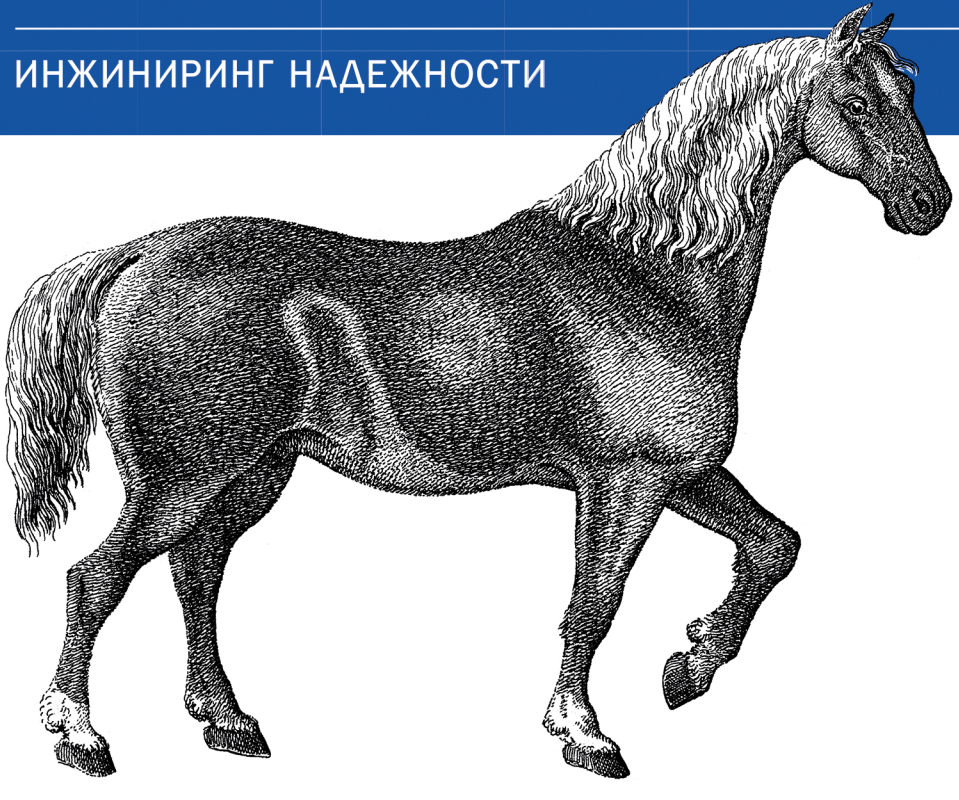
O'REILLY®

Базы данных

ИНЖИНИРИНГ НАДЕЖНОСТИ



Лейн Кэмпбелл
Черити Мейджорс

ПИТЕР®

Database Reliability Engineering

Designing and Operating Resilient Database Systems

Laine Campbell and Charity Majors

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Лейн Кэмпбелл
Черити Мейджорс

Базы данных

ИНЖИНИРИНГ НАДЕЖНОСТИ

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2020

ББК 32.973.233-018.2
УДК 004.65
К98

Кэмпбелл Лейн, Мейджорс Черити

К98 Базы данных. Инжиниринг надежности. — СПб.: Питер, 2020. — 304 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1310-1

В сфере IT произошла настоящая революция — с инфраструктурой стали работать как с кодом. Этот процесс создает не только новые проблемы, но и возможности для обеспечения безотказной работы баз данных. Авторы подготовили это практическое руководство для всех, кто желает влиться в сообщество современных инженеров по обеспечению надежности баз данных (database reliability engineers, DBRE).

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

ББК 32.973.233-018.2
УДК 004.65

ISBN 978-1491925942 англ. Authorized Russian translation of the English edition of Database Reliability Engineering (ISBN 9781491925942) © 2018 Laine Campbell and Charity Majors 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 978-5-4461-1310-1

Краткое содержание

Оглавление

Предисловие

Мы сейчас переживаем время беспрецедентных преобразований и потрясений в индустрии баз данных (БД). Жизненные циклы внедрения технологий ускорились до такой степени, что у кого угодно голова пойдет кругом — и из-за сложных проблем, и из-за открывающихся возможностей.

Архитектура БД развивается столь быстро, что привычные задачи уже не требуют решения, а связанные с ними навыки, в которые вложено так много сил, утратили актуальность. Новые требования безопасности, концепция инфраструктуры как кода (Infrastructure as Code, IaC), возможности облачных технологий (таких как база данных как сервис (Database as a Service, DBaaS)) позволили — а фактически потребовали от нас — переосмыслить пути разработки.

Нам пришлось перейти от традиционных административных задач к процессу, в котором основное внимание уделяется архитектуре, автоматизации, разработке программного обеспечения (в широком смысле), непрерывной интеграции и распространению, средствам мониторинга и обслуживания систем. При этом ценность и значимость данных, которые мы защищали и о которых заботились все это время, увеличились как минимум на порядок, и нет причин ожидать, что они не будут продолжать расти. Сегодня у нас есть прекрасная возможность внести существенный вклад в эту сферу.

Несомненно, многие из тех, кто когда-то считали себя выдающимися администраторами БД, сейчас рискуют утратить позиции или даже вовсе остаться за бортом. Вместе с тем в нашу область приходят новички, которые жаждут новой организационной парадигмы. Решение в обоих случаях одно и то же: с готовностью и радостью учиться, самосовершенствоваться, сохранять оптимизм, энтузиазм и уверенность, необходимые для выполнения задачи и доведения ее до конца, несмотря на неизбежные трудности и подводные камни. Эта книга — замечательное достижение. В ней описан новый подход к проектированию и эксплуатации инфраструктуры БД, и она представляет собой руководство и сборник идей. В книге все то, что мы привыкли делать, обсуждается и переосмысливается в новом ключе — как обеспечение надежности БД.

Пол Валле (Paul Vallée), президент
и исполнительный директор компании Pythian

Введение

В этой книге мы надеемся продемонстрировать вам общие принципы для перехода на следующий уровень работы с базами данных — уровень *специалиста по обеспечению надежности баз данных* (DBR-инженер, Database Reliability Engineer, DBRE). Как многим представляется работа администратора баз данных? Любой программист или системный инженер, сотрудничавший с этими загадочными личностями, имеет, вероятно, свое несколько предвзятое мнение о них.

Традиционно администраторы БД (Database Administrators, DBA) досконально разбирались во внутреннем устройстве БД, жили и дышали оптимизатором, системой обработки запросов, настройкой и созданием высокопроизводительных специализированных систем. Когда им это потребовалось, они приобрели и другие навыки, позволяющие улучшить работу базы данных. Они узнали, как распределять нагрузку между процессорами (Central Processing Units, CPU) компьютеров и дисками, как настраивать БД для использования особенностей CPU и как оценивать требуемую емкость подсистем хранения.

Когда администраторы БД столкнулись с проблемами области видимости, они научились строить графы для параметров сущностей, определенных в качестве ключевых. Столкнувшись с архитектурными ограничениями, узнали об уровнях кэширования. Встретившись с ограничениями отдельных узлов в системе, они изучили новые подходы и типовые решения в проектировании, такие как сегментирование данных (sharding), и способствовали разработкам в этом направлении. В то же время они продолжали осваивать новые методики работы: аннулирование кэша, перебалансировку данных и непрерывные изменения БД.

Однако администраторы БД долгое время творили в башне из слоновой кости. Они пользовались другими инструментами, работали на другом оборудовании и писали на других языках. Администраторы БД писали на SQL, системные инженеры — на Perl, программисты — на C++, веб-разработчики — на PHP, а сетевые инженеры предпочитали собственные инструменты. Лишь половина команд пользовалась хоть каким-то контролем версий, и они, конечно же, не общались между собой и не заходили на территорию друг друга. Да и как бы они могли это сделать? Это было бы похоже на пересечение границ другой страны.

Однако дни подобной модели, когда она еще может считаться эффективной и устойчивой, уже сочтены. В этой книге представлен взгляд на *обеспечение надежности* с точки зрения проектирования БД. Мы не рассчитываем осветить здесь все возможные проблемы, а опишем лишь то, что считаем важным, через призму вашего опыта. Эту основу можно будет применить к различным хранилищам данных, архитектурам и организациям.

Почему мы написали эту книгу

Мы мечтали об этой книге около пяти лет. Лейн пришла на должность администратора без какой-либо специальной технической подготовки. Она не была ни инженером-программистом, ни системным администратором, а решила построить карьеру в технической сфере после ухода из мира музыки и театра. Имея такой опыт, она легко нашла в базах данных знакомые ей концепции структуры, гармонии, контрапункта и оркестровки.

С тех пор ей пришлось иметь дело, вероятно, с сотней администраторов БД: нанять, обучить их и работать с ними. Люди, работающие с базами данных, — очень разношерстная компания. Одни пришли из программирования, другие — из системного администрирования. Есть и такие, кто раньше занимался аналитикой данных и бизнесом. Однако все они отличались чувством ответственности за безопасность и доступность данных компании.

Мы исполняли роли управляющих данными с жесткостью, граничащей с нездоровой жестокостью. А еще мы стали связующим звеном между разработчиками программного обеспечения и системными инженерами. Кто-то может сказать, что мы стали первыми системными инженерами (DevOps), стоящими на стыке областей.

Опыт Черити тесно связан с запуском стартапов и эксплуатационной работой. За ее плечами славная, хоть и пестрая история быстрого запуска (bootstrapping) инфраструктур, молниеносного принятия решений, способных создать или разрушить стартап, рискованных сложных решений в условиях сильно ограниченных ресурсов. В основном плюс-минус успешных. К тому же она оказалась администратором БД, который любит данные. Она всегда работала в командах эксплуатации, где не было отдельного администратора БД, поэтому команды разработчиков и эксплуатации программного обеспечения в итоге делили эту работу между собой.

Имея столь разное прошлое и уже довольно долго работая с базами данных, мы признали и приняли тенденции последнего десятилетия. Жизнь администратора БД часто трудна и незаметна. Теперь у нас есть инструменты и коллективная поддержка, что позволит изменить эту роль, причислить администраторов БД к людям

первого сорта и сосредоточиться на самых важных областях, в которых такие специалисты могут принести пользу.

С помощью этой книги мы хотели бы помочь следующему поколению инженеров построить по-настоящему удачную карьеру, продолжая то, что было начато предшественниками.

Для кого эта книга

Для всех, кто занимается проектированием, созданием и эксплуатацией надежных хранилищ данных. Она для вас, если вы разработчик программного обеспечения и хотите расширить свои познания о БД. Или если вы системный инженер, желающий сделать то же самое. Если вы профессионал в области БД, который хочет расширить набор навыков, то эта книга будет полезна и вам. А новичку в этой области она поможет составить ясное представление о работе с базами данных.

Мы предполагаем, что у вас уже есть некоторые технические знания в области администрирования операционных систем Linux/Unix, а также веб-систем и/или облачных архитектур. Предполагается также, что вы хорошо разбираетесь в одной из дисциплин — системном администрировании или разработке программного обеспечения — и заинтересованы в расширении своего технического кругозора и включении в него новой дисциплины — разработки БД. Либо, в другом варианте, вы начинаете карьеру в качестве специалиста по проектированию БД и стремитесь углубить технические знания.

Если же вы руководитель проекта, то книга поможет вам понять, что требуется для создания хранилищ данных, которые будут лежать в основе ваших сервисов. Мы твердо убеждены в том, что руководителям необходимо понимать принципы работы и организации БД, чтобы их команды и проекты были более успешными.

Да, возможно, у вас нет специального технического образования. Может быть, вы были бизнес-аналитиком и стали администратором БД неожиданно, сменив специализацию и научившись управлять базами данных. Есть много специалистов по базам данных, пришедших в эту область через Excel, а не через разработку или системное администрирование.

Структура издания

Мы структурировали информацию в виде двух условных частей. Первая представляет собой основной учебный курс по эксплуатации баз данных. Это основные операции с БД, которые должен знать каждый, будь то инженер по базам данных, разработчик программного обеспечения или владелец продукта. Во второй части

мы углубимся в данные: их моделирование, хранение, тиражирование, доступ к ним и многое другое. Кроме того, обсудим выбор архитектуры и конвейеры данных. Это должно быть захватывающе!

