какой шард попадет та или иная строка. Если вам известен ключ сегментирования некоторого объекта, то вы можете ответить на два вопроса.

- Где следует хранить эти данные?
- Где можно найти запрошенные данные?

Далее мы покажем различные способы выбора и использования ключа сегментирования. А пока рассмотрим пример. Допустим, что мы поступаем так же, как MySQL NDB Cluster, и используем хеш первичного ключа каждой таблицы для распределения данных по шардам. Это очень простой подход, но он плохо масштабируется, поскольку часто заставляет искать информацию во всех шар-

дах. Например, если вам нужно найти все сообщения в блоге пользователя 3, где вы можете их обнаружить? Вероятнее всего, они равномерно распределены по всем шардам, потому что сегментировались по первичному ключу, а не по идентификатору пользователя. Использование хеша первичного ключа упрощает получение информации о том, где хранятся данные, но может затруднить их извлечение. Это зависит от того, какие данные нужны и знаете ли вы первичный ключ.

Как правило вам хотелось бы, чтобы ваши запросы были локализованы в одном шарде. При горизонтальном сегментировании данных вы всегда стремитесь избежать необходимости выполнять запросы на разных шардах при решении своей задачи. Объединение данных из многих шардов усложнит прикладной уровень и сведет на нет все преимущества шардирования данных. Самый худший случай с сегментированными наборами данных — это когда у вас нет ни малейшего представления о том, где хранятся нужные данные, так что приходится просматривать все шарды без исключения, чтобы найти их.

Обычно хорошим ключом сегментирования является первичный ключ какой-нибудь очень важной сущности в базе данных. Такие сущности определяют единицу шардирования. Например, если вы сегментируете свои данные по идентификатору пользователя или клиента, то единицей шардирования является соответственно пользователь или клиент. Для начала неплохо изобразить модель данных в виде диаграммы сущностей и связей 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 при помощи визуального изучения диаграммы можно найти кандидатов на ключи сегментирования, которых в противном случае вы бы не заметили. Однако не ограничивайтесь только просмотром диаграммы, примите во внимание также запросы, выполняемые приложением. Даже если между двумя сущностями существует какая-то связь, но соединение по ней выполняется редко или вообще никогда, то при шардировании эту связь можно не принимать во внимание.

Некоторые модели данных лучше поддаются шардированию, чем другие, — все зависит от степени связности графа «сущность — связь». На рис. 11.2 слева изображена модель данных, которая легко шардируется, а справа — которая шардируется с трудом.

Модель данных слева легко шардировать, потому что она имеет много связанных подграфов, состоящих в основном из узлов, между которыми есть всего одна связь, и можно относительно легко «разрезать» связи между подграфами. Модель справа сложно шардировать, потому что в ней таких подграфов нет. К счастью, большинство моделей данных больше похожи на левую диаграмму, чем на правую.

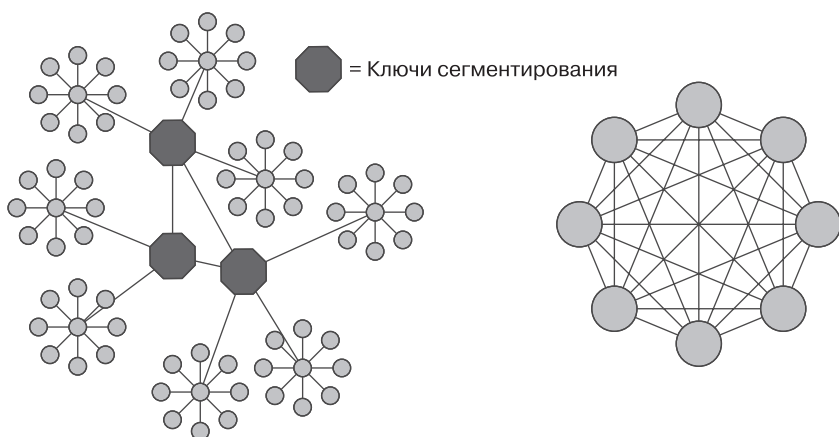


Рис. 11.2. Две модели данных — одну легко шардировать, а другую сложно¹

При выборе ключа сегментирования постарайтесь выбрать что-то, что позволит максимально избежать запросов к нескольким шардам, но при этом сделать шарды довольно маленькими, чтобы не было проблем с непропорционально большими фрагментами данных. Желательно, чтобы шарды были одинаково маленькими, если это возможно, а если нет, то по крайней мере небольшими, чтобы их можно было легко сбалансировать, объединив в группы разное количество шардов. Например, если приложение предназначено только для США и вы хотите разделить набор данных на 20 шардов, то, вероятно, не следует шардировать на основе отдельных штатов, потому что население Калифорнии очень велико. Однако можно выполнить шардирование данных по коду графства или телефонному коду местности: хотя шарды в этом случае не будут заполнены равномерно, их окажется достаточно для того, чтобы сформировать 20 наборов, которые в общей сложности будут примерно одинаково заполнены. Тогда вы сможете выбрать их так, чтобы избежать запросов к разным шардам.

Несколько ключей сегментирования

Сложные модели данных усложняют шардирование. Во многих приложениях имеется более одного ключа разделения, особенно если в данных можно выделить несколько важных параметров. Другими словами, приложению может понадобиться возможность эффективно взглянуть на информацию под разными углами зрения и получить целостное представление. Это означает, что вам может потребоваться хранить данные в своей системе, некоторые в двух разных видах.

¹ Спасибо проекту HiveDB и Бритту Кроуфорду за эти элегантные диаграммы.

Например, вам может понадобиться шардировать данные приложения для ведения блогов как по идентификатору пользователя, так и по идентификатору сообщения, поскольку и то и другое — распространенные способы просмотра данных приложением. Действительно, вы часто хотите видеть все сообщения одного пользователя и все комментарии к сообщению. Но шардирование по пользователю не поможет найти комментарии к сообщению, а шардирование по сообщению — найти сообщения для пользователя. Если вам нужно, чтобы оба типа запросов затрагивали только один шард, то придется шардировать обоими способами.

То, что вам нужны несколько ключей сегментирования, еще не означает, что придется проектировать два дублирующих друг друга хранилища данных. Рассмотрим другой пример — веб-сайт социальной сети клуба книголюбов, на котором пользователи могут оставлять комментарии о книгах. Сайт может отображать все комментарии к некоторой книге, а также все книги, которые данный пользователь прочитал и прокомментировал.

В таком случае вы можете создать одно шардированное хранилище с данными о пользователях, а другое — с данными о книгах. В комментариях хранится как идентификатор пользователя, так и идентификатор сообщения, поэтому они пересекают границы шардов. Вместо того чтобы полностью дублировать комментарии, вы можете хранить их вместе с данными о пользователях. Тогда вместе с данными о книгах достаточно хранить только заголовок и идентификатор комментария. Этого может быть достаточно для отображения большинства страниц, содержащих комментарии к книгам, без обращения к обоим хранилищам данных, а если нужно отобразить полный текст комментария, то вы можете получить его из хранилища данных о пользователях.

Запросы в разных шардах

В большинстве шардированных приложений встречаются по крайней мере несколько запросов, в которых необходимо агрегировать или соединять данные, хранящиеся в разных шардах. Например, если на сайте книголюбов нужно показать наиболее популярных или самых активных пользователей, то ему по определению необходимо обращаться ко всем шардам. Оптимизация подобных запросов — самая сложная часть реализации шардирования данных, так как то, что приложение видит как один запрос, на самом деле необходимо разделить и выполнить параллельно в каждом шарде. Хорошо написанный слой абстракции базы данных способен в какой-то мере облегчить решение этой задачи, но даже в этом случае такие запросы выполняются гораздо медленнее запросов к одному шарду, поэтому обычно необходимо применять также агрессивное кэширование.

Вы поймете, что выбранная вами схема шардирования данных была удачной, если запросы к нескольким шардам станут исключениями, а не нормой. Следует

стремиться сделать запросы как можно более простыми, чтобы они содержались в пределах одного шарда. Если требуется некоторая кросс-шардовая агрегация, рекомендуем сделать ее частью логики приложения.

Выполнение запросов к нескольким шардам можно ускорить за счет наличия сводных таблиц. Для их заполнения нужно обойти все шарды и по завершении обхода сохранить данные на серверах каждого шарда. Если дублирование данных в каждом шарде слишком расточительно, можно консолидировать все сводные таблицы в каком-нибудь специальном хранилище данных, чтобы они находились в одном месте. Данные, не относящиеся к шардам, часто размещают в глобальном узле с интенсивным кэшированием, чтобы защитить его от перегрузки.

Некоторые приложения используют, по существу, случайное шардирование, когда важно равномерное распределение данных или нет подходящего ключа сегментирования. Хорошим примером является приложение распределенного поиска. В этом случае запросы к нескольким шардам и агрегация являются нормой, а не исключением.

Запросы в разных шардах не единственная сложность, встречающаяся при организации шардирования. Трудно также поддерживать согласованность данных. Внешние ключи через границы шардов не действуют, поэтому обычно контроль ссылочной целостности выполняется на уровне приложения. Можно использовать внешние ключи внутри шарда, поскольку согласованность данных внутри него является самой важной задачей. Можно применять XA-транзакции, но из-за высоких накладных расходов на практике так поступают редко.

Можно также разработать процессы очистки, которые будут выполняться периодически. Например, если срок действия учетной записи пользователя книжного клуба истекает, не обязательно немедленно удалять ее. Вы можете написать периодическое задание для удаления комментариев пользователя из шарда с данными о книгах. Можно также подготовить скрипт проверки, который будет запускаться периодически и проверять согласованность информации в разных шардах.

Теперь, когда мы объяснили, как можно шардировать данные между несколькими кластерами и как выбрать ключ сегментирования, рассмотрим два самых популярных инструмента с открытым исходным кодом, которые могут помочь упростить как сегментирование, так и шардирование.

Vitess

Vitess — это система кластеризации баз данных для MySQL. Она была создана как внутренний продукт в YouTube, а затем стала PlanetScale, отдельным продуктом и компанией, соучредителями которой являются Джитен Вайдья и Сугу Сугумаране.

Vitess обеспечивает реализацию ряда функций:

- поддержку горизонтального шардирования, включая шардирование данных;
- управление топологией;
- управление отказоустойчивостью узла-источника;
- управление изменениями схемы;
- объединение подключений в пул;
- переписывание запросов.

Рассмотрим архитектуру Vitess и ее компоненты.

Обзор архитектуры Vitess

На рис. 11.3 представлена диаграмма с сайта Vitess, показывающая различные части ее архитектуры.

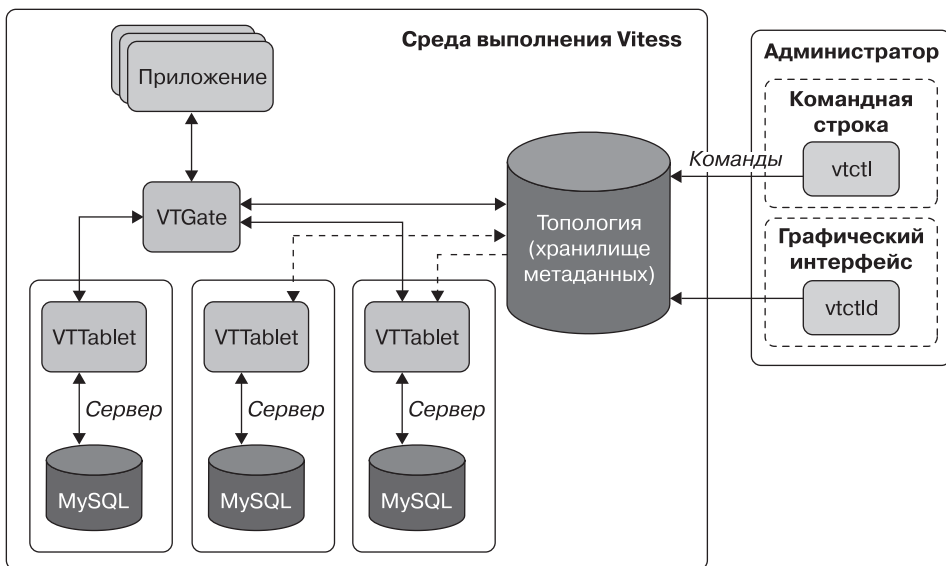


Рис. 11.3. Диаграмма архитектуры Vitess (заимствовано с сайта vitess.io)

Вот некоторые термины, которые вам необходимо знать.

- *Vitess pod* — общая оболочка набора баз данных и связанных с Vitess частей, которые поддерживают шардирование, управление топологией, управление изменениями схемы и доступ приложений к этим базам данных.

- *VTGate* — служба, которая контролирует доступ к экземплярам базы данных для приложений и операторов, желающих управлять топологией, добавлять узлы или сегментировать некоторые данные. Это похоже на балансировщик нагрузки в архитектуре, описанной ранее.
- *VTTablet* — агент, работающий в каждом экземпляре базы данных, управляемом Vitess. Он может получать команды управления базой данных от операторов и выполнять их от имени операторов.
- *Топология (хранилище метаданных)* — хранит перечень экземпляров базы данных, управляемых Vitess, в данном модуле, а также сопутствующую информацию.
- *Vtctl* — инструмент командной строки для внесения оперативных изменений в Vitess pod.
- *Vtctld* — графический интерфейс для тех же операций управления.

Архитектура Vitess начинается с согласованного хранилища топологии, которое содержит определения для всех кластеров, экземпляров MySQL и экземпляров *vtgate*. Это согласованное хранилище метаданных играет решающую роль в управлении изменениями топологии. Когда оператор хочет внести изменения в топологию кластера, управляемого Vitess, он на самом деле с помощью сервиса *vtctl* посылает команды в это хранилище данных, которое затем отправляет составные операции этой команды в *vtgate*.

Vitess предоставляет операторы баз данных, которые могут развернуть уровень *vtgate* и хранилище метаданных в Kubernetes. Наличие панели управления на платформе, аналогичной Kubernetes, повышает ее устойчивость к единым точкам отказа.

Одной из самых сильных сторон Vitess является ее подход к масштабированию MySQL, который включает в себя следующее.

- *Предпочтительное использование небольших экземпляров.* Разделяйте данные функционально, горизонтально или и так и этак. Однако меньшие экземпляры обеспечивают меньший объем разрушений, когда происходят сбои.
- *Репликация и автоматическое восстановление после отказа при записи для повышения отказоустойчивости.* Vitess не обещает 100%-ную онлайн-запись с помощью трюков с несколькими узлами записи. Вместо этого он автоматизирует восстановление после отказа записи и во время аварийного восстановления управляет как изменением топологии, так и доступом приложений к узлам базы данных, чтобы сократить время простоя при записи настолько, насколько это возможно.
- *Долговечность с использованием полусинхронной репликации.* Vitess настоятельно рекомендует использовать полусинхронную репликацию (в отличие от асин-

хронной по умолчанию), чтобы гарантировать, что записи всегда сохраняются более чем одним узлом на уровне базы данных, прежде чем подтверждать их приложению. Это важнейший компромисс между задержкой и гарантированной долговечностью, который приносит свои дивиденды, когда Vitess необходимо незапланированным образом переключать узел записи на другой ресурс.

Эти архитектурные принципы могут помочь поддерживать экспоненциальный рост бизнес-трафика с гораздо большей устойчивостью на уровне базы данных в вашей инфраструктуре. При этом многие из этих лучших практик следует применять, независимо от того, используете ли вы Vitess или другое решение в качестве части своей архитектуры.

Миграция вашего стека на Vitess

Vitess — это платформа для работы уровня базы данных, которая не является готовым решением. Поэтому необходимо тщательно спланировать, как будет осуществляться переход на Vitess, прежде чем вы примете ее в качестве уровня доступа для своей базы данных.