Есть веская причина тому, что мы уделяем столь пристальное внимание эксплуатации: вы не станете хорошим инженером по обеспечению надежности БД (DBRE), не будучи хорошим инженером по надежности вообще (RE). А им вы не станете, не будучи просто хорошим инженером. Современный DBR-инженер не только хорошо разбирается в основах системного проектирования, но и понимает проблемы данных, связанные с конкретной предметной областью.

Однако дело в том, что *запускать сервисы данных* может любой инженер. Сейчас мы говорим на одном языке. Мы используем одни и те же репозитории, одни и те же процессы проверки кода. Забота о БД — это расширение инженерной деятельности, взбитые сливки из специальных знаний и умений на вершине торта из функционирующих систем различного масштаба. Точно так же, для того чтобы стать сугубо сетевым инженером, нужно сначала выучиться на инженера, а затем получить дополнительные знания о том, как обрабатывать трафик, чего следует опасаться, каковы современные передовые методики, как оценивать топологию сети и т. д.

Вот краткий перечень того, что вас ожидает.

Глава 1 представляет собой введение в концепцию обеспечения надежности БД. Мы начнем с основополагающих принципов, перейдем к работе по сопровождению и эксплуатации — тому центру, вокруг которого вращается DBRE, — и, наконец, соорудим каркас для возведения концепции DBRE с опорой на пирамиду потребностей Маслоу.

Главу 2 начнем с требований уровня обслуживания сервисов. Они так же важны, как и требования к характеристикам продукта. В этой главе мы обсудим, что такое требования уровня обслуживания и как их определить, — что на самом деле легче сказать, чем сделать. Затем поговорим, как измерить соответствующие параметры и как работать с ними в динамике.

В главе 3 мы познакомимся с оценкой рисков и управлением ими. После обсуждения основ и рисков вообще перейдем к практическим процессам включения оценки рисков в разработку систем и БД. Рассмотрим также подводные камни и различные сложности.

В главе 4 поговорим об оперативном контроле. Обсудим показатели и события. Вы узнаете, как составить план того, что нужно начать измерять и что повторять регулярно. Подробно рассмотрим компоненты систем мониторинга и клиентские приложения, которые взаимодействуют с ними.

В главах 5 и 6 мы углубимся в проектирование инфраструктуры и управление ею. Обсудим принципы построения хостов для хранилищ данных. Мы погрузимся

в виртуализацию и контейнеризацию, управление конфигурацией, автоматизацию и оркестрацию, чтобы помочь вам разобраться во всех составных частях, необходимых для построения систем, которые обеспечивают хранение данных и доступ к ним.

Глава 7 посвящена резервному копированию и восстановлению данных. Это, пожалуй, самое важное для освоения DBE. Потерял данные — все, игра окончена. Исходя из требований к уровню качества сервиса, мы выбираем подходящие методы резервного копирования и восстановления данных, а также способы масштабирования и проверки этого важного, но часто пропускаемого шага.

В главе 8 мы обсудим управление версиями. Как тестировать, подготавливать и развертывать изменения в хранилищах данных? Как быть с изменениями в коде доступа к данным и SQL? Основные темы — развертывание, интеграция и распространение.

Глава 9 посвящена безопасности. Безопасность данных имеет решающее значение для работоспособности компании. В этой главе описаны стратегии планирования безопасности постоянно развивающихся инфраструктур данных и управления ею.

В главе 10 поговорим о хранении, индексации и репликации данных. Мы обсудим, как хранятся реляционные данные, а затем сравним их с отсортированными строками и структурированными объединениями деревьев. Рассмотрев разные варианты индексации, изучим топологии репликации данных.

Глава 11 представляет собой путеводитель в области хранения данных. Здесь мы рассмотрим множество различных факторов, которые вам предстоит находить и учитывать для тех хранилищ данных, что вы будете оценивать или сопровождать. Сюда входят как концептуальные особенности, имеющие огромное значение для разработчиков и архитекторов приложений, так и внутренние характеристики, связанные с физической реализацией хранилищ.

В главе 12 мы рассмотрим некоторые из наиболее распространенных типовых решений (паттернов), применяемых при проектировании архитектур распределенных БД, и конвейеры, с которыми они связаны. Начнем с архитектурных компонентов, из которых обычно состоит «среда обитания» (экосистема) базы данных, и познакомимся с обеспечиваемыми ими преимуществами, со связанными с ними сложностями и с основными принципами их использования. Затем исследуем архитектуры и конвейеры — хотя бы на нескольких примерах.

Наконец, в главе 13 мы обсудим создание культуры обеспечения надежности БД в организации. Исследуем различные способы, с помощью которых в современной организации можно превратить специалиста по обеспечению надежности баз данных из администратора в инженера.

Условные обозначения

В этой книге применяются следующие типографские обозначения.

Курсив

Курсивом выделяются новые термины.

Рубленый шрифт

Им обозначены URL-адреса, адреса электронной почты.

`Моноширинный шрифт`

Используется для листингов программ. Кроме того, в тексте им обозначены элементы программ, такие как имена переменных или функций, базы данных, типы данных, переменные среды, операторы и ключевые слова, имена и расширения файлов.

Этот элемент обозначает совет, подсказку или предложение.

Такой элемент обозначает примечание.

Данный элемент обозначает предупреждение.

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.

1 Введение в обеспечение надежности базы данных

Цель книги — предоставить вам руководство и создать базу для того, чтобы вы смогли стать действительно отличным инженером по обеспечению надежности БД.

Бен Трейнор (Ben Treynor), вице-президент по разработкам в Google, так говорит об обеспечении надежности: «SR-инженеры в основном выполняют ту же работу, которой ранее занималась служба эксплуатации, но их знания и навыки позволяют разрабатывать и реализовывать автоматизированные решения, заменяющие человеческий труд, и именно такую задачу ставит перед ними компания»[1].

Современные специалисты по БД должны быть инженерами, а не администраторами. Мы строим новое. Мы создаем новое. Занимаясь сопровождением систем, мы работаем в одной команде, и любая наша проблема становится общей. При проектировании, построении и эксплуатации хранилищ производственных (промышленных) данных и создаваемых внутри них структур мы, как инженеры, ориентируемся на повторяемые процессы, проверенные знания и экспертные оценки. А как специалисты по надежности БД, мы должны понимать принципы работы с базами данных и обладать опытом их эксплуатации.

Если вы обратите внимание на компоненты современных инфраструктур, не отвечающие непосредственно за хранение данных (non-storage), то увидите, что это системы, которые легко создавать, запускать и уничтожать с помощью программных и часто автоматизированных средств. Время жизни этих компонентов может измеряться днями, а иногда даже часами или минутами. Когда один из них исчезает, взамен появляется множество других, позволяющих сохранить качество обслуживания на ожидаемом уровне.

Наша следующая цель — перечислить основные принципы и методики проектирования, создания и эксплуатации хранилищ данных в рамках парадигм обеспечения надежности и культуры DevOps. Вы можете воспользоваться этими знаниями и применить их к любой технологии или среде БД, с которой вам придется работать на любом этапе развития организации.

---

[1] *Бейер Б., Джоунс К., Петофф Д., Мёрфи Н.* Site Reliability Engineering. Надежность и безотказность как в Google. — СПб.: Питер, 2019. — С. 38.

Основные принципы DBRE

Когда мы только начали писать эту книгу, один из первых вопросов, который мы себе задали, звучал так: какие принципы будут основополагающими для нового поколения специалистов по базам данных? Если мы намерены пересмотреть подход к проектированию хранилищ данных и управлению ими, то нам нужно определить основные правила поведения, которых придется придерживаться.

Защита данных

Защита данных всегда была и остается основополагающим принципом в работе любого профессионала в области БД. Общепринятый подход подразумевает следующее.

- Строгое разделение обязанностей между инженером-программистом и инженером баз данных.
- Строго определенные и регулярно тестируемые процессы резервного копирования и восстановления.
- Хорошо организованные и регулярно тестируемые процедуры безопасности.
- Дорогостоящее программное обеспечение для баз данных, гарантирующее высокий уровень надежности и сохранности данных.
- Дорогостоящая базовая система хранения с резервированием всех компонентов.
- Широкомасштабный контроль изменений и административных задач.

В командах, где царит культура сотрудничества, строгое разделение обязанностей может стать фактором не только обременительным, но даже ограничивающим инновации и скорость реагирования. В главе 8 мы обсудим способы создания систем поддержки и поговорим о том, как снизить потребность в разделении обязанностей. Кроме того, в такой среде можно больше внимания уделять тестированию, автоматизации и минимизации воздействия, а не расширять контроль изменений.

В настоящее время архитекторы и инженеры чаще, чем когда-либо, выбирают хранилища данных с открытым исходным кодом, а они не способны гарантировать такую стабильность и надежность хранения, какие прежде обеспечивала система управления базами данных (СУБД) Oracle. Иногда это дает выигрыш в производительности для команды, нуждающейся в быстром масштабировании. Выбор правильного хранилища данных и понимание последствий этого выбора мы рассмотрим в главе 11. Мы быстро привыкаем к существованию множества разнообразных инструментов для работы с данными и к возможности их эффективного выбора.

Базовая система хранения также претерпела значительные изменения. В мире, где системы часто виртуализируются, проектирование БД часто подразумевает создание сетевого и эфемерного хранилища. Мы подробно обсудим это в главе 5.

ХРАНЕНИЕ ПРОИЗВОДСТВЕННЫХ ДАННЫХ В ЭФЕМЕРНЫХ ХРАНИЛИЩАХ

В 2013 году компания Pinterest перевела копии своих баз данных MySQL в эфемерное хранилище на Amazon Web Services (AWS). Эфемерное хранилище, в сущности, означает, что если отдельное вычисление завершается сбоем или прекращает работу, то все, что хранилось на диске, теряется. Pinterest выбрала вариант эфемерного хранилища из-за постоянства его пропускной способности и низкой задержки.

Потребовались значительные инвестиции в автоматическое и высоконадежное резервное копирование и восстановление данных, а также в разработку приложений, позволяющих справиться с потерей кластера при восстановлении узлов. Эфемерное хранилище не дает возможности создавать моментальные снимки — это означало, что для восстановления необходимо было распространять по Сети полные копии базы данных, вместо того чтобы прикрепить моментальный снимок при подготовке к развертыванию журналов транзакций.

Это свидетельствует о том, что поддерживать надежность и безопасность данных в эфемерных средах вполне возможно при условии правильной организации процессов и использования правильных инструментов!

Новые принципы защиты данных могут быть такими.

- Ответственность за все общие данные несут все команды разработчиков разной специализации.
- Стандартизированные и автоматизированные процессы резервного копирования и восстановления данных проходят одобрение инженерами по обеспечению надежности БД.
- Стандартизированные политики и процедуры безопасности проходят одобрение отделом DBRE и отделом безопасности.
- Все политики внедряются путем автоматического распространения и развертывания.
- Хранилище данных выбирается исходя из требований к данным, а оценка потребности в надежности их хранения становится частью процесса принятия решений.
- Ориентироваться следует в первую очередь на автоматизированные процессы, резервирование и хорошо отлаженные методы, а не на сложное дорогостоящее оборудование.
- Внедрение изменений включается в процедуры развертывания и в автоматизацию инфраструктуры, причем акцент делается на тестировании, резервировании и смягчении возможных последствий.

Самообслуживание как фактор масштабирования

Талантливый DBR-инженер — более редкий специалист, чем инженер по обеспечению надежности информационных систем (SR-инженер, Site Reliability Engineer, SRE). Большинство компаний не могут позволить себе содержать более одного или двух таких специалистов. Таким образом, чтобы они приносили максимальную пользу, их деятельность должна выражаться в создании систем самообслуживания, которые смогут использовать все другие команды. Устанавливая стандарты и предоставляя инструменты, эти команды смогут развертывать новые сервисы и вносить соответствующие изменения в требуемом темпе, не выстаивая в очереди к и без того перегруженному работой инженеру БД. Вот несколько примеров таких методов самообслуживания.

- Обеспечить сбор необходимых показателей из хранилищ данных благодаря предоставлению соответствующих плагинов.
- Создать утилиты резервного копирования и восстановления данных, которые могут развертываться в новых хранилищах данных.
- Определить эталонные архитектуры и конфигурации для хранилищ данных, которые прошли одобрение командами эксплуатации и теперь могут развертываться командами разработчиков.
- Определить совместно с отделом безопасности стандарты для развертывания хранилищ данных.
- Разработать безопасные методы развертывания и тестовые сценарии для внедрения изменений в БД.