В частности, оценивая Vitess как возможное решение, обязательно рассмотрите следующие шаги по переходу на эту платформу.

1. *Протестируйте и задокументируйте задержки, которые вы вносите в общую систему.* Внедрение такого сложного стека, как Vitess, в стек приложений, безусловно, добавит некоторые задержки, особенно если учесть применение полусинхронной репликации. Убедитесь, что этот компромисс хорошо задокументирован и явно указан, чтобы для зависимых систем можно было принимать обоснованные решения при создании SLO, которые полагаются на эту архитектуру базы данных.
2. *Используйте модель развертывания *canary** (<https://oreil.ly/ldtnN>). Во время перехода в производство можете настроить `vtablet` как управляемый извне. Это позволяет использовать как `vtablet`, так и прямые соединения с сервером базы данных по мере постепенного увеличения количества соединений через свой набор прикладных узлов.
3. *Запустите шардирование.* После того как весь доступ к прикладному уровню будет осуществляться через `vtgate/vtablet`, а не напрямую к MySQL, можете начать использовать полный набор функций Vitess для разделения таблиц на новые кластеры, горизонтального разделения данных для увеличения пропускной способности при записи или просто добавить реплики для увеличения нагрузки при чтении¹.

¹ Эту стратегию развертывания подробно объяснил Морган Токер в своем выступлении на Kubecon 2019 (<https://www.youtube.com/watch?v=OCS45iy5v1M>).

Vitess — это мощный продукт для доступа к базам данных и управления ими, который прошел долгий путь с момента своего появления в Google. Он доказал свою способность обеспечить значительный рост и устойчивость инфраструктуры баз данных. Однако за эту мощь и гибкость приходится платить повышенной сложностью. Vitess не так прост, как балансировщик нагрузки, пропускающий трафик, и вы должны сопоставить потребности бизнеса и стоимость внедрения и обслуживания такого сложного инструмента управления базой данных.

ProxySQL

ProxySQL написан специально для протокола MySQL и выпущен под лицензией General Public License (GPL). Его основным автором является Рене Канно, администратор базы данных, который консультировал многие компании, и давний участник проекта MySQL. Сейчас это полноценная компания, которая предлагает платные контракты на поддержку и разработку продукта ProxySQL.

Рассмотрим некоторые детали его архитектуры, шаблоны конфигурации, примеры использования и возможности.

Обзор архитектуры ProxySQL

Можно использовать ProxySQL в качестве промежуточного слоя между любым кодом приложения и экземплярами MySQL. ProxySQL предоставляет основанный на MySQL-протоколе сеансно-ориентированный интерфейс для взаимодействия приложений с базами данных. Вместо того чтобы приложения открывали соединения непосредственно с экземплярами базы данных, ProxySQL открывает их от имени приложений.

Такая конструкция делает прокси-сервер невидимым для узлов приложения. Его информированность о сеансе позволяет перемещать эти соединения между экземплярами MySQL без простоев. Это особенно полезно, когда вы имеете дело с приложениями, на которые больше не тратите усилия, потому что теперь можете использовать функции ProxySQL вместо необходимости вносить в код какие-либо изменения, в которых вы, возможно, не уверены.

ProxySQL также обеспечивает мощный пул соединений. Открываемые приложениями соединения с ProxySQL изолированы от соединений, которые ProxySQL открывает с экземплярами базы данных, для подключения к которым он настроен. Такое разделение позволяет защитить экземпляры базы данных от внезапных скачков трафика на прикладном уровне.

Когда у вас есть возможность управлять соединениями на стороне клиента отдельно от того, сколько соединений с базой данных фактически установлено,

вы получаете гибкость, которой раньше не было. Теперь можно масштабировать пул узлов приложения, не беспокоясь о том, что это увеличит нагрузку на подключение к базе данных сверх того, что вы хотите поддерживать. Это позволяет реализовать различные сценарии приложений и бизнес-потребностей, как мы объясним в общих шаблонах использования ProxySQL.

Настройка ProxySQL

ProxySQL использует конфигурационный файл для запуска, но во время выполнения сохраняет свою конфигурацию как в памяти, так и во встроенном файле SQLite, к которому вы можете получить прямой доступ и выполнить запрос с помощью интерфейса администратора.

Интерфейс администратора ProxySQL позволяет выдавать команды для изменения текущей конфигурации, а затем с помощью команд MySQL выгружать новую конфигурацию для сохранения на диск. Это дает возможность вносить изменения в работающий экземпляр ProxySQL с нулевым временем простоя. Вы также можете использовать интерфейс администратора для внесения автоматических изменений, выдаваемых вашей системой управления конфигурацией или скриптами автоматизированного восстановления после отказа. На рис. 11.4 видно, как архитектура обычно использует ProxySQL и обнаружение сервисов для обеспечения надежного уровня доступа к последним.

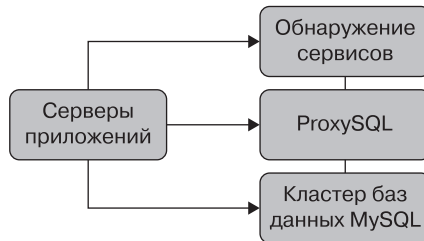


Рис. 11.4. Взаимодействие между узлами приложений, ProxySQL и обнаружением сервисов (адаптировано из диаграммы Билла Сиклза)



Важно отметить, что, хотя на этой диаграмме ProxySQL показан как один объект, мы настоятельно рекомендуем в производственных средах применять механизм кластеризации и развертывать несколько экземпляров в используемом вами стеке. Никогда не допускайте единой точки отказа (SPoF).

ProxySQL включает независимую иерархическую проверку работоспособности баз данных, к которым он подключается. Основываясь на результатах этих

проверок, ProxySQL добавляет или удаляет узлы или регулирует веса трафика. Чтобы контролировать допустимую степень отказоустойчивости в контексте потребностей вашей службы и приложения, можете указать пороговые значения задержки репликации, время успешного подключения, повторные попытки подключения при сбое и много других параметров конфигурации. Эти параметры позволяют ProxySQL точно реагировать на неотвечающие узлы, либо временно удаляя серверные базы данных и позже повторяя проверку работоспособности, либо полностью удаляя проблемный серверный элемент до тех пор, пока не будет привлечен оператор.

Использование ProxySQL для шардирования

ProxySQL очень полезен для различных топологий шардирования. Хотя он не обеспечивает автоматизацию фактического разделения данных, как это делает Vitess, но может быть отличным облегченным промежуточным слоем, который осведомлен о шардировании и способен соответствующим образом маршрутизировать соединения приложения. Рассмотрим различные способы его использования в качестве уровня маршрутизации для ваших шардов.

Шардирование по пользователям. Если данные разделены функционально или по бизнес-функциям в разных кластерах баз данных и разные группы приложений получают доступ к этим кластерам, вы должны использовать различные учетные данные базы данных для каждого из этих приложений. ProxySQL может использовать этот пользовательский параметр для маршрутизации трафика на изолированные пулы внутренних баз данных для записи или чтения.

Вы можете настроить такую маршрутизацию в ProxySQL, выполнив следующие команды в интерфейсе администратора, а затем сохранив изменения в файле конфигурации на диске:

```
INSERT INTO mysql_users
(username, password, active, default_hostgroup, comment)
VALUES
('accounts', 'shard0_pass', 1, 0, 'Routed to the accounts shard'),
('transactions', 'shard1_pass', 1, 1, 'Routed to the transactions shard'),
('logging', 'shard2_pass', 1, 2, 'Routed to the logging shard');

LOAD MYSQL USERS RULES TO RUNTIME;
SAVE MYSQL USERS RULES TO DISK;
```



Чтобы избежать неприятных сюрпризов при перезапуске процесса ProxySQL, убедитесь, что вы всегда синхронизируете конфигурацию ProxySQL среды выполнения и конфигурацию на диске.

Это повышает удобство регистрации всех операций, выполняемых этими пользователями, чтобы соответствовать требованиям, не создавая никакой нагрузки на базу данных. В главе 13 вы увидите, что мы также рекомендуем разделять пользователей базы данных по соображениям соответствия требованиям, и, следовательно, этот дизайн также согласуется с некоторыми целями соответствия.

Шардирование по схемам. Еще один способ использования ProxySQL для поддержки шардированных наборов данных — применение имен схем в качестве правил для управления маршрутизацией трафика. Вот пример того, как это можно определить в конфигурации ProxySQL:

```
INSERT INTO mysql_query_rules (rule_id, active, schemaname,  
destination_hostgroup, apply)  
VALUES  
(1, 1, 'shard_0', 0, 1),  
(2, 1, 'shard_1', 1, 1),  
(3, 1, 'shard_2', 2, 1);  
  
LOAD MYSQL QUERY RULES TO RUNTIME;  
SAVE MYSQL QUERY RULES TO DISK;
```

Обратите внимание на то, что эта конфигурация может быть использована как для горизонтального, так и для функционального шардирования при условии, что вы правильно назовете свои схемы.

Последняя важная рекомендация по применению ProxySQL, таким образом, заключается в том, чтобы обязательно использовать его встроенную функцию кластеризации, которая гарантирует, что критически важная таблица конфигурации, такая как `mysql_rules`, синхронизируется со всеми узлами ProxySQL в кластере, обеспечивая резервирование на уровне промежуточного программного обеспечения.

Другие преимущества использования ProxySQL

Обсудим некоторые распространенные шаблоны, в которых использование ProxySQL может помочь решить часто встречающиеся проблемы в быстрорастущих средах.

Когда задержка выполнения запросов начинает расти во многих приложениях, мы обычно замечаем шаблон «Открыть больше подключений к базе данных». Однако на практике это может привести к сбоям в работе¹ и, как правило, оставляет множество соединений бездействующими, потребляя ресурсы, но

¹ Для получения дополнительной информации см. статью в «Википедии» о проблеме громоподобного стада (Thundering herd problem <https://oreil.ly/YOtAt>).

не выполняя никакой работы. Когда вы открываете на прикладном уровне больше соединений непосредственно к базе данных, количество ресурсов, которые сервер базы данных тратит на управление соединениями, также увеличивается. Это приводит к появлению тысяч соединений, перегружающих и без того перегруженные экземпляры баз данных. Все эти действия приводят к длительным простоям, каскадным сбоям в нескольких микросервисах и длительному воздействию на клиентов.

Архитектура управления соединениями ProxySQL помогает защитить уровень базы данных от неожиданных пиковых нагрузок приложений, открывая только то количество соединений с базой данных, которое может выполнять работу. ProxySQL может повторно использовать эти соединения для различных запросов со стороны клиента. Такое поведение максимизирует работу, которую может выполнить одно соединение с серверами баз данных, что, в свою очередь, уменьшает количество ресурсов, управляющих соединениями, и позволяет более эффективно задействовать ресурсы памяти сервера баз данных.

Другие важные особенности ProxySQL

ProxySQL имеет ряд других особенностей, которые выделяются среди прокси-приложений общего назначения. Это:

- маршрутизация запросов на основе порта, пользователя или совпадения регулярных выражений;
- поддержка TLS как для интерфейсных подключений к приложениям, так и для серверных подключений к базам данных;
- поддержка различных версий MySQL, таких как AWS Aurora, Galera Cluster и Clickhouse;
- зеркалирование соединений;
- кэширование результирующего набора;
- перезапись запросов;
- журнал аудита.

Вы можете прочитать об обширном наборе функций ProxySQL (который выходит далеко за рамки поддержки шардирования), ознакомившись с его документацией (<https://oreil.ly/PTZFW>).

ProxySQL — это мощный инструмент, который можно использовать для масштабирования приложения с надлежащей защитой производительности на уровне базы данных и с дополнительными функциями, которые поддерживают всевозможные потребности бизнеса (например, соответствие требованиям, правила безопасности и т. д.). Если ваша компания находится на траектории

быстрого роста и в ней удачно сосуществуют новые и давно работающие сервисы, использующие ресурсы базы данных, то ProxySQL может стать мощным инструментом для обеспечения дальнейшего безопасного роста. ProxySQL предоставляет простую в развертывании абстракцию, которая может быть более сложной, чем HAProxy, но с меньшими начальными затратами на инфраструктуру и сложность. Однако он не включает некоторые из продвинутых функций Vitess, таких как автоматическое разделение наборов данных, управление изменениями схем и VReplication, которая является мощным инструментом для создания конвейеров извлечения, преобразования, загрузки (ETL) и изменения потоков данных.

Резюме

Масштабирование MySQL — это путешествие. Вы должны закончить эту главу более подготовленными, чтобы оценить свои потребности в масштабировании и понять, как масштабировать чтение и запись и как сделать рост трафика более предсказуемым, добавив очереди в свою архитектуру. Теперь вы также должны понимать, что такое шардирование для масштабирования записи и как принимать сложные решения, которые с ним связаны.

Прежде чем углубляться в узкие места масштабируемости, убедитесь, что вы оптимизировали свои запросы, проверили индексы и имеете надежную конфигурацию MySQL. Это может дать вам время, необходимое для планирования лучшей долгосрочной стратегии. После оптимизации сосредоточьтесь на определении того, привязаны ли вы к чтению или записи, а затем подумайте, какие стратегии лучше всего подходят для решения любых неотложных проблем. При планировании решения обязательно продумайте, как настроиться на долгосрочную масштабируемость.

Когда рабочие нагрузки связаны с чтением, мы рекомендуем перейти на пулы чтения, если только задержка репликации не оказывается проблемой, которую невозможно преодолеть. Если проблема связана с задержкой или записью, то в качестве следующего шага необходимо рассмотреть шардирование.

ГЛАВА 12

MySQL в облаке

По всей вероятности, вы не сможете повлиять на то, перейдет ли ваша организация к облачному провайдеру и какого из них в конечном итоге выберет. От вас зависит лишь построение среды баз данных. Вы можете выбрать два пути: управляемую MySQL или построение среды на виртуальных машинах. Управляемая MySQL, как правило, более проста в использовании, но обычно дороже и обеспечивает вам меньший контроль. Создание среды на виртуальной машине означает, что вы получаете гораздо бóльшую гибкость в том, как строить свою платформу и наблюдать за ней, но это дольше и требует более весомых операционных накладных расходов.

В этой главе поговорим об основных возможностях управляемой MySQL и о том, чем они могут быть вам полезны. Мы также объясним, как приступить к созданию виртуальной машины, включая выбор подходящих спецификаций и типов дисков, и рассмотрим операционные сложности, такие как перезагрузка хоста, к которым вы должны подготовиться при запуске MySQL на виртуальных машинах в облаке.



Мы не будем рассматривать ошибки в предложениях облачных провайдеров. Эти предложения представляют собой постоянно развивающиеся продукты, поэтому рекомендуем следить за новостями в динамичных источниках, таких как информационные бюллетени или доски объявлений об ошибках, а не ограничиваться справочными материалами на определенный момент времени, таких как эта книга.

Управляемая MySQL

Предложения по управляемой MySQL от облачных провайдеров очень удобны для команд, желающих уменьшить трудоемкость эксплуатации MySQL по мере роста их продукта и расширения набора функций. Каждое публичное облако имеет собственную интерпретацию того, как должна выглядеть и работать управляемая база данных SQL. Amazon Web Services (AWS) предлагает несколько

разновидностей Aurora MySQL (вскоре мы подробно обсудим их), Google Cloud Platform (GCP) предлагает Cloud SQL и т. д., и почти все провайдеры публичных облаков предлагают нечто подобное.

Основная привлекательность управляемых решений заключается в том, что они обеспечивают доступную настройку базы данных без необходимости углубляться в специфику MySQL. С помощью всего нескольких щелчков кнопкой мыши или приложения Terraform вы можете в режиме онлайн создать готовую к работе базу данных с репликой и запланированными резервными копиями. Это может быть очень привлекательным вариантом для компаний или команд, которые хотят быстро приступить к работе.

В то же время при использовании управляемой MySQL вы лишаетесь значительной обзорности и контроля. У вас нет доступа к операционной или файловой системе, и вы ограничены в том, что можете сделать в рамках процесса. Вы не можете проверить в системе ничего другого, кроме того, что предоставляет вам облачный провайдер. В большинстве случаев при возникновении проблемы вам остается только отправить заявку в службу поддержки и ждать ответа. Вы не можете настраивать расширенные топологии, а ваши методы резервного копирования и восстановления ограничены тем, что предлагает облачный провайдер.

Стоит отметить, что многие из облачных предложений предоставляют вам хранилище данных, совместимое с MySQL. Это хранилище данных с интерфейсом SQL, но с внутренними механизмами, которые могут кардинально отличаться от Oracle MySQL. Мы рассмотрим общие компромиссы и отличия каждого управляемого решения, чтобы помочь выбрать вариант, который лучше всего подходит для вашей команды и потребностей бизнеса.

Amazon Aurora для MySQL

Aurora MySQL — это сервис совместимой с MySQL базы данных. Наиболее привлекательным преимуществом Aurora является то, что она отделяет вычислительные ресурсы от хранилища, что позволяет им масштабироваться обособленно и более гибко. Aurora управляет рядом операционных задач, которые вы обычно выполняете, например созданием резервных копий моментальных снимков, управлением быстрыми изменениями схемы, ведением журнала аудита и управлением репликацией в пределах одного региона.

Существуют различные предложения Aurora MySQL. Мы вкратце рассмотрим разницу между этими предложениями.

Стандартное предложение Aurora — это длительно работающие вычислительные экземпляры, где вы выбираете класс экземпляра (точно так же, как при запуске собственной MySQL) и получаете подключенное хранилище с внутренней репликацией на шесть копий.

ЗАМЕЧАНИЕ О СОВМЕСТИМОСТИ

Когда Amazon говорит «совместима с MySQL», необходимо уточнить, какая основная версия MySQL подразумевается в этой фразе. Например, ни одно из решений Aurora не совместимо с MySQL 8.0, а некоторые из более старых решений совместимы только с MySQL 5.6. Если вы рассматриваете возможность перехода с самоуправляемой MySQL на Amazon Aurora MySQL, обязательно учтите это при тестировании приложений перед переносом производственных данных.



На момент написания этой книги быстрый DDL в Aurora рассматривался AWS как функция экспериментального режима. Если вы читаете эту книгу и это осталось неизменным, рекомендуем обратиться к главе 6, чтобы узнать больше о возможностях изменения схемы в режиме онлайн с помощью инструментов, внешних по отношению к базе данных.

При этом важно отметить, что репликация в кластере Aurora запатентована компанией Amazon и не является репликацией, которую мы знаем и используем в Oracle MySQL. Поскольку все экземпляры Aurora в кластере задействуют один и тот же уровень хранения для доступа к данным, репликация внутри кластера осуществляется с помощью блочного хранилища¹. Однако Aurora поддерживает запись двоичных журналов в формате, знакомом нам по бесплатной общедоступной версии MySQL, для команд, которые реплицируют данные с кластера Aurora на другой кластер, или для любых других целей применения двоичных журналов, например для сбора данных об изменениях².



Если вы хотите разместить на Aurora критически важные базы данных, настоятельно рекомендуем рассмотреть возможность использования прокси-сервера Amazon RDS Proxy для управления тем, как ваше приложение будет взаимодействовать с Aurora. В ситуациях, когда существует вероятность всплеска новых подключений со стороны приложения, RDS Proxy может оказаться очень полезным, не позволяя массе новых соединений повлиять на работу базы данных.

С момента появления Aurora MySQL на сцене в 2015 году AWS расширила ее возможности, которые вы можете применять для реализации самого широкого круга сценариев использования и бизнес-потребностей.

¹ Если вы действительно хотите узнать подробности об этой архитектуре, настоятельно рекомендуем документ SIGMOD (<https://oreil.ly/hhFhU>), который команда Aurora опубликовала в 2017 году.

² Сбор данных об изменениях — это шаблон проектирования в архитектуре данных, который используется для определения момента изменения данных и передачи этих изменений между доменами и системами. Для получения дополнительной информации об этом настоятельно рекомендуем главу 11 книги Мартина Клеппманна «Высоконагруженные приложения. Программирование, масштабирование, поддержка» (Питер, 2019).

- *Aurora Serverless.* Бессерверный режим Aurora MySQL (Serverless) устраняет необходимость в длительных вычислениях и задействует бессерверную платформу Amazon для обслуживания вычислительного уровня вашей базы данных. Это дает вам большую гибкость в затратах, если рабочая нагрузка не должна выполняться постоянно.
- *Глобальная база данных Aurora.* Это решение Aurora для тех, кому необходимо иметь данные в нескольких географических регионах, но не хочется использовать двоичную репликацию журнала для ручного управления передачей изменений данных из основного кластера в кластеры в других регионах. Обратите внимание на то, что это связано с компромиссами, поэтому всегда следует обращаться к документации Amazon, чтобы убедиться, что вы принимаете правильные решения.
- *Aurora Multi-Master.* Multi-Master — это особая разновидность кластеров Aurora, которые могут принимать записи более чем на один вычислительный узел одновременно. Он задуман как высокодоступное решение, когда доступность записи в одном регионе является наивысшим приоритетом. Обратите внимание на то, что Aurora Multi-Master поставляется с собственным набором ограничений. Во-первых, на момент написания книги он работает на ядре сервера MySQL 5.6, что не позволяет использовать ряд функций. В кластере существует ограничение на максимальное количество узлов, и вы не можете смешивать Multi-Master и Global Database в одном развертывании. Мы считаем Aurora Multi-Master очень продуманным решением для выбора доступности и согласованности при каждом взаимодействии с хранилищем данных и приложениями, но рекомендуем тщательно взвесить заявленные ограничения и рассмотреть компромиссы, прежде чем выбирать его.

AWS продолжает обновлять и улучшать управляемые реляционные базы данных, которые предлагает, поэтому не будем глубоко вдаваться в подробности различий в функциях между вариантами Aurora. На рис. 12.1 представлена блок-схема, которая поможет вам понять, какой тип Aurora лучше всего подойдет для ваших нужд и каких компромиссов потребует.

На рисунке показано базовое дерево решений, которое поможет выбрать один из вариантов Aurora. Важно то, что, хотя у Aurora есть несколько вариантов, всегда имеются компромиссы. Например, вы не можете обеспечить одновременно высокую доступность с несколькими записями и межрегиональную репликацию за доли секунды. Но можете использовать предложения AWS, чтобы представить эти компромиссы и подробно обсудить, что важнее: доступность записи или региональная репликация продукта.

Aurora не единственное предложение управляемой MySQL от облачного провайдера. У GCP есть собственный вариант.

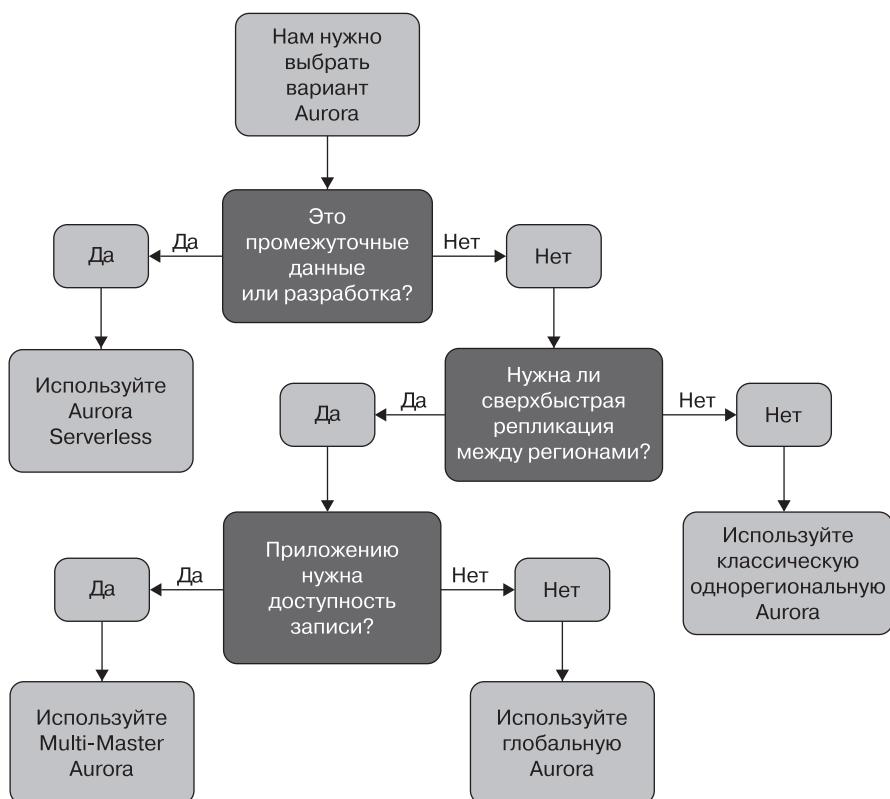


Рис. 12.1. Блок-схема, которая поможет выбрать наиболее подходящий вариант Aurora

GCP Cloud SQL

Cloud SQL — это управляемое решение GCP для MySQL. Основное отличие этого предложения от предложения AWS заключается в том, что на нем работает сервер бесплатной (community) версии MySQL, правда, некоторые функции у него отключены специально для того, чтобы обеспечить мультиарендность и управляемость продукта. Вот кое-что из того, что вы не можете использовать в Cloud SQL, несмотря на то что он работает на бесплатном сервере:

- привилегия SUPER отключена;
- загрузка плагинов отключена¹;
- некоторые клиенты также отключены, например `mysqldump` и `mysqlimport`.

¹ Cloud SQL предлагает собственное решение для ведения журналов аудита для обеспечения соответствия нормативным требованиям (<https://oreil.ly/RM7MW>).

Как и в случае с решением AWS, вы не можете получить доступ к экземплярам MySQL по SSH.

Однако имеется ряд операционных задач, которые Cloud SQL решает за вас.

- Встроенная поддержка высокой доступности. Отказоустойчивость автоматизируется с помощью параметра конфигурации.
- Встроенное шифрование данных для хранения (at rest).
- Гибко управляемые обновления с использованием различных методов. Обратите внимание на то, что в конечном итоге окна обслуживания вызывают появление некоторого времени простоя (аналогично AWS Aurora) и вы несете ответственность за то, чтобы сбалансировать это с SLO приложений¹.

Как упоминалось в начале этой главы, скорее всего, вы не сможете выбирать, у какого облачного провайдера создавать эти базы данных, поэтому, вероятно, потребуется знать, что предлагает управляемый вариант от определенного для вас облачного провайдера и как с ним работать. Или нужно обосновать необходимость перехода к использованию виртуальных машин напрямую вместо управляемого MySQL.

Теперь, когда мы рассмотрели варианты управляемых реляционных баз данных и тонкости их выбора, поговорим о чуть более сложном пути — запуске MySQL на виртуальных машинах, размещенных в облаке.

MySQL на виртуальных машинах

Возможности управляемого MySQL могут быть очень привлекательными для тех, кто хочет быстро запустить систему, так почему же кто-то принимает решение запустить свою собственную? Запуск MySQL на виртуальной машине аналогичен запуску на голом «железе» (bare metal). Вы получаете полный и абсолютный контроль над всеми операционными аспектами. Можете запустить основной экземпляр MySQL в одном регионе, но настроить реплики в других регионах для целей аварийного восстановления или запустить реплику с временной задержкой. Вы можете также настроить метод резервного копирования таким образом, чтобы он был наиболее оптимальным для вашей рабочей нагрузки. Если производительность снижается или возникают проблемы, у вас есть полный контроль над операционной и файловой системами, что позволяет выполнять любой анализ.

¹ Дополнительные сведения см. в разделе «Минимизация влияния обслуживания» в документации по Cloud SQL (<https://oreil.ly/3kNIh>).

Типы машин в облаке

Как обсуждалось в главе 4, количество ядер процессора и доступная оперативная память для MySQL оказывают непосредственное влияние на производительность MySQL. Недостатком выбора конкретных технических характеристик оборудования для центра обработки данных является то, что их нельзя легко изменить. Если в стойке стоит машина с 56 ядрами и 512 Гбайт оперативной памяти, безусловно, можно уменьшить объем установленной оперативной памяти, но вы уже заплатили за нее, поэтому, если не сможете повторно использовать оперативную память где-то в другом месте, то, возможно, вы слишком много израсходовали на оборудование.

Оптимизировать технические характеристики машины для вашей рабочей нагрузки гораздо проще, если вы пользуетесь услугами облачного провайдера. Основные облачные провайдеры позволяют выбрать спецификацию машины, которая устанавливает диапазон виртуальных процессоров (vCPU), объем доступной оперативной памяти, а также сетевые и дисковые ограничения. Вместе с этим появляется возможность изменять размер виртуальных машин по мере колебаний рабочей нагрузки. Это означает, что, если в определенные периоды, например в праздники, наблюдается пиковый трафик, можно временно увеличить технические характеристики машины, чтобы учесть это. Как только трафик снова снизится, вы сможете уменьшить размер машин. Именно благодаря такой гибкости многие переходят на облачные технологии.

Выбор правильного типа машины

Если вы уже пользуетесь услугами облачного провайдера, выбрать машину довольно просто. Столкнувшись с узким местом в виде vCPU, памяти или сети, можно найти подходящий тип машины для решения этой проблемы и изменить ее размер. Однако, если вы переезжаете из центра обработки данных, заранее определить правильную конфигурацию может быть непросто.

Центральный процессор

В главе 4 мы рассказали о том, как правильно выбрать процессор для своей рабочей нагрузки. Когда вы переходите в облако, большинство этих рекомендаций остаются актуальными. Помните, что, работая с облачными провайдерами, вы получаете виртуальные, а не физические процессоры. Это означает, что процессор не принадлежит исключительно вам. Его можно использовать совместно с другими арендаторами на одном и том же физическом узле. По всей вероятности, вы увидите больше различий в задержке и загрузке, чем на собственном эксклюзивном сервере.

При переходе с физических машин на облачного провайдера может быть непросто оценить и загрузку своего процессора. Для подсчета виртуальных процессоров мы успешно использовали следующую формулу: (Количество ядер $\times$ 95 % общей загрузки) $\times$ 2.

Предположим, у вас есть 40-ядерный сервер в центре обработки данных. За последние 30 дней пиковая загрузка процессора составляла 30 %. Сколько ядер нужно, чтобы запустить его у облачного провайдера при 50%-ной загрузке? Используя приведенную формулу, мы оценили бы их количество в 24 ядра. Если выбранный вами облачный провайдер не предлагает 24-ядерный тип машины, рассмотрите возможность округления до ближайшего типа или определите, предлагает ли провайдер нестандартные типы машин¹.



По мере увеличения использования процессора или количества ядер увеличивается и переключение контекста — переключение задач на процессоре. В связи с этим не стоит работать на 100%-ной мощности процессора, поскольку вы будете тратить много времени на переключение между потоками. Это проявится в виде задержки при выполнении запросов. Мы рекомендуем применять стандартную загрузку процессора на уровне 50 %, пиковая загрузка может достигать 65–70 %. Если вы поддерживаете загрузку процессора 70 % или выше, то, скорее всего, увидите увеличение задержки. В этом случае следует рассмотреть возможность увеличения количества процессоров.