Другими словами, работа эффективного DBR-инженера заключается в расширении возможностей других специалистов и выдаче рекомендаций для них, а вовсе не в выполнении обязанностей сторожа или охранника.

Избавление от рутины

В командах SRE Google часто используют фразу «избавление от рутины» (toil), которая обсуждается в главе 5 книги Google SRE. В той книге есть такое определение: «Рутина — это ручная, однообразная, поддающаяся автоматизации оперативная работа, связанная с поддержкой работающего сервиса. Ее результаты не имеют ценности в перспективе, а трудоемкость растет линейно по мере роста сервиса»[1].

Эффективная автоматизация и стандартизация необходимы для того, чтобы DBR-инженеров не перегружали рутиной. В этой книге мы приведем несколько примеров рутины в работе специалистов по обеспечению надежности баз данных

[1] *Бейер Б., Джоунс К., Петофф Д., Мёрфи Н.* Site Reliability Engineering. Надежность и безотказность как в Google. — СПб.: Питер, 2019. — С. 91.

и расскажем о методах, позволяющих сократить ее объем. При этом понятие «рутина» остается расплывчатым, о нем существует множество предвзятых мнений, и у разных людей оно свое. Обсуждая рутину, мы говорим именно о выполняемой вручную, повторяющейся, нетворческой и неинтересной работе.

ВНЕСЕНИЕ ИЗМЕНЕНИЙ В БАЗУ ДАННЫХ ВРУЧНУЮ

Нередко инженеров БД просят проверить базы данных и внести в них изменения, которые могут включать в себя модификации таблиц или индексов, добавление, изменение или удаление данных или любые другие действия. Все считают, что администратор базы данных вносит эти изменения и отслеживает их влияние в режиме реального времени.

На одном клиентском сайте частота изменений была достаточно велика, и это значительно влияло на его работу. В итоге мы тратили около 20 часов в неделю, внося изменения во всей среде. Излишне говорить, что несчастный администратор баз данных, который тратил половину рабочей недели на выполнение этих повторяющихся задач, выдохся и уволился.

Столкнувшись с нехваткой ресурсов, руководство наконец позволило команде управления базой данных создать утилиту для автоматизированного применения изменений во всей схеме базы. Инженеры-программисты могли просто однократно запустить ее после того, как инженер базы данных рассмотрел и одобрил очередной набор изменений. Вскоре все стали доверять этому инструменту для внесения изменений, что позволило команде DBRE переключиться на интеграцию этих процессов в стек развертывания продукта.

Базы данных — это не «особенные снежинки»

Наши системы важны не более и не менее, чем любые другие компоненты, обслуживающие запросы бизнеса. Мы должны стремиться к стандартизации, автоматизации и устойчивости. Критически важно понимать, что компоненты кластеров баз данных — это не священная корова. Мы должны быть готовы потерять любой компонент и спокойно и эффективно заменить его. Уязвимые хранилища данных за стеклом в вычислительных центрах остались в прошлом.

Сейчас, чтобы показать разницу между «особенной снежинкой» и стандартным служебным компонентом, часто используют аналогию с домашними питомцами и крупным рогатым скотом. Считают, что первым ее употребил Билл Бейкер (Bill Baker) — инженер Microsoft. Сервер — «домашний питомец» — это тот, кого вы кормите, за кем ухаживаете, а если он болеет, то заботитесь о его здоровье. У него даже есть имя. На Travelocity в 2000 году у наших серверов были имена из сериала о Симпсонах, а два SGI-сервера под управлением Oracle назывались Patty и Selma.

У серверов из разряда «крупного рогатого скота» нет имен, а есть номера. Вы не тратите время на их настройку и уж тем более не входите на каждый отдельный хост. Когда они заболевают, вы просто удаляете их из стада. Если болезней будет слишком много, то вам, конечно, придется сохранить этих коров, чтобы разобраться в причинах.

Хранилища данных — один из последних оплотов традиции «домашних любимцев». В конце концов, они хранят *Важные Данные* и просто не могут рассматриваться как расходный материал с коротким сроком службы и полной стандартизацией. А как быть со специальными правилами репликации для отчетов? Как насчет различных конфигураций серверов-дублеров?

Устранение барьеров между разработкой и эксплуатацией

Инфраструктура, конфигурации, модели данных и сценарии — все это компоненты программного обеспечения. Изучайте жизненный цикл разработки программного обеспечения и участвуйте в нем, как любой инженер-программист — пишите код, тестируйте, выполняйте интеграцию и сборку, тестируйте и развертывайте. Да, лишнее тестирование не повредит.

К такому изменению парадигмы могут быть не готовы те, кто пришел из эксплуатации и написания скриптов. Может возникнуть своего рода несогласованность нагрузки между тем, как инженеры-программисты управляют организацией, и тем, как построены системы и сервисы для нужд этой организации. Компании, занимающиеся созданием программного обеспечения, имеют очень четкие подходы к разработке, тестированию и развертыванию функций и приложений.

В традиционной среде основной процесс проектирования, создания, тестирования и развертывания инфраструктуры и сопутствующих сервисов для производства всегда состоял из разработки программного обеспечения (Software Engineering, SWE), проектирования (инжиниринга) систем (System Engineering, SE) и администрирования БД. Необходимость изменения парадигмы подталкивает к устранению описанной «несогласованности нагрузки», что означает: DBR-инженерам и системным инженерам для выполнения своей работы приходится пользоваться схожими принципами.

Инженеры-программисты должны разбираться в эксплуатации!

От специалистов команды эксплуатации часто требуют: научитесь писать код или идите домой. Мы с этим согласны, но и обратное должно быть верно. Инженеры-программисты, которых не заставляют вникать в процесс эксплуатации, а также изучать принципы и методы работы инфраструктуры, будут создавать нестабильный, неэффективный и потенциально небезопасный код. «Согласование нагрузки» будет достигнуто только в том случае, если все специалисты окажутся за одним столом!

DBR-инженеров можно включить непосредственно в команду разработчиков программного обеспечения, чтобы они могли работать с той же базой кода, анализировать, как код взаимодействует с хранилищами данных, и изменять его для повышения производительности, функциональности и надежности. Это позволяет сократить организационные издержки и на порядок повысить надежность, производительность и скорость по сравнению с использованием традиционной модели. DBR-инженеры должны адаптироваться к новым процессам и инструментам.

Обзор работы по сопровождению и эксплуатации

Одно из основных направлений деятельности инженера по обеспечению надежности БД — работа по сопровождению и эксплуатации. Это основа и строительный материал для проектирования, тестирования, сборки и эксплуатации любой системы с нетривиальными требованиями к масштабированию и надежности. Это означает, что если вы хотите стать инженером БД, то вам нужно разбираться в этих вещах.

На макроуровне работа по сопровождению не имеет особого значения. Команды эксплуатации воплощают совокупность всех навыков, знаний и установок, которые компания выстроила вокруг практики доставки и управления качеством систем и программного обеспечения. Это неявные и явные ценностные установки, привычки, коллективные знания и системы поощрений. На результат влияют все: от специалистов из отдела технической поддержки до специалистов по производству и генерального директора.

Слишком часто это не очень хорошо налажено. Во многих компаниях культура организации процесса эксплуатации просто ужасна. В результате за работой по сопровождению закрепляется отпугивающая многих плохая репутация, будь то эксплуатация системы, базы данных или сети. Несмотря на это, культура процесса эксплуатации является одним из основных показателей того, как организация решает свою техническую задачу. Поэтому, если вы скажете, что ваша компания не занимается сопровождением, мы просто не станем с ней связываться.

Возможно, вы разработчик программного обеспечения или сторонник инфраструктуры и платформ как сервиса? Или сомневаетесь, что бесстрашному инженеру базы данных так уж нужны знания о ее эксплуатации? Не стоит надеяться на то, что бессерверные вычислительные модели освободят инженеров-программистов от необходимости думать или заботиться о сопровождении. На самом деле все наоборот. Это дивный новый мир, в котором нет специальных команд эксплуатации, а разработкой соответствующей поддержки для вас занимаются инженеры Google SRE и AWS, а также PagerDuty, DataDog и др. Это мир, в котором специалисты по внедрению должны намного лучше разбираться в работе по сопровождению и эксплуатации, архитектуре и производительности, чем сейчас.

Иерархия потребностей

Одна часть наших читателей пришла к этой книге, имея опыт работы на предприятиях, а другая — в стартапах. Приступая к изучению системы, стоит подумать о том, что бы вы сделали в первый же день, взяв на себя ответственность за эксплуатацию ее БД. *Есть ли у вас резервные копии? Они работают? Вы в этом уверены? Есть ли реплика, к которой можно откатиться в случае сбоя? Вы знаете, как это сделать? Подключена ли она к тому же кабелю питания или маршрутизатору, размещена ли на том же оборудовании и находится ли в той же зоне доступности, что и основная база данных? Если резервные копии станут каким-либо образом непригодными, узнаете ли вы об этом? Как именно?*

Другими словами, нужно поговорить об иерархии потребностей базы данных.

Для людей существует иерархия потребностей Маслоу — пирамида желаний, которые должны быть удовлетворены, чтобы мы процветали. К ним относятся физиологическое выживание, безопасность, любовь и принадлежность, уважение и самоактуализация. В основе пирамиды лежат базовые потребности, такие как выживание. После удовлетворения каждого уровня можно переходить на следующий: от выживания — к безопасности, от безопасности — к любви и принадлежности и т. д. Удовлетворив желания первых четырех уровней, мы достигаем самоактуализации и теперь можем безопасно исследовать, творить и добиваться наиболее полного выражения своего уникального потенциала. Такова иерархия потребностей для людей. Используем это как метафору для того, что нужно базам данных.

Выживание и безопасность

Наиболее важные потребности вашей базы данных — это резервное копирование, репликация и восстановление после отказа. *У вас есть база данных? Она действующая? Вы можете ее пропинговать? Ваше приложение отвечает? У нее есть резервная копия? Работает ли механизм восстановления? Если вдруг это окажется не так, как вы об этом узнаете?*

Ваши данные в безопасности? Существует ли несколько рабочих копий данных? Знаете ли вы, как выполнить восстановление после сбоя? Распределены ли копии данных по нескольким зонам физической доступности, нескольким кабелям питания или стойкам? Обладают ли резервные копии целостностью? Сможете ли вы восстановить базу по состоянию на определенный момент времени? Если ваши данные окажутся повреждены, узнаете ли вы об этом? Каким образом? Приготовьтесь гораздо глубже изучить эти вопросы, когда дойдете до главы, посвященной резервному копированию и восстановлению данных.

Одновременно следует позаботиться о масштабировании. Преждевременное масштабирование — это глупо, но вам необходимо подумать о *сегментировании*, росте

и масштабировании уже сейчас, когда вы будете определять идентификаторы для ключевых объектов данных, систем хранения и архитектуры.

ПАТТЕРНЫ МАСШТАБИРОВАНИЯ

Мы будем довольно часто говорить о масштабировании. Масштабируемость — это способность системы или сервиса справляться с растущими объемами работы. Это может быть *реальная* способность, когда то, что было развернуто, поддерживает возможность роста, или же *потенциальная* способность в виде доступных строительных блоков, позволяющих добавлять компоненты и ресурсы, необходимые для масштабирования. В общем, принято выделять четыре основных пути масштабирования.

- Масштабирование по вертикали посредством распределения ресурсов, иначе называемое *вертикальным масштабированием*.
- Масштабирование по горизонтали путем добавления систем или сервисов, иначе называемое *горизонтальным масштабированием*.
- Разделение рабочей нагрузки на более мелкие функции, чтобы масштабировать каждую из них по отдельности, также называемое *декомпозицией функций*.
- Деление некоторой рабочей нагрузки на несколько «разделов», которые одинаковы, не считая различных наборов обрабатываемых данных, также называемое *сегментированием*.

Более подробно эти паттерны будут рассмотрены в главе 5.

Любовь и принадлежность

Любовь и принадлежность — это превращение данных в главный объект процессов разработки программного обеспечения. Речь идет о разрушении границ между базами данных и остальными системами. Границы носят как технический, так и культурный характер, поэтому все это можно назвать просто потребностью интеграции разработки и эксплуатации. На верхнем уровне это означает, что управление базами данных должно выглядеть и быть (насколько это возможно) таким же, как и управление остальными системами. Это также означает, что вы поощряете культуру плавных переходов и кросс-функциональности. На этапе обеспечения любви и принадлежности вы постепенно прекращаете заходить в эту систему и выполнять там обязанности пастуха от имени пользователя root.