Обратите также внимание на семейство процессорного чипа, если это возможно. Если вы используете веб-приложение с высокой посещаемостью, вероятно, необходимо убедиться, что применяется чип более позднего поколения. А если занимаетесь серверной обработкой данных, то подойдут более старые, немного более медленные семейства процессорных чипов, что, возможно, позволит сэкономить средства.

Память

Как говорилось в главах 1 и 4, оперативная память может сильно повлиять на производительность MySQL.

Выбирайте машину любой спецификации, которая лучше всего соответствует вашим потребностям и рабочему набору данных, при этом предпочтительнее ошибиться в сторону слишком большого объема оперативной памяти, чем ее недостатка.

¹ Имейте в виду, что нестандартные типы машин могут стоить дороже, чем заранее определенные. Работая с большим количеством экземпляров, при выборе параметров всегда важно учитывать стоимость.

Производительность сети

Хотя размеры процессора и памяти являются наиболее важными составляющими при выборе типа машины, изучите также доступные ограничения производительности сети, чтобы убедиться, что вы не «морите голодом» свои приложения. Например, если у вас есть пакетный процесс, который будет считывать большое количество данных, вы можете обнаружить, что исчерпали пропускную способность на машине меньшего типа.



Стоит отметить, что право выхода из сети между облачными зонами и регионами обычно сопряжено с определенными затратами. Это может стать неожиданностью при настройке реплик, но мы по-прежнему считаем, что для обеспечения резервирования важно размещать реплики в отдельных зонах.

Выбор правильного типа диска

Хотя типы машин, как правило, являются динамическими, выбор, который вы сделаете для хранилища данных, вероятно, будет самым сложным решением. После того как выбран тип диска и вы начали использовать его для хранения данных, перейти на другой тип диска становится затруднительно. Как правило, необходимо смонтировать второй диск и скопировать на него данные. Все исправить можно, но, безусловно, это более сложная задача, чем просто быстрая перезагрузка для добавления процессоров.

Выбор правильного типа диска в значительной степени зависит и от предположительной рабочей нагрузки. Рабочие нагрузки, требующие интенсивного чтения, выиграют от увеличения объема памяти по сравнению с дисковой производительностью, поскольку доступ к памяти на несколько порядков быстрее. Если рабочий набор больше, чем буферный пул InnoDB, вы всегда будете в итоге обращаться к диску для чтения некоторых данных. Рабочие нагрузки, требующие интенсивной записи, всегда будут обращаться к диску, и именно здесь большинство пользователей начнут замечать первые узкие места на диске.

Типы подключений дисков

Первое решение, которое необходимо принять, заключается в том, используете ли вы диски с локальным или с сетевым подключением. Преимущество локально подключенных дисков заключается в невероятно высокой производительности и стабильной пропускной способности, но они уязвимы к потере данных. Это связано с тем, что они рассматриваются как диски только для эфемерных данных. Если аппаратное обеспечение, на котором работает виртуальная машина с локально добавленными данными, сломается, вы можете потерять все данные, размещенные на локальном диске. А при некоторых обстоятельствах

даже выключение экземпляра может означать, что при повторном запуске вы окажетесь на другом хост-компьютере с пустыми дисками. Локально подключенные диски обычно не имеют репликации или RAID-массива. Отказ диска на уровне хоста может привести к потере данных. Если вы пойдете по этому пути, мы настоятельно рекомендуем рассмотреть возможность использования программного RAID, чтобы хотя бы минимизировать вероятность потери данных. Более подробную информацию смотрите в обсуждении RAID в главе 4.

В отличие от локально подключенных сетевые диски идут другим путем, отдавая предпочтение избыточности и надежности вместо производительности. Это не означает, что производительность сетевых дисков плохая — просто они не такие производительные, как локальные. На подключенном к сети диске могут возникать зависания, а на локально подключенном — нет. Кроме того, обычно локально можно достичь гораздо более высоких показателей пропускной способности и количества операций ввода/вывода в секунду.

Облачные провайдеры предоставляют удобные инструменты резервного копирования или моментальных снимков при использовании дисков, подключенных к сети. Они отлично подходят для применения MySQL, при условии, что у вас настроены параметры, совместимые с ACID¹, и решение для резервного копирования спроектировано правильно. Вы можете сделать снимок диска в любой момент и восстановить его с помощью обычного аварийного восстановления после сбоя без каких-либо проблем.

Можно также использовать моментальный снимок диска для создания чрезвычайно быстрых реплик даже на дисках размером во много терабайт. Делая это, вы сводите к минимуму задержку репликации, которую необходимо наверстать, прежде чем с репликой можно будет работать.

Обратите внимание: если вы используете локально подключенный диск вместо подключенного к сети, вам нужно будет решить, как самостоятельно создавать резервные копии данных с помощью LVM или стороннего инструмента, например XtraBackup. Более подробно резервное копирование обсуждалось в главе 10.

Последнее замечание о типах подключений заключается в том, что облачные провайдеры не предлагают что-то вроде кэша записи (с питанием от батареи или флеш-памяти), как можно увидеть в аппаратных RAID-картах.

SSD в сравнении с HDD

По большому счету, желательно использовать твердотельный накопитель для всего, особенно для тома данных MySQL. Если ваш бюджет существенно ограничен, можете применять жесткие диски в качестве более дешевого варианта

¹ Напоминаем, что это `innodb_flush_log_at_trx_commit=1` и `sync_binlog=1`.

загрузочного диска. В ходе экспериментов мы обнаружили, что твердотельный накопитель загружается в два-три раза быстрее, чем жесткий диск. Если время загрузки имеет значение, особенно в случае сбоя или перезагрузки, всегда используйте твердотельный накопитель.

Операции ввода/вывода и пропускная способность

Еще одним сложным фактором является определение требований к операциям ввода/вывода и пропускной способности. Вы должны хорошо понимать, как каждое из этих требований выглядело в прошлом и как станет выглядеть в будущем, прежде чем выбрать необходимый диск.

Если выполняете миграцию с существующей рабочей нагрузкой, то в идеале у вас уже есть показатели использования диска для нее, которые позволят выбрать диск наилучшим образом. Если нет, можете задействовать `pt-diskstats` из пакета `Percona Toolkit` для сбора показателей за день для измерения пиков.

Для новых баз данных потратьте немного времени, чтобы понять, насколько интенсивным будет приложение. Может помочь даже такая элементарная вещь, как понимание того, каким будет соотношение чтения и записи. Если все остальное не помогает, найдите золотую середину между производительностью и стоимостью и определите ожидания, которые, возможно, позже придется корректировать.

Дополнительные советы

Если вы решите запустить собственную MySQL на виртуальной машине, вам придется отвечать за гораздо большее, чем управляемый сервис. Нужно будет самостоятельно выполнять такие действия, как определение размера диска, резервное копирование и т. д. Вот несколько советов, которые следует учесть, если вы пойдете по этому пути.

Работа с перезагрузками хоста

Ваша виртуальная машина на самом деле работает на чужом оборудовании. Как бы нам это не нравилось, оборудование может выйти из строя, и, когда такое случается, ВМ немедленно завершает работу. Если сделана настройка, виртуальная машина начнет загрузку на другом узле. Если это происходит во время обслуживания производственного трафика, особенно на узле-источнике, принимающем записи, в работе ваших пользователей может случиться сбой.

Не существует волшебных решений, которые позволят избежать таких проблем, — с этим придется смириться. В данном случае у вас, как правило, есть

два варианта: инициировать процесс аварийного переключения на реплику (рассматривается в разделе «Отказоустойчивость репликации» главы 9) или подождать, пока источник не вернется в оперативный режим. Решение проблемы незапланированного повышения реплики может быть очень сложным. Мы советуем просто позволить серверу вернуться в оперативный режим, а репликации снова подключиться к нему естественным образом.

Вы можете облегчить этот процесс, придерживаясь следующих рекомендаций.

- Используйте твердотельный накопитель в качестве загрузочного диска, чтобы перезагрузка происходила как можно быстрее. Часто системы возвращаются в оперативный режим менее чем за 5 мин.
- Подавите все имеющиеся уведомления об отключении хоста на срок до 5 мин, чтобы дать системе возможность полностью перезагрузиться и восстановить работоспособность.
- Вы можете запрограммировать параметр для динамического отключения флага `read_only`, если сервер-источник был перезагружен, что позволит продолжить запись без вмешательства человека. Это хорошо работает в сочетании с опцией `crond@reboot`, которая запускает скрипт при старте системы. Единственная оговорка заключается в том, что вам нужно иметь возможность запросить систему, чтобы определить, должна ли она принимать записи.
- Максимизируйте коммуникацию, автоматически отправляя электронные письма или сообщения в чате командам или каналам, которым, возможно, необходимо знать о сбое. Вероятно, достаточно отправить сообщение: «Узел *<полное доменное имя узла>* неожиданно отключился и должен вернуться в оперативный режим через 5 минут», чтобы люди перестали писать вам сообщения или даже вызывать вас по пейджеру.

Разделяйте данные операционной системы и MySQL

Независимо от того, какой вариант подключения дисков вы выберете — локальное подключение или сетевое, рекомендуем хранить данные операционной системы и MySQL отдельно по следующим причинам.

- Моментальные снимки диска будут ограничены только данными MySQL и не будут содержать никакой информации об операционной системе.
- Если диск подключается к сети, можно легко отсоединить его и снова подключить к другой машине.
- Кроме того, на дисках, подключенных к сети, можно обновить или заменить операционную систему, для чего не требуется повторно копировать данные в файловую систему.

Следует также подумать о том, куда поместить определенные файлы, например файл идентификатора процесса MySQL, любые файлы журналов и файл сокетов. Рекомендуем хранить их в операционной системе, хотя журналы могут остаться на диске с данными.

Резервное копирование двоичных журналов

Отправляйте свои двоичные журналы в корзину. Установите контроль жизненного цикла корзины, чтобы автоматически удалять старые файлы по истечении определенного времени. Запретите удаление файлов до определенного времени или удаление в целом.

Не забывайте о безопасности. Если оставить корзину открытой для чтения всему миру, это может стать кошмаром, который только и ждет своего часа. Контроль над тем, кто может читать или удалять эти данные, очень важен для поддержания стратегии безопасного резервного копирования. Рассмотрите возможность разрешить всем машинам базы данных записывать, но ни одной из них не разрешать читать или удалять. Контролируйте чтение и удаление отдельно с ограниченной учетной записи, машины или и того и другого.

Автоматическое расширение дисков

При задействовании дисков, подключенных к сети, вы платите за объем выделенного, а не используемого пространства. Это может означать, что расточительно оставлять большое количество зарезервированного, но неиспользуемого пространства на дисках с данными MySQL. Один из способов оптимизации — запланировать гораздо более высокий процент использования дискового пространства, например 90 %, но как уменьшить риск нехватки дискового пространства?

Поставщики облачных услуг обычно предоставляют доступ к API-интерфейсу для увеличения размера вашего диска. С помощью небольшого количества кода вы можете определить, превышают ли ваши серверы отметку заполнения диска 90 %, и вызвать API для расширения дисков. Это может снизить вероятность выгрузки страниц для сервера, на котором почти закончилось дисковое пространство. В целом этот процесс может существенно повлиять на то, сколько вы тратите на предоставляемое дисковое пространство. Однако сделаем несколько предостережений по этому поводу.

- Подумайте о том, как часто вы должны запускать код, который определяет процент использованного дискового пространства. Вам нужно выяснить, основываясь на пропускной способности диска, сколько времени потребуется процессу, чтобы полностью заполнить оставшийся диск. Код должен выполняться чаще.

- Если процесс запустится и продолжит неограниченно расширять диск, вы можете получить том объемом 64 Тбайт. Это может стать дорогостоящим сюрпризом, когда придет время оплачивать счет от провайдера.
- Вызов API-интерфейса расширения диска может привести к кратковременной остановке диска. Обязательно протестируйте под нагрузкой код вызова этого API, чтобы убедиться, что он не окажет негативного влияния на пользователей.

Резюме

Если вы работаете с одной из тысяч компаний, предоставляющих услуги публичного облака, то у вас есть множество вариантов запуска баз данных. Как инженера по базам данных, вас будут спрашивать, какое управляемое решение использовать, стоит ли вообще применять управляемые решения для реляционных баз данных и каковы компромиссы для каждого варианта. Внося свой вклад в эти обсуждения, самое важное, что нужно иметь в виду, — то, что бесплатных обедов не бывает. Каждый из вариантов сопровождается определенными компромиссами. Самое полезное, что вы можете сделать, — это сформулировать компромиссы в контексте того, как работает бизнес и на какой стадии зрелости он находится, чтобы помочь своей организации выбрать наиболее подходящий вариант. Мы надеемся, что эта глава помогла вам подготовиться к таким разговорам и быть способными сопоставить имеющиеся компромиссы с потребностями компании.

Соответствие MySQL нормативным требованиям

Роль команд разработчиков баз данных представляет интерес для многих заинтересованных лиц из сферы бизнеса. Как мы уже говорили, вам приходится планировать не только производительность и время безотказной работы, но и затраты на инфраструктуру, аварийное восстановление и всевозможные требования к соответствию нормативам.

Ваша работа не ограничивается управлением этими данными в ходе деятельности предприятия. Вы также должны помочь ему защитить данные и сертифицировать их на соответствие нормативным требованиям, которые либо диктуются законом, либо критически важны для бизнеса. Вы должны понимать практические задачи для удовлетворения этих требований и включить их во все проекты архитектуры данных, включая способы автоматизации операционных задач, управления доступом и преобразования административных задач в код для их автоматизации.

В этой главе рассматриваются разные типы сертификатов соответствия, которые может получить компания, и различные проблемы, связанные с базами данных, которые могут возникнуть. Мы попробуем объяснить, как разрабатывать системы с учетом различных требований соответствия, и обсудим, как ведение журнала доступа может стать важной частью выполнения требований соответствия. Наконец, расскажем о суверенитете данных как о новой проблеме для архитектуры данных во всех типах компаний.



Эта глава не ставит цель дать вам юридическую консультацию. Мы стремимся помочь вам управлять требованиями соответствия в ходе работы с большим количеством баз данных и обеспечить соответствие требованиям на ранних стадиях разработки. В поисках совета о том, как правильно выполнять конкретные меры контроля, вам всегда следует консультироваться с юридическим отделом компании.

Что такое соответствие

Система управления, управление рисками и соответствие (GRC) — это принципы, процессы и законы, которые определяют, как компания оценивает и назначает приоритеты рисков для своих активов и как соблюдает законы, регулирующие обработку и передачу персональных или медицинских данных, которые она может использовать для работы своего продукта. Стартапы на ранних стадиях развития, как правило, не предъявляют особых требований к соблюдению норм, поскольку они считают свой продукт подходящим для рынка. Однако по мере роста бизнеса вы начинаете применять целый ряд нормативных актов. Некоторые нормативные акты должны распространяться на все данные компании, а некоторые — на определенные их части.

Регулярно используемым термином в контексте соблюдения требований является «*контроль*». Средства контроля — это процессы и правила, которые компания определяет и применяет на практике для снижения вероятности нежелательного исхода риска.

Познакомимся с некоторыми нормативными документами, о которых вы должны знать. А после рассмотрим архитектурные изменения, которые могут помочь сделать выполнение этих требований более управляемым.

Элементы контроля сервисной организацией типа 2

Элементы контроля сервисной организацией типа 2 (Service Organization Controls Type 2, SOC 2, <https://oreil.ly/PwWBg>) — это набор средств контроля соответствия, которые сервисные организации могут использовать для составления отчетности о своей практике, связанной с безопасностью, доступностью, целостностью обработки, конфиденциальностью и неприкосновенностью частной жизни. Инженеры по базам данных в организациях, желающих получить сертификат SOC 2, должны иметь хорошо отлаженную практику управления изменениями баз данных, процедур резервного копирования и восстановления, а также управления доступом к экземплярам баз данных.

Закон Сарбейнса — Оксли

Закон Сарбейнса — Оксли (Sarbanes — Oxley Act, SOX, <https://oreil.ly/qHAN0>) от 2002 года — это закон, который должны соблюдать все компании, которые становятся публичными. Он предназначен для защиты инвесторов путем повышения точности и надежности корпоративной информации, раскрываемой в соответствии с законами о ценных бумагах и для других целей. Для инженерной организации обязанности SOX требуют доказательства того, что доступ к базам

данных, содержащим сведения, влияющие на доходы, имеют только те, у кого есть в этом необходимость, и что любые изменения этих данных регистрируются в журнале, а сами изменения вносятся по документально подтвержденным причинам.

Если вы являетесь публичной компанией, 404 SOX — это обязательный по закону элемент контроля, который вы должны знать и выполнять. Его цель — предъявлять доказательства того, что финансовые показатели, представленные компанией, подкреплены методами доступа к данным и управления изменениями, которые точно соотносят оказанные услуги с полученным доходом и обеспечивают контроль любых изменений в таких данных.

Стандарт безопасности данных индустрии платежных карт

Стандарт безопасности данных индустрии платежных карт (Payment Card Industry Data Security Standard, PCI DSS, <https://oreil.ly/V6GBh>) — это стандарт, обязательный для всех финансовых учреждений, обрабатывающих данные кредитных карт. Его цель состоит в том, чтобы защитить данные владельцев кредитных карт от утечки и использования в мошеннических транзакциях.

Важным аспектом контроля PCI-DSS, когда речь заходит о вашей работе в качестве инженера баз данных, является управление доступом к данным о держателях карт. Это означает, что вам необходимо будет учитывать этот элемент контроля в своей архитектуре, чтобы убедиться, что данными карт управляют отдельно. Мы обсудим, как вы можете достичь этого, позже в данной главе, когда будем рассматривать разделение ролей.

Закон о переносимости и подотчетности медицинского страхования

Закон о переносимости и подотчетности медицинского страхования (The Health Insurance Portability and Accountability Act, HIPAA, <https://oreil.ly/fKeQd>) от 1996 года — это нормативный акт США, разработанный для защиты конфиденциальности данных, связанных со здоровьем отдельных лиц, при их сборе и обработке поставщиками медицинских услуг, планами медицинского страхования или их деловыми партнерами. Этот закон применяется к данным, определяемым как электронная личная медицинская информация (ePHI). Организациям, предлагающим продукты, которые требуют соответствия требованиям HIPAA, необходимо, чтобы их инженеры по базам данных внедрили такие средства контроля, как контроль доступа к ePHI, шифрование всех ePHI и ведение журнала действий при каждом доступе к ePHI.

Федеральная программа управления рисками и авторизацией

Для компаний, работающих в США и желающих вести бизнес с американскими государственными структурами, Федеральная программа управления рисками и авторизацией (Federal Risk and Authorization Management Program, FedRAMP, <https://oreil.ly/ZVQqq>) представляет собой сертификацию, предлагаемую федеральным правительством для квалификации компаний в качестве поставщиков облачных услуг для федеральных структур. Она представляет собой набор стандартов, необходимых для того, чтобы иметь право принимать федеральные организации в качестве клиентов. Эти стандарты включают управление конфигурацией, контроль доступа, оценку безопасности и аудит доступа и изменений данных.

Общий регламент по защите данных

Общий регламент по защите данных (General Data Protection Regulation, GDPR) — это регламент Европейского союза, введенный в 2016 году для регулирования порядка хранения персонально идентифицируемой информации о лицах из ЕС и управления ею организациями, которые выступают в качестве обработчиков данных, независимо от их местонахождения. В нем были представлены первые шаги по управлению конфиденциальностью данных, такие как требование согласия перед сбором личных данных, установление ограничений на доступ к этим личным данным в рамках всей организации, занимающейся их обработкой, а также предоставление физическим лицам юридического права требовать удаления их данных из систем любого обработчика данных, который, возможно, мог собрать их в результате онлайн-активности этих лиц. Это известно как право человека на забвение.

Schrems II

В 2020 году суд ЕС вынес решение по судебному делу между ЕС и ирландским подразделением Facebook. Постановление, известное как Schrems II, оказало огромное влияние на все американские компании, которые собирают данные о лицах из ЕС.

Privacy Shield — это юридическая норма, на основании которой американские компании работали в ЕС в течение многих лет. Постановление Schrems II объявило ее недостаточной для защиты конфиденциальности лиц из ЕС, когда их данные собирают подразделения американских компаний, расположенных в Евросоюзе. В основе аннулирования Privacy Shield лежит решение Суда ЕС

о том, что эта норма не является достаточной гарантией того, что за лицами из ЕС не будет следить правительство США законными средствами (с использованием механизма, предусмотренного законом о надзоре за иностранными разведками 1978 года — FISA), и, следовательно, персональные данные о лицах из ЕС, собранные юридическими лицами США, должны оставаться в ЕС и не попадать в активы США и не быть доступными лицам из США.

Это постановление усложняет анализ архитектуры данных по сравнению с первоначальной версией GDPR. Из-за того что оно было принято недавно, данные по его исполнению неизвестны. В этой ситуации каждая компания должна сама определить, какой объем данных она собирает и обрабатывает и каким образом. Можно с уверенностью предположить, что Schrems II будет применяться для приложений и инфраструктуры передачи данных, которые вы используете, если у вас есть клиенты в Европе или будут в дальнейшем.

Выстраивание контроля за соблюдением нормативных требований

Как видите, мир соблюдения нормативных требований для предприятий и, как следствие, контроля данных, которые они используют для работы, огромен, и средства контроля могут быть многочисленными в зависимости от цели каждого закона или сертификации, необходимой вашему бизнесу. Хорошей новостью является то, что одна и та же работа может охватывать более одного элемента контроля и тем самым повысить эффективность и последовательность действий при управлении инфраструктурой. И вам необходимо понимать, какие из этих средств контроля требуются для бизнеса и с какими целями. Как только ваша компания вырастет до таких размеров, когда ей потребуются начать внедрение какого-либо подмножества регулятивных механизмов контроля, вы станете тем человеком, которому придется представлять доказательства соответствия различным аудиторам. Понимание того, на что направлен каждый элемент контроля, будет иметь большое значение для представления правильных доказательств, облегчающих проведение проверок.



Обеспечение соответствия требованиям — это непрерывный процесс, который не может быть легко добавлен при необходимости. Многие рекомендации по архитектуре, представленные в этой главе (разделение ролей, отслеживание изменений и т. д.), относятся к тому, о чем вам следует подумать и чье применение отстаивать, как только ваша компания преодолеет стадию «все еще соответствует рынку». Именно такие практики приведут ваш бизнес к успеху к тому времени, когда соблюдение требований станет реальной необходимостью.

Управление секретами

Прежде чем обсуждать, как управлять секретами, определимся, что в вашей инфраструктуре может подпадать под это определение. Это:

- пароли для приложений, взаимодействующих с базами данных;
- пароли для сотрудников службы поддержки либо операторов, управляющих экземплярами баз данных;
- API-токены, позволяющие получать доступ к данным или изменять их;
- закрытые ключи SSH;
- ключи сертификатов.

Основная компетенция, которая необходима в вашей организации для упрощения многих элементов управления безопасностью, — это механизм безопасного управления секретами отдельно от управления конфигурацией. Вам нужен способ доставки и оборота конфиденциальных данных, таких как учетные данные доступа к базам данных, независимо от того, используются ли они приложениями или командами, для целей отчетности.

Если вы запускаете свои приложения и базы данных в облачной среде, мы настоятельно рекомендуем выяснить, какое решение для управления секретами предпочтительно для данного провайдера облачных услуг, прежде чем создавать собственное. Вам нужно что-то, что обеспечивает как минимум уровень шифрования, приемлемый для стандартов Национального института стандартов и технологий (NIST)¹, поскольку этого требует ряд нормативных актов, включая HIPAA и FedRAMP.

Если у вашего облачного провайдера нет приемлемого решения для управления секретами, вам, вероятно, придется создать собственное. Возможно, это новое начинание для вашей организации, и оно потребует более масштабных усилий, чем сказано в данной книге.

Независимо от того, используете ли вы управляемое решение или в итоге станете разрабатывать собственное, помните о том, что решение для управления секретами усложняет вашу архитектуру. Цель этого решения — управлять секретами, а не быть единой точкой отказа для вашего продукта. Вам потребуется точное описание принятых компромиссов, объясняющее, что произойдет, когда решение для управления секретами будет внедрено. Предварительный

¹ Чтобы получить более подробную информацию об этих стандартах, поговорите с вашей командой по информационной безопасности или изучите карманное руководство NIST Cybersecurity Framework (<https://oreil.ly/s7XOJ>) от O'Reilly.

разговор с юридическим отделом и отделом безопасности о том, какие секреты и в каком виде будут кэшироваться, поможет избежать в дальнейшем несоответствия ожиданиям.

Часто решения, принятые компанией на ранних этапах развития и в то время признанные удобными, следует пересмотреть задолго до планирования контроля соответствия. Вы должны быть готовы объяснить своему руководству, почему эта работа важна для улучшения системы безопасности и снижения рисков.

Не предоставляйте общий доступ пользователям

Не применяйте общие учетные данные базы данных для разных служб. Это решение окупается многократно, если у вас произошла случайная утечка данных из базы и вам теперь придется оценивать «размер бедствия» — сколько частей стека приложений и процессов должны получить новые учетные данные базы данных. Как инженерам по базам данных, нам часто приходилось работать в стартапах, где люди пользуются удобным на первый взгляд способом «весь код использует одни и те же учетные данные для доступа к базе данных». Поверьте, это очень дорогой способ, и вы скажете себе спасибо в будущем, если дадите каждому пользователю базы данных доступ только к тем услугам, которые ему необходимы.

Теперь, когда мы рассмотрели этот базовый, но крайне важный фундаментальный принцип, поговорим о том, что нужно учитывать при выборе решения для хранения учетных данных базы данных или секретов в целом.

Не храните учетные данные производственных баз данных в репозиториях кода

Это может показаться очевидным, но так происходит постоянно, как видно из большого количества отчетов об инцидентах, связанных с безопасностью, в больших и малых компаниях. Важно относиться к этому настороженно и не предполагать, что такая ошибка в вашей организации маловероятна. Применение принципа «Доверяй, но проверяй» сыграет важную роль в предотвращении проблем в будущем. Сканирование репозитория кода на наличие потенциальных секретных строк до того, как запрос на слияние изменений с кодом в репозитории будет выполнен, — общепринятая практика (и это то, что могут сделать для вас размещенные сервисы репозитория, такие как GitHub). Если в вашей организации еще не думали об этом, возможно, вам придется стать инициатором и защитником этой необходимости. Помните, что соответствие требованиям

и безопасность необходимы для организации в целом, и хотя не все может или должно делаться командой базы данных, вы являетесь заинтересованным лицом в том, как инженерная служба в целом обсуждает эти вопросы.

Эти методы имеют основополагающее значение для того, чтобы изначально организовать правильный подход к соблюдению требований и обеспечению безопасности. Они сделают ряд операций управления секретами, которые мы собираемся рассмотреть, еще более простыми и еще значительно снизят риск для бизнеса, если возникнет необходимость в срочных изменениях.

Поговорим о компромиссах при выборе решения для управления секретами.

Выбор решения для управления секретами

Выбор решения для управления секретами будет зависеть от того, в какой среде вы работаете и что может быть наиболее легко интегрировано не только с базами данных, но и со всем стеком приложений. Всегда придется находить компромиссы между удобством и удовлетворением всех ваших потребностей. Поэтому вам необходимо четко разъяснить всем заинтересованным сторонам (некоторые из них не занимаются инженерным делом), в чем заключаются ограничения и каковы компромиссы для обеспечения доступности или отказоустойчивости. Некоторые из компромиссов, которые следует учитывать при проверке того, что может предложить облачная (или частная) инфраструктура, включают следующее.

- *Ограничения пространства.* Некоторые облачные решения для управления секретами делают предположения о длине секрета, что может привести к неожиданностям, если вы хотите хранить что-то более длинное, чем пара из имени пользователя базы данных и пароля. Если ваши элементы управления соответствуют требованиям, требуют рассматривать более длинные текстовые строки, такие как секреты (например, ключи SSH или частные сертификаты SSL), следует проверить максимальный размер, в котором можно сохранить данный ключ. Одна из «мин», на которые натываются некоторые организации, заключается в том, что по мере роста объема секретов требуется новое, отличающееся от прежнего решение для управления секретами, чтобы разместить более длинные секреты. Теперь им приходится заниматься либо миграцией, что может повлиять на время безотказной работы, либо управлением инструментарием и интеграцией с двумя отдельными решениями для управления секретами, что сопряжено с определенными сложностями.
- *Ротация секретов.* Если вы работаете в публичном облаке и можете использовать его решение для управления секретами, то для вас есть хорошие

новости: все три основных облачных решения на момент написания этой книги предлагают некоторый метод автоматизации ротации секретов наряду с управлением версиями, чтобы сделать переход к новым секретам плавным для ваших служб. Однако если выбранное вами решение для управления секретами не поддерживает их ротацию, то необходимо спланировать, как вы будете это делать и при запланированном изменении (например, у вас может быть элемент управления, который требует, чтобы учетные данные базы данных ротировались с определенной периодичностью), и при незапланированном экстренном изменении (например, кто-то случайно сохранил пароль базы данных в публичном хранилище). Как это сделать без ущерба для работающих приложений? Это в значительной степени зависит от того, как вы доставляете изменения конфигурации в работающие приложения и как работает конвейер развертывания. Вот общая суть того, как это организовать. Это изменение представляет собой развертывание. Даже если ваши приложения обращаются к учетным данным базы данных в виде строки конфигурации, все равно необходимо распространить это изменение конфигурации на весь парк приложений, а также, как правило, организовать перезапуск, не влияя на доступность сервиса в целом.

- *Региональная доступность.* Учтите, что ваше решение по управлению секретами предназначено не только для хранения секретов, но и для их доставки. Если вы хотите избежать известной плохой практики, такой как хранение секретов в коде, то нужно, чтобы приложение могло извлекать секреты, необходимые для обработки запросов, во время выполнения. Это означает, что теперь нужно думать о том, как эти секреты будут извлекаться, что произойдет, если приложение не сможет получить доступ к службе управления секретами, и какие режимы отказа вводит эта новая зависимость. Если вы отвечаете за приложения, которые должны работать во многих географических регионах, региональные возможности вашего решения для управления секретами становятся еще одним вопросом, который необходимо учитывать. Обеспечивает ли решение вашего облачного провайдера автоматическую репликацию секретов в другие регионы, или вам придется создавать такую возможность?

Разделение ролей и данных

Важной целью перечисленных ранее нормативных актов является разделение данных в зависимости от риска, который их утечка несет для предприятия или его клиентов. Это концепция наименьших привилегий как для доступа людей, так и для доступа кода приложений. Такое разделение позволяет обеспечить более адекватный контроль и отслеживание изменений в зависимости от того, что представляют собой данные и какой риск с ними связан.