Именно в этот момент вы начинаете использовать те же практики проверки и развертывания кода, что и в остальном программном обеспечении. Построение инфраструктуры базы данных и подготовка ее к работе должны быть частью процесса, реализуемого и для всех остальных архитектурных компонентов. Работа с данными должна соответствовать всем остальным частям приложения, чтобы каждый

сотрудник компании чувствовал, что он может взаимодействовать со средой базы данных и поддерживать ее.

Не поддавайтесь желанию заставить своих разработчиков бояться вас. Это довольно легко сделать и довольно заманчиво, ведь вы наверняка считаете, что будете чувствовать себя увереннее, если станете все контролировать. Но это не так. Всем будет гораздо лучше, если вы направите эту энергию на построение ограничителей, чтобы было максимально сложно случайно уничтожить данные. Обучайте людей, дайте каждому возможность отвечать за изменения, которые он вносит. Даже не упоминайте о предотвращении сбоев — это невозможно. Другими словами, создавайте отказоустойчивые системы и поощряйте всех максимально работать с хранилищем данных.

ОГРАНИЧЕНИЯ В ETSY

Компания Etsy представила инструмент *Schemanator* для безопасного внесения в БД изменений, иначе называемых *наборами изменений*, в свои производственные среды. Был установлен ряд ограничений и проверок, чтобы применять эти изменения могли сами разработчики программного обеспечения.

- В определение схем включен эвристический анализ наборов изменений для проверки соответствия стандартам.
- Выполняется тестирование наборов изменений для проверки успешности выполнения сценариев.
- Проводится предстартовая проверка, чтобы показать инженеру текущее состояние кластера.
- Развертывание изменений выполняется так, чтобы серьезным воздействиям подвергались только неактивные базы.
- Процесс установки разбивается на подзадачи так, чтобы можно было их отменить в случае возникновения непредсказуемых проблем.

Подробнее об этом читайте в блоге Etsy (http://bit.ly/2zy74uz).

Уважение

Уважение — одна из высочайших потребностей в пирамиде. Для людей это означает признание со стороны других, а для БД — пригодность для наблюдения и отладки, способность самодиагностики и наличие соответствующего инструментария. Речь идет о возможности понять ваши системы хранения данных, а также сопоставлять события в стеке. На данном этапе есть два аспекта: один касается того, как на текущий момент развиваются ваши производственные сервисы, а другой — влияния человеческого фактора.

Сервисы должны сообщать вам о том, что они включились/отключились или в их работе возникли ошибки. Вы не должны смотреть на графики, чтобы узнать об

этом. По мере развития сервисов темп внесения изменений несколько замедляется и траектория становится более предсказуемой. Работая в реальной производственной среде (среде промышленной эксплуатации), вы каждый день будете все больше узнавать о слабых сторонах системы хранения, об особенностях ее поведения и условиях отказа. Для инфраструктуры данных это время подобно подростковому периоду: больше всего сейчас нужно видение происходящего. Чем сложнее продукт, тем больше у него «шестеренок» и тем больше циклов проектирования нужно выделить для разработки инструментов, необходимых для выяснения того, что происходит.

Вам также нужны средства настройки. Нужна возможность снижать качество обслуживания выборочно, а не полностью, например:

- флаги, с помощью которых можно перевести сайт в режим «только для чтения»;
- отключение отдельных функций;
- постановка операций записи в очередь с отложенным применением;
- возможность занести в черный список отдельные «плохие» акторы или клиентские рабочие места.

Потребности работников компании похожи, но не полностью совпадают. Как правило, вскоре после ввода системы в эксплуатацию команды начинают реагировать слишком активно. Им не будет хватать общего видения ситуации, и они будут это компенсировать, отслеживая все и вся и слишком часто оповещая друг друга о разных событиях. Легко перейти от полного отсутствия графиков к буквально сотням тысяч графиков, 99 % которых совершенно бесполезны. Это не лучший, а, возможно, даже худший вариант. Если система генерирует так много шума, что ваши люди не способны найти верный сигнал и вынуждены постоянно отслеживать файлы журналов и пытаться угадать, что случилось, то это ничуть не лучше или даже хуже, чем полное отсутствие графиков.

Именно в этот момент можно довести людей до выгорания, постоянно прерывая их и приучая не обращать внимания и не реагировать на поступающие предупреждения. Если вы хотите, чтобы все работали по запросу, вам нужно еще на ранних стадиях написать документацию. При запуске системы и активации вызовов вы выталкиваете людей за пределы их зоны комфорта — помогите им. Напишите минимальную эффективную документацию и перечислите необходимые процедуры.

Самоактуализация

Как уникальны лучшие черты личности каждого человека, так же уникален и уровень самоактуализации хранилища данных в каждой организации. Идеал системы хранения данных для Facebook не похож на идеальную систему для Pinterest или GitHub, не говоря уже о крошечном стартапе. Но точно так же, как существуют модели поведения для здоровых самореализованных людей (они не устраивают истерики в продуктовом магазине, правильно питаются и занимаются спортом),

существуют типичные признаки того, что систему хранения данных можно рассматривать как здоровую и самореализованную.

В данном контексте под самореализацией понимается то, что инфраструктура данных помогает достичь поставленной цели, и рабочие процессы в базе данных не препятствуют развитию. Напротив, они помогают разработчикам выполнять свою работу и избежать ненужных ошибок. Типичные эксплуатационные проблемы и рутинные сбои должны исправляться сами, а система должна поддерживаться в работоспособном состоянии без помощи людей. Это будет означать, что у вас есть история масштабирования, которая работает на вас. Неважно, стоит ли за этим десятикратный рост каждые несколько месяцев или просто устойчивость, стабильность и надежность в течение трех лет, прежде чем вам придется начать беспокоиться о емкости. Можете считать, что у вас развитая инфраструктура данных, если большую часть времени вы занимаетесь другими делами. Например, думаете о том, как создавать новые продукты, или пытаетесь спрогнозировать будущие проблемы, а не только реагировать на текущие.

Это нормально — время от времени колебаться вперед и назад между этими уровнями. Эти уровни служат главным образом той основой, которая помогает думать об относительных приоритетах, например о том, что наличие рабочих резервных копий *гораздо важнее*, чем написание сценария для динамического повторного разделения и увеличения емкости. Но если вы все еще находитесь на той стадии, когда имеется только одна копия данных в Сети, или не знаете, что делать в случае сбоя, если главная база отключится, то вам, вероятно, следует прекратить всю прочую деятельность и сначала решить эти вопросы.

Резюме

Переход от других, уже давно известных ролей к DBRE — это смена парадигмы и в первую очередь новый подход к функциям управления хранилищами данных в постоянно меняющемся мире. В следующей главе мы начнем подробно изучать эти функции, отдавая приоритет задачам обслуживания в силу их важности в повседневной разработке БД. А теперь смело иди вперед, отважный инженер!

2 Управление уровнем качества обслуживания

Одним из первых шагов, необходимых для успешного проектирования, создания и развертывания сервиса, является понимание того, чего следует от него ждать. В этой главе мы дадим определение того, что такое управление уровнем качества обслуживания, и рассмотрим его компоненты. Затем обсудим, как определить ожидаемые от сервиса показатели и как проводить мониторинг и составлять отчеты, чтобы убедиться, что эти ожидания оправдываются. В этой главе также создадим продуманный набор требований к качеству обслуживания, чтобы объяснить этот процесс.

Зачем нужны целевые показатели качества обслуживания

Сервисы, которые мы проектируем и создаем, должны удовлетворять набору требований к характеристикам в процессе функционирования. Это часто называют *соглашением о гарантированном уровне качества обслуживания* (Service-Level Agreement, SLA). Однако SLA — это нечто большее, чем просто перечень требований. SLA включают в себя средства возмещения ущерба, способы воздействия и многое другое, что выходит за рамки этой книги. Итак, мы сосредоточимся на понятии *«целевой уровень качества обслуживания»*, или *«целевой показатель качества обслуживания»* (Service-Level Objective, SLO). SLO — это обязательства архитекторов и операторов, которые определяют структуру и функционирование системы для выполнения этих обязательств.

Управлять качеством обслуживания сложно! Сократив эту тему до одной главы, мы многое упростили, но важно понимать нюансы. Приведем несколько примеров, чтобы проиллюстрировать, почему эта проблема так сложна.

- Вам может показаться, что достаточно сообщить о проценте запросов, которые были успешно обработаны разработанным вами API. Хорошо... но кто должен это сообщать? Сам API? Очевидно, это проблема: что случится, если

балансировщики нагрузки откажут? А что произойдет, если система вернет ошибку 200 из базы данных, поскольку ваша система обнаружения сервисов обнаружила недоступность конкретной базы?

- Или, предположим, вы решите: «Хорошо, будем использовать стороннюю систему сквозной проверки и подсчитаем, сколько запросов считывают и записывают корректные данные?» Сквозные проверки — отличная вещь, это лучшее средство оповещения о проблемах надежности. Но проверяют ли они *всю* серверную часть?
- Включаете ли вы в свои SLO менее важные сервисы? Ваши клиенты, вероятно, предпочли бы получить уровень доступности API 99,95 % и уровень доступности продукта пакетной обработки 97 вместо 99,8 % и для API, и для пакетной обработки.
- Насколько сильно вы контролируете клиентов? Если доступность вашего API составляет 98 %, но мобильные клиенты автоматически повторяют попытки, что обеспечивает надежный ответ в 99,99 % случаев в течение трех попыток, то они могут этого никогда не заметить. Каково точное число?
- Возможно, вы скажете: «Я просто посчитаю процент ошибок». Но как быть с ошибками, вызванными тем, что пользователи отправляют недействительные или неправильно сформированные запросы? Вы ничего не можете с этим поделать.
- Может быть, вы получаете правильный результат в 99,999 % случаев, но в 15 % случаев задержка превышает 5 секунд. Это приемлемо? В зависимости от поведения клиента это может означать, что ваш сайт не отвечает на запросы некоторых людей. С технической точки зрения вы можете получить эти пять девяток, но на самом деле пользователи могут быть ужасно — и справедливо — недовольны.
- Что делать, если сайт доступен на 99,999 % для 98 % пользователей, но только на 30–70 % для остальных 2 % пользователей? Как это вычислить и оценить?
- Что делать, если не работает или работает медленно только один фрагмент или одна серверная часть? Что, если вследствие ошибки при обновлении были потеряны 2 % данных? Что, если вы потеряли данные за целый день, но только для определенных таблиц? Что, если из-за характера этих данных пользователи никогда не замечали их потерю, но вы сообщили о потере 2 % данных, чем вызвали у всех тревогу и побудили их мигрировать с вашего ресурса? Что, если эти 2 % потерянных данных фактически состояли из перезаписанных указателей на активы, так что, хотя на самом деле данные не были утеряны, их нельзя было найти?
- Что, если для некоторых пользователей доступность составляет 95 %, потому что у них плохой Wi-Fi, старый кабельный Интернет или плохие таблицы маршрутизаторов между их клиентом и вашим сервером? Могут ли они привлечь вас к ответственности?

- Что, если так происходит с целыми странами? Что ж, тогда, вероятно, за это вас могут обвинить (например, перегрузку UDP-пакетами DNS у некоторых провайдеров вы можете исправить).
- Что, если ваш показатель равен 99,97 %, но каждая ошибка приводит к тому, что весь сайт перестает загружаться? Что делать, если доступность составляет 99,92 %, но каждая страница содержит 1500 компонентов и пользователи почти никогда не замечают невозможность загрузить крошечный виджет? Какой из этих вариантов лучше?
- Что лучше, подсчитывать частоту ошибок в данный момент или по временным интервалам? Или число интервалов (минут или секунд), когда количество ошибок или задержек превышало порог?

Пять девяток?

Многие используют число 9 для краткого описания доступности. Например, если система спроектирована с доступностью пять девяток, это означает, что она рассчитана быть доступной 99,999 % времени, тогда как три девятки будут означать 99,9 %.

Вот почему практика проектирования, настройки и адаптации SLO и показателей доступности с течением времени является проблемой не столько сферы вычислений, сколько социальных наук. Как рассчитать процент доступности, который точно отражает опыт ваших пользователей, повышает доверие и правильно стимулирует их поведение?

С точки зрения вашей команды, какие бы показатели доступности вы ни сочли важными для доставки пользователям, эти цифры будут всячески обыгрываться, хотя бы подсознательно. Именно на эти цифры вы будете обращать внимание, определяя, становится ли система более или менее надежной и не нужно ли переключить ресурсы с разработки функций на надежность или наоборот.