Разделение по основаниям соответствия нормативным требованиям

Одной из причин, которая может заставить разделить наборы данных на отдельные кластеры, является наличие различных требований к соответствию нормативным требованиям с абсолютно разными элементами контроля. Допустим, вы поставщик услуг маркетинговых коммуникаций, создающий новый продукт с упором на технологии здравоохранения. Данные, которые используются в настоящее время, не были связаны со здравоохранением и не попадали под ряд юридических требований. Как только компания переходит в область медицинских технологий, у нее появляется подмножество клиентов и их данных, по отношению к которым она несет юридическое бремя обработки личной медицинской информации (РНИ). В этом случае имеет смысл с самого начала разрабатывать новый продукт в выделенном хранилище данных, чтобы можно было более адекватно применять средства контроля соответствия требованиям HIPAA, не возлагая чрезмерную нагрузку на существующий набор данных и зависимые от него приложения.

Разделение пользователей базы данных

По мере того как все более сложным становится ваш продукт, а вслед за ним и технологический стек, который его поддерживает, у вас появится более одного приложения с доступом к данным. Очень важно как можно раньше внедрить в организации хорошо контролируемый доступ к данным, не используя одни и те же учетные данные доступа к базе данных в нескольких кодовых базах. Инциденты безопасности и случайные утечки учетных данных — это события, которые происходят в любых компаниях независимо от их размера. И когда такое случится, будет крайне важно, чтобы утечка секретов сказалась лишь на ограниченной изолированной области бизнес-операций, что позволит вам быстро изменить эти секреты.

Отслеживание изменений

Ряд нормативно-правовых актов, регулирующих соблюдение нормативных требований, содержит элементы контроля за отслеживанием изменений в подмножествах данных, которые влияют на финансовую отчетность, в системах, генерирующих счета-фактуры, а также за тем, как эти изменения рассматриваются и тестируются. Одним из первых очевидных мест, где этот вид контроля соответствия становится уместным, является база данных. Поскольку вы работаете в компании с расширяющимися обязанностями по обеспечению соответствия, такие процессы, как просмотр, применение и отслеживание изменения схемы в производственной среде, требуют большей строгости и планирования, чем когда инженер просто регистрируется в производственной среде

на узле-источнике и вносит изменения. Если вы подготовитесь и спланируете проведение аудитов вместе с командой внутреннего аудита или командой по соблюдению нормативных требований, это позволит сократить ваши накладные расходы.

Цель этих действий заключается в том, чтобы избежать превращения ежегодных аудиторских проверок в серьезное разрушительное событие, когда команда в спешном порядке собирает доказательства для аудиторской группы. Внося некоторые изменения в то, как протекают обычные бизнес-операции, вы сможете получить «встроенные» доказательства, которые гораздо проще собирать и использовать для аудита. В этом разделе вы увидите повторяющуюся тему создания структурированных журналов регистрации всех операций. Так сделано намеренно. Это имеет большое значение для обеспечения одинакового отслеживания всевозможных изменений, и такая согласованность помогает предприятию достичь целей аудита с гораздо меньшими сбоем. Вы можете видеть, как все это складывается воедино, на рис. 13.1.

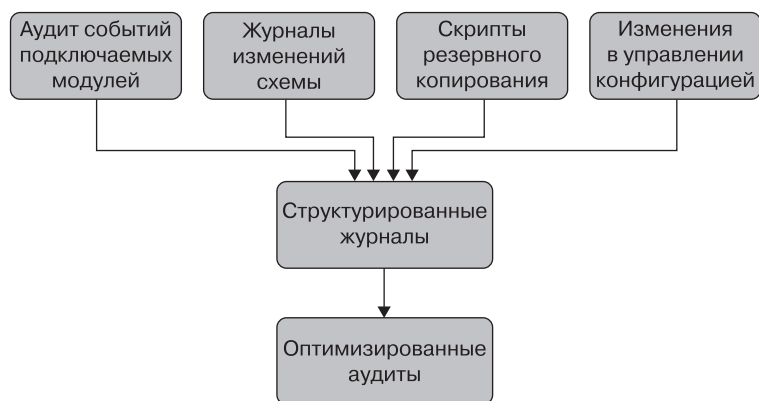


Рис. 13.1. Примеры различных операционных задач, отправляющих структурированные журналы в одно место для упрощения аудита

Рассмотрим различные типы изменений в системах баз данных и то, как автоматизировать отслеживание соответствия для них.

Ведение журнала доступа к данным

Многие средства контроля соответствия требуют, чтобы вы вели журналы изменений или доступа к определенным наборам данных. Это может быть необходимо для отслеживания изменений в финансовых данных или для таких нормативных актов, как PCI или HIPAA, где данные являются достаточно

конфиденциальными для того, чтобы необходимо было отслеживать любой доступ к ним.

Вы можете удовлетворить это требование непосредственно на уровне базы данных, используя либо плагин журнала аудита *Percona*, если применяете сборку (fork) *Percona*, либо эквивалентный плагин *MySQL Enterprise Audit*, если действуете *MySQL Community Server*. Преимущество здесь заключается в том, что теперь можно отслеживать изменения на последнем этапе перед изменением данных, особенно если вы работаете в среде, где изменения в базу данных можно вносить несколькими способами.

Нежелательные варианты отслеживания изменений. Вы можете спросить: «Почему бы не использовать триггеры для отслеживания любых изменений в нужных мне таблицах?» Такой подход мы определенно встречали в прошлом, но применять его не рекомендуется. Вот несколько причин.

- Известно, что триггеры снижают производительность записи, что может сказаться в самый неподходящий момент.
- Триггеры равносильны хранению бизнес-логики в базе данных, что не рекомендуется.
- Хранение кода в базе данных, скорее всего, позволит обойти любой процесс тестирования, промежуточной подготовки и развертывания этого кода. Триггеры легко могут стать сюрпризом для вашей команды во время инцидентов.
- Триггеры могут поддерживать только отслеживание действий записи. Это решение не может быть расширено для отслеживания доступа для чтения, если возникнет такая необходимость.

Посмотрим, как можно использовать подключаемый модуль журнала аудита *Percona* и как его настроить.

Установка и настройка журналов аудита *Percona*. Плагин журнала аудита *Percona* поставляется как часть сборки (fork) *MySQL* от *Percona*, но по умолчанию он не установлен и не включен. Вы можете установить его, выполнив в своих экземплярах *MySQL* следующую команду как часть процесса начальной загрузки любого нового экземпляра:

```
INSTALL PLUGIN audit_log SONAME 'audit_log.so';  
SHOW PLUGINS;
```

Вторая команда выводит список запущенных плагинов и должна подтвердить, что плагин журнала аудита действительно запущен как часть серверного процесса. Помимо его включения, вам также необходимо определить, как будут использоваться его результаты. Именно здесь происходит настоящее планирование.

Подключаемый модуль журнала аудита Персона позволяет определить, какие глаголы операторов вам нужно отслеживать. Это обеспечивает гибкость при выполнении различных элементов контроля, не создавая в журналах аудита большого шума, который не имеет отношения к тому, что вас интересует. Обязательно ознакомьтесь с документацией, чтобы при необходимости правильно настроить эту переменную конфигурации.

Одним из несомненных достоинств плагина является то, что вы можете установить его, но не отслеживать запросы¹. Это может быть полезно, если вы все еще работаете над тем, как применять его выходные данные, и вам нужно выключать и включать его так, чтобы это не влияло на время безотказной работы. Но вместе с гибкостью приходит и сложность. Помимо управления переменными конфигурации, которые поставляются с подключаемым модулем журнала аудита, вам необходимо постоянно контролировать его работу. Если это критически важная функция для бизнеса, значит, ее стоит контролировать. Поскольку плагин можно отключить на лету без перезагрузки сервера, вам нужно нечто большее, чем просто проверка файла `my.cnf` на диске, чтобы убедиться: он делает то, что должен делать. Лучше всего использовать запросы оболочки для анализа текущего состояния плагина и подтверждения того, что он действительно отслеживает запросы. Вот два примера однострочных запросов для проверки каждого из них:

```
# Single liner to check that the audit log plugin is active
$ mysql -e "show plugins" | grep -w audit_log | grep -iw active
```

```
# Single liner to check that the plugin policy is actually monitoring queries
$ mysqladmin variables | grep -w audit_log_policy | grep -iw queries
```

В этих примерах предполагается, что вы хотите отслеживать только запросы. Нужно будет отредактировать проверку, если вы планируете применять плагин также для отслеживания входов в систему.

Получение и использование журналов плагина аудита. Как видите, плагин журнала аудита очень гибок, но он создает только события аудита. Вы сами должны определить наилучший способ получения этих журналов, размещения их в таком месте, где их можно легко найти и анализировать, а также обнаруживать в них аномалии, не создавая при этом большой нагрузки. Плагин может просто сбрасывать выходные данные в локальные файлы, но это может увеличить риск возникновения сбоев в работе из-за заполнения этими журналами дисков узла базы данных.

¹ Подробнее об этом читайте в документации к подключаемому модулю: <https://oreil.ly/gwqc2>.

Более сложным вариантом является использование возможности плагина отправлять свои выходные данные в `rsyslog`, обычную утилиту управления журналами Unix, и оттуда задействовать `rsyslog` для пересылки всех этих событий на выбранную вами платформу структурированного ведения журнала. Этот вариант привлекателен, поскольку он переносит данные в то же место, где ваша организация уже использует структурированное хранилище журналов, что уменьшает препятствия для просмотра, поиска и анализа этих событий заинтересованными сторонами, не входящими в команду базы данных. Однако имейте в виду, что применение `rsyslog` для пересылки журналов таким образом потребует от вас ознакомления с тем, как это работает. Убедитесь, что вы решили, каким образом будет настроен `rsyslog` для этого потока данных¹, и задокументировали это. Вполне возможно, что в `rsyslog` существует ряд конфигураций по умолчанию, которые не способствуют достижению желаемого результата, и именно вы должны проявить должную осмотрительность, чтобы найти их и изменить соответствующим образом.



Обязательно задокументируйте, как выходные данные плагина журнала аудита хранятся, пусть даже временно, на узлах базы данных. Если способ доставки этих файлов замедляется, влияние буферизации событий в подключаемом модуле может сказаться на производительности самого сервера базы данных. Это состояние сбоя трудно отладить, потому что его единственным симптомом является замедление выполнения запросов. Планируйте тестирование хаоса для всей цепочки этих журналов с учетом отказоустойчивости.

Плагин журнала аудита `Person` — это мощный инструмент, который может помочь вам выполнить ряд проверок соответствия требованиям. Наш опыт свидетельствует: это гораздо более производительное решение, чем использование триггеров, и оно хорошо интегрируется с программным обеспечением для управления конфигурацией и структурированного ведения журнала, что делает решение эффективным для многих заинтересованных групп.

Управление версиями для изменений схемы

В главе 6 были рассмотрены различные стратегии и инструменты, облегчающие масштабирование схемы. Поговорим о том, как эти стратегии обеспечивают соответствие нормативным требованиям.

Использование системы управления версиями как для отслеживания, так и для выполнения изменений в вашей схеме включает возможность встроенного

¹ Для начала вот целая страница, посвященная надежной пересылке журналов: <https://oreil.ly/76e9K>.

отслеживания того, кто запросил изменение, кто его рассмотрел и утвердил, а также как оно выполнялось в производственной среде. Это является веской причиной для использования отдельного репозитория для каждого кластера баз данных. По мере увеличения количества баз данных в компании вы обнаружите, что не все они созданы равными. Некоторые из них, например базы данных, хранящие финансовые сведения, требуют более строгого соблюдения нормативных требований, а некоторые применяются для экспериментов с не столь критическими продуктами. Когда придет время аудита, наличие записей об изменениях для каждого набора данных и кластера станет огромным облегчением.

Разделение управления данными и схемой кластера на основе требований соответствия также упрощает контроль за тем, кто может вносить или утверждать изменения схем в выбранной вами системе управления версиями. Обычно при проведении бизнес-аудита необходимо обосновать, кто может вносить изменения в базы данных. Сужение круга операторов, которые могут вносить изменения в данные, соответствует принципу безопасности, базирующемуся на минимальных привилегиях.

Управление пользователями баз данных

Изменения в базах данных не ограничиваются изменениями схемы. Вам также необходимо управлять пользователями баз данных и их привилегиями таким образом, чтобы их можно было отслеживать и повторять. Выясним, как можно выполнить некоторые общие требования контроля соответствия требованиям, которые касаются контроля доступа к базам данных.

Применяйте управление конфигурацией. Простой способ сделать отслеживание пользователей базы данных соответствующим требованиям — это использовать тот же процесс, с помощью которого обеспечивается совместимость изменений конфигурации базы данных. Вы управляете всем этим в репозитории управления конфигурацией и применяете систему управления версиями исходных текстов, процесс запросов на извлечение (pull request) и экспертную оценку (peer review), чтобы представить доказательства того, что все изменения пользователей базы данных были выполнены таким образом, что их можно проверить и отследить.

Планируйте изменение учетных данных. Из-за незапланированных инцидентов безопасности или из-за того, что у вас есть элемент контроля, требующий изменения учетных данных по расписанию, вам необходимо иметь план изменения учетных данных пользователей базы данных без ущерба для работоспособности приложения. Вероятно, это означает изменение как имени пользователя, так и строки пароля, используемых приложениями. Если вы еще не применяете

последнюю и самую лучшую основную версию с поддержкой двойного пароля, вот шаги, которые необходимо выполнить для изменения учетных данных базы данных в производственном приложении без ущерба для работоспособности сервиса.

1. Введите в базу данных новую пару «имя пользователя/пароль».
2. Убедитесь, что новые учетные данные имеют те же привилегии доступа, что и старые. В идеале вы должны делать это автоматически в процессе развертывания новых учетных данных, сравнивая разрешения с помощью команды `SHOW GRANTS` и подтверждая, что привилегии идентичны.
3. Создайте развертывание приложения, которое заменит учетные данные в конфигурации вашего приложения.
4. Перезапустите все экземпляры этой службы, чтобы убедиться, что используется новая пара.
5. Удалите старую пару имени пользователя и пароля.

Этот процесс должен работать одинаково независимо от того, является изменение плановым или срочным, предпринимаемым из-за утечки учетных данных или угрозы безопасности. Поскольку последнее не является событием, которое вы можете контролировать или предотвратить, лучше всего, если этот процесс будет автоматизирован или по крайней мере хорошо задокументирован в руководстве и станет выполняться регулярно, чтобы ваша команда не испытывала страха, когда это произойдет неожиданно.



Смена паролей пользователей баз данных в MySQL раньше была сложной организационной задачей, которую нужно было выполнить без ущерба для доступности. В MySQL 8.0.14 появилась поддержка двойных паролей, что в сочетании с поддержкой политики истечения срока действия паролей значительно упрощает работу.

Удаляйте неиспользуемых пользователей базы данных. Любой неиспользуемый пользователь базы данных, который остается активным на ваших экземплярах, — это ненужная угроза безопасности. Важно регулярно проверять пользователей базы данных, активных в ваших экземплярах базы данных, сравнивая их с теми, которые настроены в приложениях, и удалять всех пользователей, которые не задействуются активно в приложениях.

При выполнении требований соответствия для вашей компании вы увидите, что многие из элементов контроля соответствия требуют, чтобы организация отслеживала любые изменения в определенных активах. Эти элементы контроля типичны для отчетов, таких как SOC 2, о которых мы рассказывали ранее в этой

главе, где основной задачей является представление доказательств целостности и безопасности данных.

Существует несколько способов узнать, задействуется определенный пользователь базы данных или нет. В главе 3 мы подробно рассматривали схему Performance Schema как способ проверки производительности сервера. В Performance Schema есть таблица `users`, в которой хранится информация о пользователях, подключавшихся к серверу ранее. Отслеживание исторических данных распространяется на весь период жизни серверного процесса и ограничено максимальным размером, допустимым для этой таблицы, в зависимости от того, что произойдет раньше. Поскольку таблица отслеживает пользователей, которые подключались, а не тех, которые этого не делали, вам нужно будет перебрать известных пользователей и посмотреть, не появляются ли они в этой таблице. Вот запрос для включения этого инструмента в Performance Schema:

```
mysql> UPDATE performance_schema.setup_instruments
-> SET ENABLED='YES' WHERE NAME='memory/sql/user_conn';
Query OK, 1 rows affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

Как только вы включите эту функцию, для поиска данной информации можно будет воспользоваться таблицей `performance_schema.users`.

Если для контроля соответствия вы применяете решение для ведения журналов аудита, например плагин Percona, который упоминался ранее, то можете использовать эти журналы для определения того, подключался ли пользователь к экземплярам базы данных в течение определенного количества недель.

Независимо от того, каким способом вы это определяете, рекомендуется установить политику, согласно которой после шести месяцев бездействия пользователь базы данных, который не подключался к ней, будет удален. Эта практика поможет предотвратить получение доступа, который не нужен и в настоящее время является помехой.

Базы данных, которыми вы помогаете управлять, будут попадать в сферу контроля, требующего определенного уровня осматрительности. По мере того как компания становится более зрелой и в ней начинают задумываться о том, чтобы сделать ее более соответствующей нормативным требованиям, вам понадобятся материалы для демонстрации доказательств того, что изменения в производственных базах данных проверяются и отслеживаются до их применения. Еще одно действие, на котором будет сосредоточен контроль соответствия, — это подтверждение вашей способности восстанавливать данные и сервис в случае катастрофического события. Для этого нужно перейти к резервному копированию и восстановлению и посмотреть, как они вписываются в этот процесс.

Процедуры резервного копирования и восстановления

В главе 10 рассматриваются различные виды резервного копирования. Резервные копии, безусловно, важны. Они могут быть чрезвычайно полезными при инцидентах, а также являются ключевой частью многих механизмов контроля соответствия. В большинстве реализаций SOC 2 предусмотрены средства контроля как для создания, так и для тестирования резервных копий (в любом случае резервные копии следует тестировать). По мере роста числа кластеров баз данных, которыми вы управляете, вы быстро обнаружите, что не можете продолжать вручную выполнять такие процессы, как резервное копирование и тестирование резервных копий, или даже сообщать об успехах и неудачах после визуального чтения файлов журналов.

Вам необходимо выполнить некоторые требования при оценке того, как вы будете управлять резервным копированием, для соблюдения требований соответствия нормативам.

- Следует автоматизировать процесс резервного копирования.
- Необходимо, чтобы процесс резервного копирования предупреждал вас о сбоях.
- Нужно автоматизировать тестирование резервных копий.
- Неудачные тесты резервных копий также должны быть событиями, которые вы можете где-то отслеживать.

Далее мы обсудим, как планировать резервное копирование и тесты резервного копирования.

Выполнение автоматического резервного копирования и тестов резервного копирования

Чтобы выполнить перечисленные требования, вам нужен механизм, который не просто выполняет запланированные задания (например, `cron` в системах Linux), но и может работать по расписанию, а также имеет возможность отправлять события в систему мониторинга и систему тикетов, чтобы предупредить о сбоях и отслеживать сбой для последующего аудита. Один из способов сделать это — запустить резервное копирование и тестирование резервного копирования в качестве мониторинга¹, но резервное копирование может за-

¹ В статье блога «Использование Sensitive для задач администратора баз данных» (<https://oreil.ly/Meunp>) вы найдете несколько примеров того, как сделать резервное копирование частью вашего решения для мониторинга базы данных.

нять некоторое время, особенно если у вас есть несколько экземпляров баз данных размером в терабайты. Запуск резервного копирования в качестве элемента мониторинга может работать до тех пор, пока система мониторинга, с помощью которой вы собираетесь решать задачи резервного копирования, способна обрабатывать проверки, которые могут выполняться за периоды, значительно превышающие обычные несколько секунд. Поэтому убедитесь, что ваша команда, работающая с системой мониторинга, осведомлена об этом сценарии использования.

Если система мониторинга не может справиться с таким сценарием применения, убедитесь, что у вас есть способ оставить «хлебные крошки» для отслеживания того, что резервное копирование произошло и завершилось успешно и что тест резервного копирования выполнен, и тоже успешно. Одной из таких «хлебных крошек» может быть файл с временной меткой, который процесс резервного копирования редактирует в конце каждой резервной копии или теста резервного копирования в качестве доказательства того, что задача была выполнена и завершена. Как только эта «хлебная крошка» будет создана, можете использовать систему мониторинга для более быстрой проверки ее существования.

Во всех этих стратегиях отслеживание успешного завершения резервного копирования, а также любых неудачных резервных копий, показывающее, что они были превращены в надлежащим образом отслеживаемые рабочие элементы, — это то, что вам хочется иметь и что требуется вашему элементу контроля SOC 2.

Централизованные журналы резервного копирования и тестов резервного копирования

Вас также могут попросить показать журналы, подтверждающие успешное завершение резервного копирования и тестирования резервных копий. Было бы полезно подготовиться к такому элементу аудита, используя централизованное решение для ведения журналов, в которое вы сможете отправлять журналы для обеспечения непрерывности. Помните, что решения, которые мы создаем для этих бизнес-потребностей, должны предполагать, что серверы легко и многократно заменяемы, а не изготавливаются на заказ. Поэтому локальные файлы журналов на случайном экземпляре не идеальный вариант, если вы когда-нибудь выведете эту машину из эксплуатации и замените ее до следующего аудита. Вы хотите, чтобы все связанные с бизнесом активы, такие как журналы резервного копирования, хранились централизованно в месте, доступ к которому может получить любой пользователь с необходимыми правами доступа.

Планирование аварийного восстановления с помощью резервных копий

Частью SOC 2 является требование о надлежащем планировании аварийного восстановления. Это означает доказательство того, что вы тестируете все резервные копии, которые создает ваша система, отслеживаете, когда эти тесты завершаются неудачей и то, что сбой был устранен, и в идеале у вас есть представление о том, сколько времени занимает аварийное восстановление данных. Последняя часть требует отслеживания показателей того, сколько времени занимает тестирование резервной копии. В главе 2 упоминался размер экземпляра базы данных в качестве показателя для определения того, становится ли он слишком большим для восстановления резервной копии в течение запланированного целевого времени. Способ сделать этот цикл самосовершенствующимся состоит в том, чтобы резервное копирование и скрипты, тестирующие резервное копирование, также отправляли сведения, показывающие, сколько времени занимает каждое из них. Таким образом, у вас есть показатели по кластеру баз данных, говорящие о том, сколько времени занимает резервное копирование и сколько — восстановление и тестирование резервных копий. Теперь у вас есть способ отслеживать, не становится ли тот или иной набор данных слишком большим для того, что ожидает бизнес с точки зрения MTTR. Если это так, то у вас есть данные, позволяющие либо определить приоритетность работы по разделению набора данных до приемлемого размера, либо пересмотреть SLA для восстановления.

Важное заключительное замечание о резервном копировании: лицам, отвечающим за безопасность, потребуется контролировать доступ как к действующим базам данных, так и к резервным копиям. Убедитесь, что настройки резервного копирования у вашего любимого облачного провайдера средства управления доступом не используют разрешение по умолчанию для сегментов резервного копирования. Многие нарушения безопасности происходят не из-за взлома действующей инфраструктуры, а из-за утечки резервных копий из какого-то сегмента хранилища.

Резюме

Соответствие нормативным требованиям — это широкий спектр политик и элементов контроля, а также их интерпретаций. Они влияют не только на то, как вы управляете базами данных в своем бизнесе, но и на работу юридического, финансового и ИТ-отделов и даже на то, как вы внедряете изменения в свое программное обеспечение. В этой главе мы уделили основное внимание тому, как каждый распространенный тип нормативных требований влияет на ваши

обязанности как инженера баз данных. Затем рассмотрели различные методы работы и архитектурные решения, на которые могут повлиять эти нормативные акты и которые вам также необходимо учитывать.

В целом лучший способ предотвратить кошмары, связанные с контролем, — это заблаговременное планирование. Разделите пользователей ваших приложений, разработайте стратегию ротации учетных данных и убедитесь, что пароли всегда хранятся в зашифрованном виде и никогда — в виде обычного текста. Убедитесь, что до начала протоколирования доступа к базе данных у вас уже есть конвейер протоколирования, которому можно доверять. И наконец, все изменения схемы следует контролировать и регистрировать.

Цель этой главы не в том, чтобы перегрузить вас размышлениями обо всех элементах контроля, относящихся к вашей инфраструктуре в целом, а в том, чтобы упростить задачу представления доказательств для ее частей, подпадающих под действие каждого нормативного акта, и максимально автоматизировать или упростить ее выполнение. В конечном счете эти элементы контроля предназначены для защиты бизнеса и конфиденциальности ваших клиентов. Четкое понимание того, на что нацелен каждый элемент контроля, сделает эту важную задачу лучше выполнимой для вас и вашей команды по мере роста компании и ее выхода на более широкие рынки.

Обновление MySQL

Обновление — это компромисс между стабильностью¹ и функциональными возможностями. Вы должны учитывать это при выборе обновления. Одним из наибольших преимуществ использования MySQL является его широкая база установки. Это означает, что вы получаете преимущество от того, что множество других людей тестируют и применяют MySQL. Если вы обновитесь до слишком новой версии, то можете неосознанно внести ошибку или регрессию в свою среду. Если же останетесь слишком далеко позади, можете столкнуться с неочевидными ошибками или не сможете воспользоваться преимуществами функции, которая была оптимизирована для повышения производительности.

Зачем обновлять версию

Принятие решения об обновлении версии может быть рискованным процессом. Обычно оно включает в себя резервное копирование всех ваших данных, тестирование изменений, а затем запуск процесса обновления. Прежде чем переходить к деталям, важно понять, почему вы хотите обновить версию.

Существует ряд причин для обновления.

- *Уязвимости в системе безопасности.* С годами это стало менее вероятным, но все еще существует возможность того, что пользователи обнаружат уязвимости в системе безопасности MySQL. Вы или ваша служба безопасности можете оценить их и решить, что следует выполнить обновление.

¹ Стюарт Смит, давний член сообщества MySQL, сформулировал известное правило dot-20: «[Правило] заключается в том, что часть программного обеспечения никогда не является по-настоящему зрелой до выпуска “точки-20”. Хотя это не является жестким и непреложным правилом, оно подчеркивает компромисс между новыми релизами и стабильностью».

- *Известные ошибки.* При обнаружении неизвестного или необъяснимого поведения в производственной среде рекомендуем выяснить, какую версию MySQL вы используете, а затем прочитать примечания к релизам последующих версий до последней. Вполне возможно, что ситуация, с которой вы столкнулись, — это программная ошибка в MySQL. Если ваша проблема решена, возможно, потребуется обновить MySQL.
- *Новые функциональные возможности.* MySQL не всегда придерживается строгой стратегии выпуска основных/минорных/точечных релизов в отношении того, как добавляются новые функциональные возможности. Многие ожидают, что точечный релиз (например, с 8.0.21 на 8.0.22) будет содержать только исправления ошибок, а незначительное изменение версии (с 8.0 на 8.1) включает незначительные функциональные возможности. Oracle часто выпускает в незначительных точечных релизах новые функции, которые могут повлиять на вашу рабочую нагрузку. Эта стратегия является палкой о двух концах, именно поэтому перед обновлением нужно прочитать все примечания к выпуску.
- *Поддержка окончания срока службы в MySQL.* Oracle устанавливает временные рамки окончания срока службы (EOL) для MySQL. Как правило, рекомендуется оставаться в пределах поддерживаемой версии, чтобы как минимум по-прежнему поддерживались исправления безопасности.

Теперь, когда мы рассмотрели различные факторы, которые влияют на решение об обновлении и о том, до какой конкретно версии, обсудим процесс планирования и безопасного завершения обновления.

Жизненный цикл обновления

Приняв решение о том, что обновление — это правильно, вы обычно предпринимаете следующие шаги.

1. Изучаете примечания к выпуску данной версии, включая все незначительные изменения.
2. Прочитываете примечания к обновлению в официальной документации.
3. Тестируете новую версию.
4. И наконец, обновляете свои серверы.

В примечаниях к выпуску часто содержится важная информация — о новых функциях, изменениях или устаревших функциях, и обычно список исправленных ошибок. В примечаниях к обновлению дается подробный обзор того, как

выполнить обновление. Здесь к вашему вниманию любая важная информация, которую необходимо знать, прежде чем продолжать.

Кроме того, у вас должен быть план действий на случай возникновения проблем, например, если запрос начнет выполняться плохо или, что еще хуже, произойдет сбой. Для всех основных и второстепенных изменений версий (например, с 8.0 до 5.7 или с 5.7 до 5.6) единственным способом перейти на более раннюю версию является восстановление резервной копии, сделанной до обновления. Это делает обновление особенно рискованным, поэтому убедитесь, что у вас есть план действий.



Важно отметить: начиная с версии MySQL 8.0, вы не можете понижать точечные версии релизов. Например, если используете версию 8.0.25, то не сможете перейти на 8.0.24 без экспорта всех данных и повторного импорта.

Тестирование обновлений

Прочитав примечания к выпуску и обновлению, вы должны хорошо представлять любые проблемы и области, на которые следует обращать внимание при тестировании. Следующий шаг — тестирование того, как новая версия будет обращаться с вашей рабочей нагрузкой. Вы также должны убедиться, что просмотрели конфигурационные файлы. В более новых версиях MySQL переменные часто переименовываются или полностью утрачивают свою актуальность.

Тестирование — сложный этап, и у каждого из методов имеются свои нюансы. Учтите риск, о котором мы упоминали ранее, при переходе на более раннюю версию вы должны использовать как можно больше методов тестирования до обновления.

Тестирование среды разработки

Надеемся, у вас есть среда разработки для своих данных. Это отличное место для начала тестирования как на общей базе данных для разработки, так и на автономной базе данных. Основная цель использования этой среды — выявить любые очевидные проблемы с синтаксисом. Большинство сред разработки не содержит данных такого же размера, как производственные, поэтому выполнение точного тестирования может быть затруднено. Например, вы можете запустить часто используемые запросы и убедиться, что они работают нормально, потому что обращаются только к десяти строкам таблицы. Когда же перейдете в производственную среду с 10 млн строк в той же таблице, можете увидеть регрессию.

Копия производственных данных

Другим вариантом может быть создание копии производственных данных и отправка на нее копии вашего SQL-трафика. Этот метод был продемонстрирован в статье <https://oreil.ly/yByfy> блога Code As Craft на сайте Etsy. Короче говоря, вы создаете вторую копию своей производственной базы данных, прекращаете использование репликации и обновляете MySQL в этой копии. После завершения обновления отправьте трафик как в производственную систему, так и в копию с помощью комбинации `tcpdump` и `ptquery-digest`. Ваше приложение по-прежнему использует только производственную систему для реального трафика, в то время как копия с обновленной версией может предоставить показатели производительности и выявить ошибки в синтаксисе.