С точки зрения потребителей, самое важное в показателе — то, чтобы он максимально *отражал их опыт*. Если у вас есть возможность вычислять показатели для каждого клиента или среза и разбивать данные на произвольные интервалы измерений — даже с большим количеством элементов, таких как UUID, — это невероятно эффективно. Именно так сделано в Scuba для Facebook и в Honeycomb.io.

Показатели уровня обслуживания

При оценке требований к SLO мы обычно учитываем конечный набор *показателей* (*индикаторов*, *метрик*), требования к которым и будем формулировать. Для этого рассмотрим идеальные и рабочие параметры. SLO можно представить как множество, состоящее из одного или нескольких показателей, которые определяют

ожидания от сервиса, обычно потому, что эти показатели объективно связаны с ожиданиями.

Например, задержка отклика свыше определенного порога может превратиться в проблему доступности, потому что система фактически станет непригодной для использования. Задержка в отрыве от пропускной способности легко может ввести в заблуждение, она не обязательно адекватно отражает загрузку системы. В следующих разделах перечисляются и разъясняются типичные показатели.

Задержка

Задержка, также известная как время отклика, является количественной характеристикой времени, требуемого для получения результата запроса. Лучше не разбивать задержку на компоненты, а измерять ее целиком, от запроса до ответа на стороне клиента. Это клиентоориентированный параметр, который имеет ключевое значение для любой системы, имеющей клиентов, то есть для любой системы!

Задержка или время отклика?

Тонны чернил и крови были пролиты в дебатах на тему «задержки» и «времени отклика». Есть мнение, что задержка — это время, требующееся для получения услуги, тогда как время отклика — это время, необходимое для выполнения запроса. В этой книге мы используем понятие «задержка» для обозначения общего времени прохождения запроса от его генерации до получения полезного результата.

Доступность

Обычно доступность выражается в процентах от общего времени, когда система должна быть доступной. Доступность определяется как способность вернуть ожидаемый ответ запрашивающему клиенту. Обратите внимание: здесь не учитывается время, поэтому большинство SLO включают в себя и время отклика, и доступность. После определенной величины задержки систему можно считать недоступной, даже если запрос все-таки выполняется. Доступность часто выражается в процентах, например 99,9 % для определенного временного окна, в течение которого проводятся измерения. Для этого результаты всех замеров, сделанных в данном окне, должны быть собраны в единый показатель.

Пропускная способность

Другим распространенным SLI является пропускная способность, или количество успешных запросов за определенный период времени, обычно измеряемая в еди-

ницах в секунду. Пропускная способность весьма полезна в качестве параметра, дополняющего задержку. Когда команда готовится к запуску системы и измеряет задержку, она должна делать это для максимальной пропускной способности, в противном случае тесты будут бесполезны. Задержка имеет тенденцию оставаться стабильной до определенного переломного момента, и мы должны знать, когда он наступит, определяя целевой уровень пропускной способности.

Надежность хранения

Надежность хранения касается систем хранения и хранилищ данных. Этот параметр характеризует, насколько качественно сохраняются данные после операции записи в хранилище и насколько они впоследствии доступны для извлечения. Надежность хранения может быть выражена как временное окно, например: в случае сбоя системы могут быть потеряны данные, записанные в течение не более чем последних 2 секунд.

Стоимость — эффективность

Показатель «стоимость — эффективность» часто игнорируется или не учитывается при обсуждении уровня обслуживания. Вместо этого обнаруживается, что он отнесен к бюджету проекта и зачастую не отслеживается как следует. Так или иначе, общая стоимость услуг для большинства предприятий является критическим компонентом. В идеале это должно выражаться в стоимости действия, например просмотра страницы, подписки или покупки.

Организация должна ожидать выполнения следующих действий в рамках операций своих сервисов.

- *Новый сервис*. Цели SLO определены. В более традиционных моделях это можно было бы назвать соглашениями на операционном уровне.
- *Новые SLO*. Установка соответствующего мониторинга для оценки фактических и желаемых показателей.
- *Существующий сервис*. Необходимо запланировать регулярные проверки SLO для подтверждения того, что для определенных SLO принимается во внимание текущая критичность обслуживания.
- *Выполнение SLO*. Регулярные отчеты для указания исторического и текущего состояния выполнения или нарушения SLO.
- *Проблемы с обслуживанием*. Портфель проблем, которые повлияли на уровни обслуживания и их текущее состояние с точки зрения обходных путей и исправлений.

Определение целей обслуживания

SLO должны быть построены из того же набора требований, к которому относятся функции продукта. Мы называем это структурой, ориентированной на клиента, потому что должны определять требования, основываясь на потребностях наших клиентов. Обычно мы вводим не более трех показателей. Большее их количество редко дает что-нибудь значимое. Лишние дополнительные показатели часто содержат сведения, вторичные по отношению к основным (первичным).

Показатели задержки

SLO задержки может быть выражен как диапазон значений соответствующего показателя. Например, мы могли бы сказать, что задержка до получения результата запроса должна быть меньше 100 миллисекунд (что на самом деле обозначает диапазон от 100 до 0 миллисекунд, если определять обе границы явно). Величина задержки исключительно важна при взаимодействии с пользователем.

Почему так важна задержка

Медленно или неустойчиво работающие сервисы способны потерять больше клиентов, чем система, которая не работает вообще. Скорость действительно важна настолько, что, по исследованиям Google Research, введение задержки от 100 до 400 миллисекунд приводит к сокращению поисковых запросов на 0,2–0,6 % в течение 4–6 недель. Более подробную информацию вы найдете в Speed Matters по адресу http://googleresearch.blogspot.com/2009/06/speed-matters.html. Вот еще несколько впечатляющих цифр:

- Amazon: каждые дополнительные 100 миллисекунд задержки сокращают продажи на 1 %;
- Google: увеличение времени загрузки страницы на 500 миллисекунд приводит к сокращению количества поисковых запросов на 25 %;
- Facebook: замедление загрузки страницы на 500 миллисекунд приводит к уменьшению трафика на 3 %;
- задержка отклика на 1 секунду снижает удовлетворенность клиентов на 16 %.

SLO доступности можно описать так: *задержка выполнения запроса не должна превышать 100 миллисекунд*.

Если оставить нижнюю границу равной 0, это может привести к определенным проблемам. Специалист по производительности может потратить неделю на то, чтобы уменьшить время отклика до 10 миллисекунд, но мобильные устройства, применяющие приложение, обычно не будут иметь доступа к достаточно быстрой

сети, способной воспользоваться такой оптимизацией. Другими словами, ваш специалист потратит неделю работы впустую. Мы можем доработать формулировку SLO следующим образом: *задержка выполнения запроса должна составлять от 25 до 100 миллисекунд*.

Теперь поговорим о том, как собирают эти данные. Если это делается путем анализа журналов, то можно взять запросы за 1 минуту и усреднить их. Однако здесь есть проблема. Большинство сетевых систем показывают распределение случайных величин задержки с небольшим процентом существенных «отрывов». Это и искажает среднее значение, и маскирует характеристики полной рабочей нагрузки от инженеров, наблюдающих за системой. Другими словами, агрегирование времени отклика — это *преобразование с потерями*.

На самом деле нужно представлять задержку как распределение величин задержек. Задержки почти никогда не распределены по закону Гаусса или Пуассона, поэтому средние значения, медианы и среднеквадратичные отклонения в лучшем случае бесполезны, а в худшем — ложны (http://bravenewgeek.com/everything-you-know-about-latency-is-wrong/).

Чтобы лучше понять сказанное, взгляните на рис. 2.1 и 2.2, предоставленные Circonus — продуктом для мониторинга крупных систем. В блоге Circonus эти графики используются для демонстрации *размывания всплесков* — именно это явление мы обсуждаем. На рис. 2.1 мы построили график средних значений с большим временным окном для каждого усреднения, чтобы собрать данные за месяц.

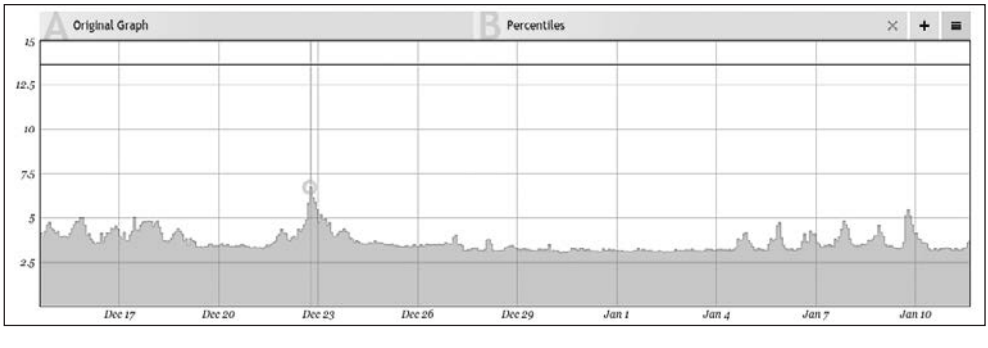


Рис. 2.1. Средние значения задержки с большим окном усреднения

На рис. 2.2 выполняется усреднение по намного более узким временным окнам, поскольку показаны данные только за 4 часа.

Несмотря на то что набор данных один и тот же, средние значения на рис. 2.1 показывают пик со значением примерно 9,3, а на рис. 2.2 — 14!

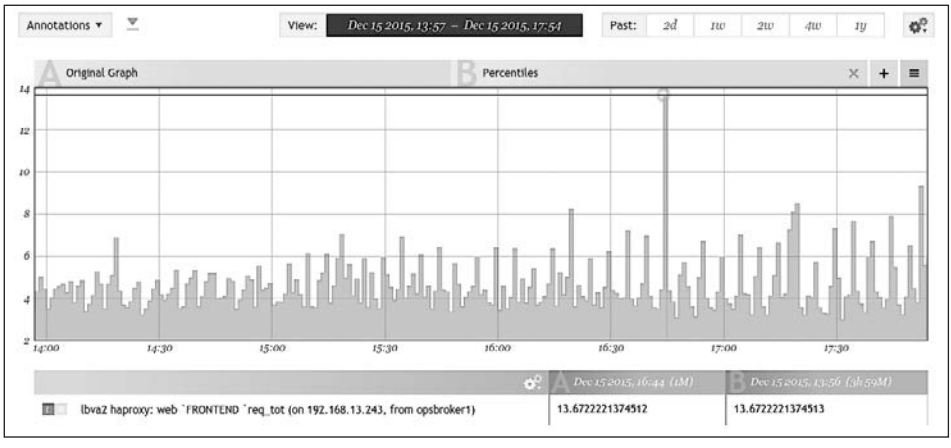


Рис. 2.2. Средняя задержка с меньшим временным интервалом усреднения

Будьте внимательны, сохраняя средние значения!

Не забывайте сохранять фактические, а не только усредненные значения! Если у вас есть приложение для мониторинга, которое усредняет значения каждую минуту и не сохраняет реальную полную историю значений, то однажды вам обязательно понадобится получить среднее значение за 5 минут, используя средние значения за 1 минуту. Вы получите абсолютно неверные данные, потому что первоначальное усреднение было выполнено с потерями!

Если вы рассматриваете данные, полученные за 1 минуту, как полный набор данных, а не как среднее значение, то, возможно, захотите визуализировать влияние выбросов (на самом деле вас могут интересовать именно выбросы). Это можно сделать несколькими способами. Во-первых, можно визуализировать минимальное и максимальное значения относительно среднего. Можно также отделить выбросы, отфильтровав для усреднения задержки по определенному критерию, например только наиболее быстрые 99,9, 99 и 95 % запросов. Если совместить эти три вычисленных значения с результатами усреднения по полной выборке (рис. 2.3) и найти минимум/максимум, то получим очень хорошее представление о выбросах.

Теперь, продвигаясь дальше, подумаем о нашем SLO с учетом задержки. Если выполнять усреднение каждую минуту независимо от того, каков SLO, то мы не сможем доказать, что достигли цели, поскольку измеряем только средние значения! Почему бы не сформулировать цель так, чтобы она больше соответствовала реальной рабочей нагрузке? Мы можем изменить SLO следующим образом: *в результате измерений в течение 1 минуты задержка для 99 % запросов должна составлять от 25 до 100 миллисекунд*.