Реплика

Если в вашей топологии есть реплики для чтения и имеется возможность отключить реплику от пула, можете сначала рассмотреть возможность обновления одной из реплик. Это позволит увидеть, как трафик чтения работает с реальной производственной нагрузкой. Если вы заметите ошибки или регрессию, то сможете удалить реплику и внести коррективы. Недостатком этого метода является то, что не получится протестировать производительность или трафик записи.

Инструментарий

Percona Toolkit предлагает инструмент `pt-upgrade`, который принимает входные запросы, запускает их для двух различных объектов и создает отчет, сообщающий о любых различиях в количестве строк, данных строк или ошибках. Поскольку он может принимать множество различных типов входных данных (журнал медленных запросов, общий журнал запросов, двоичные журналы), это может быть хорошим вариантом для получения дополнительного тестового покрытия.

Лучший способ его использования таков. Сначала соберите наиболее интересующие вас запросы с помощью либо журнала медленных запросов, либо двоичного журнала. Затем настройте две одинаковые системы, обновите одну из них до новой версии и запустите `pt-upgrade` на обеих, чтобы увидеть различия.

Масштабное обновление

Обновление MySQL очень просто и подробно описано в официальной документации к ней. Вкратце: если вы выполняете обновление на месте, то оставьте MySQL, замените двоичные файлы, запустите MySQL, а затем скрипт `mysql_upgrade`.

Этот процесс приходится повторять многократно, если вы делаете это в парке из сотен серверов MySQL. Советуем максимально автоматизировать его. Один из способов сделать это — использовать Ansible.

Вот рекомендуемый базовый процесс для выполнения безопасных обновлений, который можно применить в качестве руководства для создания Ansible playbook.

1. *Проверьте цель.* Первое, что необходимо сделать, — предотвратить любые случайные обновления производственных систем. Если у вас есть система, которую вы можете запросить, чтобы определить, активно ли база данных принимает трафик, — это подходящее место для проверки. Если вы следовали нашим советам из главы 5, то должны использовать флаг `read_only` для предотвращения непредвиденных записей на свои реплики. Это может стать хорошей альтернативой, если у вас нет системы, которую можно проверить. Если сервер доступен для записи, скорее всего, вы не захотите его обновлять, так как он может выполнять производственные записи. На этом шаге можно также убедиться, что вы еще не обновили сервер. Это позволит вам позже запустить на нем Ansible playbook, и он не предпримет никаких действий.
2. *Установите время простоя.* Надеемся, для вашей системы выполняется мониторинг. Следующий шаг включает в себя установку некоторой формы времени простоя или подавления предупреждений, чтобы вы не получали сообщений на этапе, когда MySQL перезапускается в новой версии.
3. *Другие предварительные условия.* Если у вас есть еще какие-то зависимые службы, такие как средство управления конфигурацией или другие инструменты мониторинга, которые будут генерировать предупреждения об ошибках, пока MySQL находится в автономном режиме, сейчас самое время отключить их.
4. *Удалите старые пакеты.* На этом этапе мы предпочитаем удалить все установленные пакеты для MySQL. Это поможет избежать любых конфликтующих пакетов для основных версий (от 5.7 до 8.0).
5. *Установите новые пакеты.* Далее необходимо установить новые пакеты в систему.
6. *Запустите `mysqld`.* Запустите службу `mysqld`.
7. *Запустите `mysql_upgrade`.* Если MySQL старше версии 8.0¹, запустите процесс `mysql_upgrade`. Особое примечание: если вы используете MySQL с параметром `super_read_only`, как мы рекомендуем, то на этом шаге нужно установить для `mysql_upgrade` значение `OFF`.

¹ В MySQL 8.0 процесс `mysql_upgrade` перенесен в запуск самого сервера. Теперь нет необходимости запускать его как дополнительный шаг.

8. *Перезапустите mysqld.* Сейчас мы предпочитаем полностью перезапустить `mysqld`. Это гарантирует, что он корректно запустится с обновленными файлами и ваши конфигурационные файлы тоже будут работать.
9. *Убедитесь, что можете подключиться.* Просто подключитесь и выполните `SELECT 1`, чтобы убедиться, что MySQL запущен и работает.
10. *Восстановите все отключенные службы.* Если вы отключили какие-либо инструменты управления конфигурацией или мониторинга, включите их снова.
11. *Очистите время простоя.* Выведите сервер из режима простоя, чтобы можно было наблюдать, нет ли сбоев в процессе обновления.

С помощью описанного процесса вы можете направить модуль `runbook` на любой сервер и обновить только необновленные узлы, которые не принимают трафик.

Резюме

Существует множество причин для обновления MySQL, наиболее убедительными из которых являются исправление ошибок, с которыми вы часто сталкиваетесь, или возможность использовать новую функцию. Например, в MySQL 8.0 появилась функция InnoDB, позволяющая добавлять столбцы мгновенно и не перестраивая всю таблицу. Такое усовершенствование функции может значительно сэкономить время компаниям, которые выполняют большое количество команд `ALTER TABLE ... ADD COLUMN`. Усилия, которые вы вложили в создание безопасного процесса обновления, в итоге окупятся за счет времени, сэкономленного на выполнении операций добавления столбцов, а также благодаря тому, что разработчики усовершенствовали свои навыки.

Однако обновление основных версий может оказаться непростой задачей. Вам необходимо приложить много усилий для тестирования обновлений на предмет любых побочных эффектов. Как правило, нужно проверить наличие любых отклонений в задержках запросов или новых ошибок в результате обновления. Как только убедитесь в их отсутствии, без спешки развертывайте обновление и обеспечьте процесс отката.

Наконец, если вам нужно управлять большим парком серверов, подумайте о том, чтобы вложить значительные средства в максимально возможную автоматизацию процесса. Автоматизация может сделать процесс обновления легко повторяемым и более быстрым, чем при непосредственном входе на каждый сервер, а также несколько снизить вероятность опечаток и случайных простоев из-за того, что вы находитесь не на том сервере.

MySQL на Kubernetes

Если вы хоть немного работали в сфере технологий в течение последних пяти лет, то, скорее всего, слышали о Kubernetes, сотрудничали с командами, использующими Kubernetes, или видели множество выступлений на конференциях, или читали много сообщений в блогах, объясняющих Kubernetes. Если в вашей организации работают собственные кластеры Kubernetes, то в какой-то момент вас спросят, не стоит ли запускать на них MySQL. На первый взгляд это кажется разумным решением. Управление многими кластерами Kubernetes — это сложная задача, для решения которой обычно требуются особые специалисты, и в вашей организации разумно использовать этот опыт не только для рабочих нагрузок без сохранения состояния. Есть веские причины, чтобы изучать возможности запуска MySQL на Kubernetes, и не очень веские не делать этого. Давайте здесь разберем некоторые страхи, неуверенность, сомнения относительно запуска MySQL на Kubernetes.

Предоставление ресурсов с помощью Kubernetes

До того как Kubernetes достигла пика технической популярности, многие компании либо создавали полностью индивидуальные технологические стеки для инициализации виртуальных машин и серверов на голом «железе» и управления ими, либо объединяли проекты с открытым исходным кодом, которые выполняли меньшие части жизненного цикла ресурсов. Затем появилась Kubernetes как более полная экосистема для управления вычислительными ресурсами и ресурсами хранения, и перспектива ее использования в качестве стека подготовки для управления всеми ресурсами становилась все более привлекательной. Тем не менее нагрузки с сохранением состояния, такие как MySQL, отставали и были лишены этой дополнительной возможности, потому что здравый смысл гласил: «Базы данных не могут работать в контейнерах».

Тщательно определите свою цель

Важно помнить, какое конкретное преимущество мы хотим здесь получить. Kubernetes отлично подходит для нагрузок без сохранения состояния, поскольку обеспечивает эластичность и эффективность вычислительных ресурсов. Однако при рассмотрении унифицированного стека инициализации разумно свести выигрыш до «Мы хотим использовать Kubernetes только для предоставления и настройки систем для ресурсов базы данных». Это означает, что вам нужно заранее уяснить, что рабочие нагрузки баз данных, которые будут предоставляться с помощью Kubernetes, будут управляться отдельно от рабочих нагрузок без сохранения состояния, потребуют от оператора других навыков и станут по-разному обрабатывать отказы контейнеров.

Выберите плоскость управления

В настоящее время существуют различные операторы MySQL, но выбор наилучшего из них будет зависеть в основном от того, что вы определите в качестве плоскости управления MySQL в Kubernetes. Понадобится ли вам оператор, который делает все — инициализацию, обработку отказов и управление подключением к базам данных? Или вы просто будете использовать Kubernetes в качестве стека инициализации и задействовать другие средства для управления базами данных после их ввода в эксплуатацию? Заранее определите, чего ожидаете от плоскости управления, поскольку от этого будет зависеть множество более тонких деталей функциональности.

Более тонкие детали

Как только вы решили начать выделение ресурсов MySQL с помощью Kubernetes, вам необходимо договориться в вашей организации о том, какой размер данных подходит для этого решения. Помните, что теперь это новая операционная модель для работы с реляционной базой данных и на этом хуже изученном пути все становится сложнее по мере увеличения объема данных. Вот некоторые важные моменты, которые следует учитывать при совместной работе с командой разработчиков Kubernetes (надеемся, у вас есть специальная команда для этого) по вопросам поддержки рабочих нагрузок с сохранением состояния.

- Какой максимальный размер набора данных для одного экземпляра базы данных будет поддерживаться?
- Будете ли вы монтировать тома в контейнеры и управлять восстановлением контейнеров отдельно от монтирования данных? Или данные будут частью контейнера?

- Какая максимальная пропускная способность запросов будет поддерживаться? Как вы станете управлять ресурсами?
- Как будете обеспечивать, чтобы узлы Kubernetes, на которых выполняются рабочие нагрузки баз данных, были выделены для этого и не применялись совместно с более эластичными рабочими нагрузками без сохранения состояния?
- Какую плоскость управления вы будете использовать для запуска экземпляров базы данных? Является ли она нативной для Kubernetes?
- Как будет происходить резервное копирование? Каков процесс восстановления?
- Как вы будете контролировать и безопасно внедрять изменения конфигурации и обновления MySQL?
- Как станете обновлять сами кластеры Kubernetes, не вызывая сбоев?

Взаимопонимание с командой инженеров Kubernetes в вопросе о том, как будет работать это решение, имеет важное значение при создании хорошо зарекомендовавших себя SLO для функциональных групп, желающих использовать данное решение. Это также важно и для правильного информирования команд функциональных групп о том, какие задачи оно выполняет, а что им предстоит сделать самостоятельно.

Наш совет при запуске MySQL на Kubernetes заключается в следующем: нужно инвестировать в изучение плоскости управления, которая уже проверена и доказала свою эффективность в экосистеме Kubernetes, такой как Vitess. Но прежде, чем пытаться бежать, пройдитесь. MySQL не должна быть первым подопытным кроликом для запуска рабочих нагрузок на Kubernetes в вашей организации. Всегда доказывайте жизнеспособность и изучайте острые углы в команде с рабочими нагрузками без сохранения состояния, прежде чем пытаться запустить более сложные сценарии использования, такие как MySQL. Определяя наилучшие начальные сценарии применения для внедрения, начните с небольших наборов данных (баз данных, занимающих всего несколько гигабайт на диске) и менее важных, чтобы ваша команда, команда Kubernetes и функциональные команды познакомились с новой операционной моделью выполнения рабочих нагрузок с сохранением состояния в Kubernetes с меньшим риском для бизнеса¹.

Выполнение рабочих нагрузок с сохранением состояния на Kubernetes совершенствовалось в течение последних нескольких лет и продолжает развиваться

¹ В качестве отличного выступления на конференции о запуске рабочих нагрузок баз данных на Kubernetes мы рекомендуем доклад Алисы Голдфусс The Container Operator's Manual (<https://oreil.ly/TVD6c>).

благодаря критически важному вкладу компаний, которые потратили огромное количество ресурсов на то, чтобы приблизить эту работу к практическому использованию. Однако процесс все еще находится в зачаточном состоянии по сравнению с запуском непосредственно на виртуальных машинах, и вы обнаружите, что медленный и тщательный подход к внедрению — это то, что окупается в долгосрочной перспективе. Особенное внимание обратите на то, как выглядят режимы отказа MySQL на Kubernetes, и спросите себя: «Если что-то пойдет не так, как я смогу снова собрать все? Потеряю ли я данные?». Убедитесь, что у вас есть ответ на эти вопросы.

Резюме

Kubernetes сейчас является одной из самых быстрорастущих инфраструктурных платформ в сфере технологий, и на то есть веские причины. Скорость разработки, которую она обеспечивает, и богатая экосистема, поддерживаемая Cloud Native Computing Foundation, делают ее привлекательной инвестицией для компаний. Однако вы должны рассматривать такие решения, как запуск MySQL на Kubernetes, через призму риска и выгоды для вашей команды и всей компании. Убедитесь, что у вас есть общее понимание того, какое место в Kubernetes вашей организации занимают сервисы с сохранением состояния, такие как хранилища данных. Вполне понятно желание использовать существующие инвестиции в Kubernetes для всех рабочих нагрузок, но оно должно быть хорошо согласовано с потребностями в стабильности уровня хранилища данных.

Об авторах

Сильвия Ботрос — архитектор программного обеспечения в Twilio. За время работы в SendGrid она помогла создать платформу базы данных с возможностью отправки миллиардов электронных писем, поддержки других продуктов и разработки хранилищ данных от начала до производственной эксплуатации.

Джереми Тинли — старший системный инженер Etsy с более чем 20-летним опытом работы с MySQL. На протяжении своей карьеры он управлял десятками тысяч экземпляров MySQL, обеспечивая их доступность, надежность и эффективность работы.

Иллюстрация на обложке

На обложке изображен ястреб-перепелятник (*Accipiter nisus*), небольшая хищная птица семейства соколиных, обитающая в Евразии и Северной Африке. У перепелятников длинный хвост и короткие крылья, самцы голубовато-серые со светло-коричневой грудкой, а самки — коричнево-серые с почти полностью белой грудкой. Самцы обычно несколько меньше (28 см), чем самки (38 см).

Ястребы-перепелятники живут в хвойных лесах и питаются мелкими млекопитающими, насекомыми и птицами. Гнездятся на деревьях, а иногда на уступах скал. В начале лета самка откладывает от четырех до шести яиц — белых, с красными и коричневыми пятнами, в гнезде, устроенном на ветвях самого высокого дерева. Самец кормит самку и птенцов.

Как и все ястребы, перепелятник способен к стремительным рывкам в полете. Независимо от того, парит птица или планирует, она совершает характерные движения: хлоп-хлоп — планирование. Длинный хвост позволяет ястребу легко лавировать и поворачивать на лету.

Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой исчезновения, однако все они важны для нашего мира.

Обложка выполнена Карен Монтгомери и основана на старинной гравюре из книги Людеккера *The Royal Natural History*.

Сильвия Ботрос, Джереми Тинли

MySQL по максимуму

4-е издание

Перевел с английского В. Дмитриуценок

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>Н. Гринчик</i>
Научный редактор	<i>В. Дмитриуценок</i>
Литературный редактор	<i>Н. Рощина</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>О. Андриевич, Е. Павлович</i>
Верстка	<i>Г. Блинов</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,

Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 03.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 31.01.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 34,830. Тираж 700. Заказ 0000.

Робин Никсон

СОЗДАЕМ ДИНАМИЧЕСКИЕ ВЕБ-САЙТЫ С ПОМОЩЬЮ PHP, MYSQL, JAVASCRIPT, CSS И HTML5

6-е издание



Новое издание бестселлера описывает как клиентские, так и серверные аспекты веб-разработки. Книга, наполненная ценными практическими советами и подробным теоретическим материалом, поможет вам освоить динамическое веб-программирование с применением самых современных технологий. Для закрепления усвоенных знаний автор расскажет, как создать полнофункциональный сайт, работающий по принципу социальной сети.

КУПИТЬ

O'REILLY®

4-е издание

MySQL

по максимуму

Проверенные стратегии



Сильвия Ботрос
Джереми Тинли