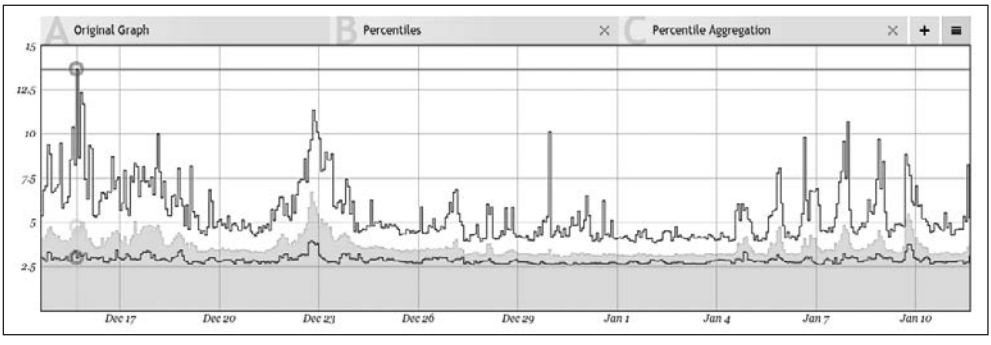


Рис. 2.3. Среднее значение задержки (размер выборки — 100 %), совмещенное с минимумом и максимумом

Почему же мы все-таки выбираем что-то вроде 99 % вместо 100 %? Распределения величины задержки имеют тенденцию быть очень мультимодальными. Есть нормальные периоды, а есть крайние случаи, вызванные разнообразными случайностями, встречающимися в сложной распределенной системе, такими как сборка мусора в Java Virtual Machine (JVM), очистка базы данных, очистка кэша и многое другое. Таким образом, мы ожидаем определенного процента выбросов и наша цель при формулировании SLO состоит в том, чтобы определить тот процент выбросов, который мы готовы терпеть.

Теперь рассмотрим нагрузку. Мы говорим просто о единичном результате, который, например, получаем из вызова API? Или оцениваем визуализацию страницы, которая представляет собой совокупность многих вызовов, выполняемых в течение некоторого времени? Если измерять время визуализации страницы, то нам могут понадобиться как два отдельных показателя время до получения первого отклика и время до полного ее завершения, поскольку этот процесс может оказаться довольно длительным.

Показатели доступности

Как уже упоминалось, доступность — это доля времени, обычно выражаемая в процентах, в течение которого сервис может отвечать на запросы должным образом. Например, мы можем потребовать, чтобы система была доступна 99,9 % времени. Это может звучать так: сервис должен быть доступен 99,9 % времени.

Это дает нам около 526 минут простоя для работы в течение года. Почти 9 часов! Это какой-то праздник простоя. Вы можете спросить, почему бы просто не потребовать 100 %? Если вы владелец продукта или продавец, то, вероятно, так и сделаете. Считается, что повышение требований с 99 до 99,9 и далее до 99,99 % каждый раз на порядок увеличивает сложность управления, стоимость системы и трудоемкость

ее сопровождения. Кроме того, если это приложение, взаимодействующее через Интернет или географически удаленное, то можно ожидать, что транспортные среды будут вносить собственные сбои, которые не дадут воспользоваться более чем 99,0–99,9 % времени безотказной работы вашей системы.

Нужно также учесть, что разница между 526 отключениями на 1 минуту за год и одним 526-минутным отключением очень велика. Чем короче время простоя, тем вероятнее, что большинство пользователей даже не заметят сбоя. Напротив, восьмичасовой перерыв в работе некоторых служб приводит к появлению новостных статей, тысяч твитов и подрывает доверие пользователей. Имеет смысл рассматривать доступность вашего сервиса сразу в двух аспектах. Первый из них — это среднее время между отказами (Mean Time Between Failures, MTBF). Предотвращение сбоев всегда было приоритетом, то есть чем больше MTBF, тем лучше. Второй аспект — это среднее время восстановления (Mean Time To Recover, MTTR). Это время, необходимое для возобновления обслуживания после сбоя. Чем оно меньше, тем лучше!

Доступность: безотказность или отказоустойчивость?

В последнее десятилетие было много дискуссий о построении устойчивых систем, которые имеют три специфических свойства:

- низкое MTTR вследствие автоматизированного восстановления для хорошо контролируемых сценариев отказов;
- ограниченное влияние сбоев благодаря распределенности и избыточности;
- возможность рассматривать сбой как обычный сценарий в системе, для чего необходимо, чтобы процедуры автоматического и ручного восстановления были хорошо документированы, тщательно спроектированы, проверены на практике и интегрированы в повседневную работу.

Обратите внимание: здесь не делается акцент на устранении сбоев. Системы без сбоев, несмотря на надежность, становятся нестабильными и непрочными. При возникновении сбоя в таких системах команды, которые должны реагировать на инцидент, скорее всего, окажутся не готовы, и это может значительно усугубить последствия. Кроме того, имея дело с надежной, но уязвимой системой, пользователи могут ожидать от нее большей надежности, чем та, которая указана в SLO и на которую был рассчитан сервис. Это означает, что в случае сбоя, даже если условия SLO не нарушаются, клиенты могут быть очень расстроены.

Вооружившись этими знаниями и оценивая доступность SLO, необходимо задать себе несколько ключевых вопросов.

- Есть ли обходные пути решения проблем во время простоя? Можете ли вы работать в ограниченном режиме, например только чтения? Можно ли использовать закэшированные ранее данные, пусть даже и устаревшие?

- Зависит ли допустимость простоя от того, какой процент пользователей он затрагивает?
- Какие последствия для пользователей имеют простои разной длительности:
  - один неудачный запрос;
  - 30 секунд;
  - 1 минута;
  - 5 минут;
  - 1 час и больше?

После этого можно пересмотреть и заново оценить базовые SLO доступности, определив:

- временной интервал;
- максимальную продолжительность инцидента;
- процентную долю затронутых пользователей, по достижении которой система считается недоступной.

С учетом этого можно определить SLO, например, следующим образом:

- 99,9 % доступности в среднем за неделю;
- ни один инцидент не приводит к недоступности более чем на 10,08 минуты;
- учитываются простои, затронувшие более 5 % пользователей.

Проектирование с учетом допустимого времени простоя

На этом новом уровне мы можем разрабатывать такие процессы, как обработка и преодоление отказов, блокировка базы данных и перезапуск в соответствии с указанными параметрами. Мы можем выполнять непрерывные обновления, которые затрагивают менее 1 % пользователей. И заблокировать таблицы для построения индексов, если это занимает менее 10 минут и если на этой неделе еще не было простоев. Планируя время простоев, а не пытаясь достичь их полного отсутствия, мы можем более эффективно проектировать систему и допускать некоторые риски ради обновления и ускорения.

Стоит отметить, что даже в современном мире, где 99,9 % безотказной работы считается нормой, бывает, что сервисы действительно безболезненно переносят простои, если те ожидаемы и управляемы. Хотелось бы также видеть допустимым четырехчасовой простой, если о нем сообщено заранее, а последствия смягчены ограничением функциональности до режима «только чтение» и распределением их среди небольших групп пользователей, так как это позволяет избежать многочасовых тщательно спланированных миграций данных, связанных с риском повреждения данных, нарушения конфиденциальности и многого другого.

С учетом всего этого вы, возможно, захотите переформулировать SLO доступности, добавив ограничения на запланированное время простоя, чтобы руководить группой эксплуатации в ходе работы по обслуживанию.

Вот пример SLO доступности, версия 2:

- система доступна в среднем 99,9 % времени в течение недели;
- ни один инцидент не приводит к недоступности более чем на 10,08 минуты;
- учитываются только простои, затронувшие более 5 % пользователей;
- допускается один ежегодный четырехчасовой простой при условии, что:
  - об этом было сообщено пользователям как минимум за две недели;
  - он влияет не более чем на 10 % пользователей одновременно.

Показатели пропускной способности

Пропускная способность как показатель качества обслуживания отражает пиковую величину нагрузки, которую сервис должен поддерживать, одновременно обеспечивая заданные SLO для задержки и доступности. Вы можете сказать: «Лейн, Черити, почему мы должны это делать? Разве задержки и доступности не должно быть достаточно?» На что одна из нас ответила бы: «Отличный вопрос, бравый скаут!» А потом задумчиво затянулась бы трубкой...

В системе может присутствовать узкое место, «бутылочное горлышко», которое задает верхнюю границу пропускной способности, не обязательно снижая производительность или доступность. Возможно, в вашей системе есть блокировка, которая ограничивает вас 50 запросами в секунду (qps). Ответы на них могут быть короткими и быстрыми, но если такого ответа ожидает 1000 человек, то это уже проблема. Поскольку измерить сквозную задержку (полное время прохождения запроса) не всегда возможно, показатели пропускной способности часто служат дополнительным уровнем проверки работоспособности системы и ее соответствия потребностям бизнеса.

Как и задержки, показатели пропускной способности подвержены проблемам усреднения и недостаточной «разрешающей способности» выборки. Пожалуйста, помните об этом при мониторинге.

Показатели «стоимость — эффективность»

При поиске действенных показателей стоимости системы самое важное — то, с помощью чего вы будете оценивать затраты. Это, в сущности, решение бизнес-уровня, но вам придется отыскать в сервисе, где получить эти значения. Если вы являетесь поставщиком контента, таким как онлайн-журнал, то решающее значение имеют

страницы, которые нужно доставлять. Если вы поставщик продукта типа «программное обеспечение как услуга» (Software as a Service, SaaS), то имеет смысл выбрать в качестве показателя количество подписок на вашу услугу. Для розничных продавцов подойдет подсчет транзакций.

Что нужно учесть

Почему вам, инженеру БД, нужно все это знать? Вы управляете только одним компонентом сервиса, так почему вы должны заботиться об общих требованиях? В черные дни, когда работа системы нарушена, вашей задачей будет только ваше хранилище данных и значение будет иметь только ваша способность его обслуживать. Но, будучи частью большой команды, вы получаете прекрасные возможности влиять на скорость и доступность всего сервиса.

Зная все SLO, вы можете расставить приоритеты по своему усмотрению. Зная, что SLO задержки составляет 200 миллисекунд, можно предположить, что в это время включены:

- разрешение адресов DNS;
- балансировка нагрузки;
- перенаправление на http-сервер;
- прикладные функции приложения;
- запросы к базе данных;
- транспортировка данных TCP/IP через океаны и континенты;
- чтение данных с твердотельных устройств (SSD) и жестких дисков.

Итак, если хранилища данных функционируют хорошо и их вклад в задержку минимален, то следует сосредоточиться на чем-то другом. Если же вы видите, что SLO в опасности, а сочный фрукт производительности висит достаточно низко, можете потратить немного времени, чтобы сорвать его.

Собирая все SLO-показатели для вашего нового сервиса, являющегося объектом внимания, необходимо учесть несколько дополнительных моментов.

- *Не переборщите.* Мы накапливаем показатели и понимаем почему. Но, пожалуйста, постарайтесь, чтобы список был простым и кратким и текущее состояние показателей можно было просмотреть на одной странице панели инструментов.
- *Ориентируйтесь на пользователя.* Подумайте, что ваши пользователи сочтут критичным, и начните именно с этого. Помните, что для большинства приложений основными показателями сервисов будут задержка, пропускная способность и доступность, а сервисы хранения добавляют к этому списку надежность хранения данных.

- *Определение SLO — итеративный процесс.* Если у вас есть процесс проверки SLO, можете со временем изменять и дополнять показатели. На ранних стадиях, возможно, не стоит быть слишком агрессивными в отношении SLO. Это позволит инженерам сосредоточиться на прикладных функциях и улучшениях.

Используйте SLO, чтобы определить, как вы хотите проектировать сервисы, процессы и инфраструктуру.

Мониторинг SLO и построение отчетов

Теперь, когда у вас есть четко определенные SLO, очень важно следить за тем, что происходит в реальной жизни и как это соотносится с поставленными идеальными целями. В этой книге мы еще не говорили об оперативном контроле, но есть некоторые важные вещи, которые нужно обсудить, прежде чем перейти к следующей теме.

Главная цель мониторинга при управлении уровнем качества обслуживания — упреждающее выявление и устранение любых потенциальных факторов, которые могут привести к нарушению SLO. Другими словами, мониторинг нужен не для того, чтобы узнавать о фактах нарушений *в настоящее время*. Это как сплав по реке на каноэ: нам не нужно знать, что на реке есть пороги, если *мы на них уже попали*. Мы хотим знать, что из происходящего может указывать на то, что пороги будут ниже по течению, *пока мы еще идем по спокойной воде*. Кроме того, мы хотим иметь возможность предпринять соответствующие действия, чтобы гарантировать, что мы остаемся в пределах SLO, на которые настроены сами и настроили свои системы.

При мониторинге мы всегда будем полагаться на автоматический сбор и анализ показателей. Затем результаты этого анализа будут передаваться в программное обеспечение для автоматического принятия решений и исправления, оповещения людей-операторов (другими словами, вас) или формулирования задач для дальнейшей работы. Кроме того, вы захотите визуализировать эти данные для анализа в реальном времени людьми и, возможно, создать панель мониторинга для высокоуровневого отображения текущего состояния. Мы рассмотрим эти три сценария, когда будем обсуждать отслеживаемые показатели.

Другими словами, предположим, у вас есть допустимые 10,08 минуты простоя в течение недели и ко вторнику (то есть в течение трех дней) длительность простоев достигла 3 минут из-за событий Stop the World от сборщика мусора Cassandra Garbage Collection плюс 1 минута после сбоя балансировщика нагрузки. Вы уже использовали 40 % SLO, осталось еще четыре дня. Похоже, пора настроить сборку мусора! Получив предупреждение после достижения определенного порогового значения (30 %), система генерирует электронное письмо о создании нового задания[1] для инженера по обеспечению надежности БД, и тот может немедленно переключиться на решение проблемы.

[1] В принятой терминологии ticket. — *Примеч. науч. ред.*

Мониторинг доступности

Обратимся к показателю SLO доступности, который мы определили в предыдущем разделе. Как его отследить? Чтобы получить адекватную картину, потребуется мониторинг доступности системы, а также ошибок прикладного характера. Напомним, что текущая «демонстрационная» версия SLO выглядит следующим образом:

- 99,9 % доступности в среднем в течение недели;
- длительность любого инцидента не превышает 10,08 минуты;
- учитываются только простои, затронувшие более 5 % пользователей;
- допускается один четырехчасовой простой в год, если:
  - пользователи оповещены по крайней мере за две недели до начала простоя;
  - простой влияет не более чем на 10 % пользователей одновременно.

Исторически сложилось так, что сотрудники отдела эксплуатации, как правило, основное внимание уделяют относительно низкоуровневому мониторингу, который информирует о доступности системы. Например, они могут следить, активен ли и доступен ли конкретный хост, а также активны ли и доступны ли размещенные на нем сервисы. В распределенной системе такой подход быстро становится неприемлемым, поскольку плохо сказывается на прогнозировании доступности сервисов в целом. Если у нас 1000 виртуальных машин Java, 20 экземпляров базы данных и 50 веб-серверов, как узнать, влияет ли каждый из этих компонентов на всю систему и если влияет, то в какой степени?

Исходя из этого, первое, чем следует заняться, — частота ошибок выполнения запросов пользователей. Это называется также *мониторингом реальных пользователей* (Real User Monitoring, RUM). Например, когда пользователь отправляет HTTP-запрос из браузера, получает ли он корректный ответ от сервиса? Если сервис популярен, то таких данных может быть очень много. Например, важное событие в глобальной службе новостей привлекает более 70 000 обращений к веб-сервису за секунду. Подсчитать частоту появления ошибок для такого объема данных вполне по силам любому современному процессору. Все эти данные передаются приложением (например, Apache HTTP-сервером) в демон протоколирования (например, в системный журнал Linux).

Дальнейшие пути извлечения полезных данных системой из этих журналов и передачи их в соответствующие инструменты мониторинга и анализа очень разнообразны. Пока остановимся на этом и будем считать, что мы сохранили сведения об успешной работе сервиса и ошибках в рабочем хранилище данных без какого-либо агрегирования или усреднения. Это уже обсуждалось в предыдущем разделе, но стоит повторить: хранение одних только средних значений приводит к потере ценной информации.

При наличии сохраненных данных можно элементарно определить, превышен ли однопроцентный порог неуспешных запросов, и если да, то пометить эту секунду как время простоя. Регулярный подсчет времени простоя можно суммировать, сравнивать с нашим лимитом 604,8 секунды за неделю и отображать результат на панели мониторинга в браузерах, на мониторах в сетевом операционном центре, в офисе или любом другом месте, чтобы все заинтересованные стороны видели, как работает ваша команда.

В идеале мы хотим иметь возможность использовать эти данные, чтобы предсказать, приведут ли текущие простои к превышению лимита этой недели. Самая большая проблема для большинства систем — это изменение рабочей нагрузки в ходе разработки и модернизации продукта. В системе, для которой выпуски обновлений происходят еженедельно, а иногда и ежедневно, любые предыдущие выборки данных становятся почти бесполезными. Это особенно верно для более старых выборок, по сравнению со свежими данными, и называется *затухающей функцией*.

Подробное знакомство с прогнозирующим анализом данных выходит за рамки этой книги. Но есть множество подходов, которые вы можете использовать уже сейчас, чтобы предсказать, нарушите ли вы свой SLO на текущей неделе или, возможно, в будущем. Для этого стоит взять данные за N предыдущих недель (в стабильных системах N может быть больше, а в условиях непрерывного развертывания обновлений равняться 1) и посмотреть, сколько нарушений SLO произошло за те периоды, для которых время простоя не превышало значение, полученное в текущем периоде, или меньше его.

Например, скрипт вашего анализа получил текущее время простоя, которое составляет, допустим, 10 секунд за неделю, и длительность этой недели в секундах. Мы получим 10 секунд простоя на 369 126 секунд в неделе.

Затем можно рассмотреть предыдущие 13 недель, и для каждой недели, когда время простоя составляло 10 секунд или менее в один и тот же момент недели (от 1 до 369 126 секунд), оценить, произошло ли тогда нарушение SLO. Затем можно присвоить вес на основе близости предыдущего периода. Например, из этих 13 недель последней неделе присваивается вес 13 баллов, предпоследней — 12 и т. д. Суммируя веса за те недели, когда произошли нарушения SLO, вы можете подготовить высокоприоритетное задание для команды операционистов и уведомлять их в чате, если совокупное значение достигнет или превысит 13. Это лишь один пример того, как обеспечить определенный уровень мониторинга данных, если у вас нет команды первоклассных аналитиков, у которой достаточно времени и желания просматривать данные о качестве обслуживания. Здесь целью является предварительное изучение потенциальной проблемы, прежде чем она возникнет во время чрезвычайной ситуации, а это будет означать меньше забот и меньшее снижение доступности.

В дополнение к мониторингу реальных пользователей полезно создать второй набор данных для искусственных тестов. Это называется *синтетическим* мониторин-

гом. То, что тесты являются искусственными, не означает, что они не идентичны по активности реальному пользователю. Например, компания транзакционной электронной почты может инициировать запросы электронной почты от учетных записей QA, как это делал бы любой другой клиент.

Тестовые наборы и воздействия для синтетического мониторинга должны обеспечивать полное и всестороннее покрытие. Пользователи могут приходить из разных регионов и быть активными в разное время. Это может привести к появлению слепых зон, если не отслеживать все возможные регионы и пути выполнения кода в нашем сервисе. С помощью синтетического мониторинга можно определить те места, где доступность или задержка ухудшились или нестабильны, и принять соответствующие меры для предотвращения или смягчения проблем. Примерами таких мер могут быть добавление емкости хранилищ, запросы на настройку производительности и даже отвод трафика из нестабильной зоны.

С помощью синтетических тестов и RUM можно обнаружить снижение доступности системы и даже предсказать, когда может произойти нарушение SLO. Но это не поможет, если речь идет о более серьезных воздействиях, вызванных сбоями системы или нехваткой ресурсов. Одна из важнейших целей, преследуемых при построении надежного мониторинга, — обеспечить сбор достаточного количества данных для прогнозирования сбоев и перегрузок до их возникновения.

Мониторинг задержки

Мониторинг задержки очень похож на мониторинг ошибок запросов. Однако если доступность выражается логическим значением («да» или «нет»), то задержка — это значение времени, количественная величина, которую нужно измерить, чтобы оценить, соответствует ли она указанным в SLO ограничениям.

SLO для задержки

Время ожидания для 99 % запросов в течение 1 минуты должно составлять 25–100 миллисекунд.

Как и в случае с телеметрией ошибок, мы предполагаем, что параметры HTTP-запросов были направлены в системный журнал и записаны в хранилище данных как последовательность событий. Теперь мы можем взять временной интервал, упорядочить все отсчеты по времени ожидания и исключить 1 % самых длительных запросов. Затем мы усредняем значения в каждом односекундном временном окне. Если для какого-либо из оставшихся 99 % запросов ожидание превысило 100 миллисекунд, это означает нарушение SLO и это временное окно учитывается как время простоя.

Данные такого типа можно использовать также для прогнозирующей аналитики с помощью любого набора инструментов или скриптов. Измеряя предыдущие задержки в течение аналогичного времени или для аналогичных типов трафика, можно выявлять аномалии, указывающие на рост времени отклика, который может привести к нарушению SLO.

Мониторинг пропускной способности

Пропускную способность легко оценить с помощью тех же данных, которые собираются и анализируются для SLO доступности и задержек. Если вы сохраняете все записи, то сможете легко измерить количество транзакций в секунду. Если количество обрабатываемых транзакций превосходит минимальное, указанное в SLO, то все в порядке. Если система не генерирует достаточно трафика, чтобы достичь уровня SLO, то остается полагаться на периодические нагрузочные тесты: так можно убедиться, что система способна выполнить требования SLO. Нагрузочное тестирование будет более подробно описано позже.

Мониторинг стоимости и эффективности

Оценка выполнения SLO стоимости и эффективности может стать сложной задачей для мониторинга, потому что не все затраты можно легко описать количественно. Приходится учитывать суммарную стоимость за промежуток времени. Если вы работаете в облачной среде, в которой выставляются счета за арендуемые ресурсы, подобно счетам за коммунальные услуги, то можно достаточно легко оценить расходы на хранилище данных, процессор для вычислений, память и пропускную способность между вычислительными узлами. Если же вы используете собственные серверы, то вам нужно вычислить затраты на оборудование для всех машин, выделенных для сервисов, с отдельной оценкой расходов в тех случаях, когда задействуются общие ресурсы. Цена обычно рассчитывается для достаточно длительного непрерывного отрезка времени, поэтому может оказаться сложно вычислить затраты за определенные периоды времени, например за час, имея только ежемесячные отчеты от поставщика услуг.

Для ресурсов с фиксированной стоимостью, таких как вычислительные узлы и хранилища, вы можете иметь матрицу затрат, полученную от поставщика услуг или из собственных внутренних БД. На эти данные можно опираться для оценки предполагаемых затрат при развертывании ресурсов и выведении их из эксплуатации. Для оценки затрат, зависящих от интенсивности использования, таких как пропускная способность, IOPS и т. п., можно ссылаться на другие показатели, собранные в течение заданного временного интервала.

Вам также необходимо иметь в виду расходы на персонал, который обслуживает данный сервис. Сюда могут входить инженеры по эксплуатации, базам данных

и сетям, все дежурные и менеджеры, координирующие проект. Эти люди могут работать и на другие проекты, поэтому вам снова нужно оценить процент времени, посвященного вашему сервису. Если в вашей компании действует мониторинг рабочего времени, можете использовать эти данные, чтобы построить некие оценки затрат человеческих ресурсов в реальном времени. В противном случае придется использовать принятые нормы времени, учитывая такие факторы, как увольнения, появление новых сотрудников и другие изменения в команде.

Эта работа выполняется вручную, и некоторые операции плохо поддаются автоматизации, однако она предоставляет весьма ценную информацию о стоимости сопровождения и эксплуатации. Сравнивая их с ценностью результатов работы сервисов, инженеры по надежности могут определять цели в направлении повышения эффективности.

Резюме

Управление уровнем качества обслуживания — краеугольный камень проектирования и эксплуатации инфраструктуры. Мы не устанем подчеркивать, что все действия должны быть результатом планирования, чтобы избежать нарушений наших SLO. SLO создают правила игры, в которую мы играем. Мы используем SLO, чтобы решить, на какие риски можем пойти, какие архитектуры выбрать и как спроектировать процессы, необходимые для поддержки этих архитектур.

Усвоив материал этой главы, вы теперь понимаете основные концепции управления качеством обслуживания, включая SLA, SLO и SLI. Должны знать общие показатели, включая доступность, задержку, отказоустойчивость и эффективность. Вам также следует понимать подходы к эффективному мониторингу этих показателей, чтобы выявлять проблемы еще до того, как ваши SLO будут нарушены. Это должно дать вам хорошую основу для эффективного информирования о том, что ожидается от сервисов, которыми вы управляете, и для достижения поставленных целей.

В главе 3 мы рассмотрим управление рисками. Именно здесь мы начнем оценивать то, что может повлиять на показатели качества обслуживания, которых мы стремимся достичь. Используя эти требования к качеству обслуживания и зная потенциальные риски, мы можем эффективно разрабатывать сервисы и процессы, обеспечивая выполнение обязательств, которые дали бизнесу.

3 Управление рисками

Работа по сопровождению и эксплуатации — это набор обязательств[1] и действия, направленные на их выполнение. В главе 2 мы обсуждали, как создавать и формулировать эти обязательства, контролировать и документировать их выполнение. Управление рисками — это то, что мы делаем для выявления, оценки и приоритизации факторов неопределенности, которые могут привести к нарушению наших обязательств. Это также касается применения ресурсов (технологий, инструментов, людей и процессов) для мониторинга и снижения вероятности возникновения неопределенностей.

Это не наука о достижении идеала! Цель здесь не в том, чтобы устранить все риски. Это была бы безумная цель, которая лишь зря поглощала бы ресурсы. Цель состоит в том, чтобы внедрить оценку и снижение рисков во все наши процессы и постепенно уменьшить воздействие рисков, используя различные методы их предотвращения и смягчения. Данный процесс должен идти постоянно с учетом наблюдений за инцидентами, внедрения новых архитектурных компонентов, увеличения или уменьшения эффекта по мере развития организации. Цикл процесса можно разбить на семь категорий:

- определить возможные опасности/угрозы, создающие операционный риск для сервиса;
- оценить каждый риск с учетом его вероятности и последствий;
- классифицировать вероятность и последствия рисков;
- определить средства управления для смягчения последствий или снижения вероятности риска;
- определить приоритетность рисков, с которыми нужно бороться в первую очередь;
- внедрить средства управления и контролировать эффективность;
- повторить процесс.

---

[1] Выраженных в SLO. — *Примеч. науч. ред.*

Повторяя этот процесс, вы практикуете *кайдзен*, то есть постоянно совершенствуетесь. И самое важное здесь — оценка рисков, где нужно развивать стратегию постепенно.

Факторы риска

Качество *процессов оценки рисков* зависит от множества факторов. Их можно разбить на следующие категории:

- неизвестные факторы и сложность;
- наличие ресурсов;
- человеческий фактор;
- групповые факторы.

Чтобы разработать реалистичный процесс для вашей команды, необходимо учесть все эти факторы, и мы кратко рассмотрим в данной главе каждый из них.

Неизвестные факторы и сложность

Трудность оценки рисков заключается в огромной сложности современных систем. Чем сложнее и запутаннее предметная область, тем больше трудностей испытывают люди, перенося свои знания на ситуации, с которыми еще не сталкивались. Тенденция к чрезмерному упрощению понятий, чтобы с ними можно было легко справиться, называется *редуктивным смещением*. То, что работает при начальном обучении, не работает, когда приобретаются более глубокие знания. Существует огромное количество неизвестных рисков, многие из которых находятся вне нашего контроля. Вот некоторые примеры таких рисков:

- влияние других клиентов внешних совместно используемых систем, таких как Amazon или Google;
- влияние поставщиков, интегрированных в инфраструктуру;
- обновление кода программистами;
- движения на рынке, создающие всплески нагрузки на систему;
- выше- и нижестоящие сервисы в иерархии;
- патчи, изменения в репозитории и другие постепенно накапливающиеся изменения программного обеспечения.

Решение проблем в этих областях очень помогает оценивать риски в таких средах. Служба эксплуатации должна использовать свой коллективный опыт и постоянно

наращивать знания, чтобы строить все более точные модели планирования. Необходимо также понимать, что учесть все заранее невозможно, поэтому следует создавать систему отказоустойчивой и адаптируемой к новым неизвестным обстоятельствам.

Наличие ресурсов

Те из нас, кому приходилось работать в отделе, которому не выделяют достаточно средств, или в бессистемном стартапе, знают, что получить ресурсы для длительных, работающих на перспективу задач, подобных рассматриваемой, бывает... скажем так, непросто (слышали про сизифов труд?). Например, у вас может быть 4 часа или даже всего 30 минут в месяц на то, чтобы заняться процессами управления рисками. Поэтому ваша работа должна давать полезный результат — «приносить прибыль». Цена ваших времени и ресурсов, используемых для смягчения рисков, должна быть меньше цены бездействия. Другими словами, в первую очередь уделяйте время самым вероятным рискам, а также тем, последствия которых могут оказаться наиболее серьезными. Создавайте устойчивые системы и делайте выводы из возникающих инцидентов.

Человеческий фактор

Стоит людям начать что-то делать, как возникает *множество потенциальных проблем* (http://bit.ly/2zyoBmm). Мы все великолепны, но в прилагающихся к нам инструкциях много написанного мелким шрифтом. Вот некоторые моменты, которые могут испортить дело.

- *Синдром бездействия*. Многие сотрудники эксплуатационных отделов обнаружат, что их начальник или коллеги не склонны к риску. Будучи инертными по природе, эти люди выбирают бездействие, поскольку считают, что от перемен риск выше, чем от бездействия. Важно уметь просчитывать риск, а не опускать руки перед неизвестностью.
- *Игнорирование знакомых опасностей*. Опытные инженеры часто игнорируют типичные риски, уделяя больше внимания экзотическим и редким событиям. Например, те, кто привык иметь дело с переполнением дисков, могут сосредоточиться на проблемах центра обработки данных в целом, забыв адекватно спланировать управление дисковым пространством.
- *Страх*. Страх как фактор стресса может иметь как положительный, так и отрицательный эффект. Некоторые люди отлично чувствуют себя в условиях большой нагрузки и высоких ставок — они будут очень полезны, работая в области планирования, смягчения рисков и других вопросов промышленной эксплуатации систем. Но если человек действительно боится, он из-за этого нередко игнорирует

худшие сценарии. Это может привести к недостаточной проработке ключевых, наиболее ответственных и подверженных рискам компонентов и систем. Поэтому важно распознать такие реакции, если они есть у членов вашей команды.

- *Излишний оптимизм*. Другая тенденция человеческого поведения как реакции на риск — излишний и необоснованный оптимизм. Мы часто верим в лучшее относительно себя и других членов наших команд и поэтому склонны рассматривать только идеальные ситуации (никто не утомлен, нас не отвлекают другие инциденты, младшие сотрудники принимают участие в работе). Это касается не только людей, но и происходящих событий. Приходилось ли вам думать: *«Три диска в один день из строя не выйдут»*, а потом получить некачественную партию дисков, с которой именно это и случится?

Обсуждая риски, мы также должны учитывать такие факторы, как утомляемость и отвлечение на выполняемые вручную восстановительные работы (так сказать, аврал-пожар). Каждый раз, оценивая трудоемкость и риски, например, при внесении изменений вручную или при исследовании инцидента, мы должны учитывать, что специалисты-эксплуатационщики, погружающиеся в новую сложную проблему, пришли к нам, возможно, после долгого трудового дня. Может быть, это и не так, но это тоже нужно учитывать. При разработке средств управления для уменьшения или устранения рисков также следует учитывать, что тот, кто будет выполнять ручное восстановление, тоже может быть утомлен или, возможно, тушит несколько пожаров одновременно.

Усталость от оповещений

Усталость от оповещений (http://bit.ly/2zyfqCv) — это ситуация, когда перегрузка и усталость возникают из-за слишком большого количества ненужных сообщений. Вы должны учитывать это, принимая решение о том, сколько предупреждений (для реагирования и исправления вручную) встроено в процессы мониторинга. Такая ситуация бывает вызвана ложными срабатываниями (оповещениями о проблемах, которые на самом деле проблемами не являются, часто из-за плохо настроенных пороговых значений) или тем, что оповещения используются вместо предупреждений о тенденциях, способных стать опасными в ближайшем будущем.

Групповые факторы

Подобно тому как у каждого человека есть свои «слепые зоны», группам тоже свойственны особенности поведения и тенденции, которые могут нарушать процесс управления рисками. Вот факторы, которые следует учитывать.

- *Поляризация группы*. Поляризация группы, или «уклон в риск», возникает потому, что группы, как правило, принимают более экстремальные решения, чем

их отдельные члены. Это приводит к тенденции сдвига в противоположную от первоначальных взглядов сторону. Например, люди, обычно осторожные и осмотрительные, среди единомышленников будут более склонны к риску. А те, кто относился к риску спокойно, станут стремиться его избежать. Люди часто не хотят выглядеть наиболее консервативными в группе. Это может подтолкнуть команду рисковать сверх необходимого.

- *Передача риска*. Группы также будут допускать больше риска, когда у них есть тот, на кого можно его переложить. Например, планируя работу группы эксплуатации, мы можем пойти на больший риск, если знаем, что у нас есть отдельная группа сопровождения базы данных, к которой можно обратиться. Здесь помогут выработка чувства сопричастности и кросс-функциональные команды, где перекладывать риск не на кого.
- *Передача решения*. Передача решения может произойти в том случае, когда команды переоценивают риск и поэтому стремятся передать ответственность за конкретные решения другим. Например, если изменения, риск которых оценивается как высокий, требуют одобрения технического директора (следовательно, и ответственность ляжет тоже на него), то люди будут склонны завышать оценки риска, чтобы спихнуть принятие решений с себя вверх по цепочке. Этого можно избежать, создавая более автономные группы, которые опираются на знания и опыт отдельных специалистов и групп, а не на процедуры иерархического утверждения.

Что мы делаем

Реальность такова, что процесс управления рисками легко может стать чересчур обременительным. Даже при наличии значительных ресурсов команды все равно не смогут охватить все потенциальные риски, способные повлиять на доступность, производительность, стабильность и безопасность системы. Имеет смысл строить процесс повторяющимся и совершенствующимся с течением времени. Кроме того, предпочтение устойчивости в управлении рисками вместо стремления полностью их устранить позволяет разумно рисковать во имя инноваций и улучшений.

Мы хотели бы подчеркнуть, что устранение всех рисков — плохая цель. У систем, не испытывающих стрессовых воздействий, нет стимулов к улучшению и укреплению. В результате они оказываются нестойкими по отношению к неизвестным, не запланированным заранее факторам. Системы, регулярно испытывающие стрессы и, следовательно, спроектированные с учетом обеспечения устойчивости, способны успешнее справляться с неизвестными рисками.

Есть основания считать, что для систем следует планировать бюджет времени простоя, как в Google, чтобы реализовать новые перспективные возможности, сохраняя контроль над риском. В Google при бюджете простоя 30 минут за квартал, если это время не было израсходовано, считают оправданным допустить повышение риска ради обновления функций, улучшений и усовершенствований. Использовать бюджет простоя для внедрения инноваций, вместо того чтобы полностью уклоняться от риска, — это превосходно.

Итак, как из этого можно сформировать реальный подход к оценке риска как процесса? Начнем с того, чего делать не надо!

Чего не надо делать

Итак, нам нужно многое принять во внимание! Вот несколько советов, которые необходимо учесть при изучении процесса управления рисками — пока еще не поздно:

- не позволяйте субъективным предубеждениям повредить вашему процессу;
- не допускайте, чтобы основным источником оценки риска были анекдоты и сарафанное радио;
- не сосредотачивайтесь только на предыдущих инцидентах и проблемах — смотрите вперед;
- не останавливайтесь, сделанное ранее можно и нужно пересматривать;
- не игнорируйте человеческий фактор;
- не игнорируйте эволюцию архитектуры и рабочего процесса;
- не думайте, что ваша среда сейчас такая же, как и прежде;
- не создавайте нестабильные средства управления и не игнорируйте вероятность, что все пойдет не так.

Несомненно, со временем вы дополните этот список, но он неплох, чтобы держать его в памяти для начала, — это поможет избежать подводных камней, ожидающих вас при изучении своих систем.

Рабочий процесс: запуск

Независимо от того, является ли сервис новым или наследником существующего, процесс начинается с запуска. В ходе запуска процесса (рис. 3.1) цель состоит в том,

чтобы распознать основные риски, которые могут поставить под угрозу SLO вашего сервиса, или наиболее вероятные риски.

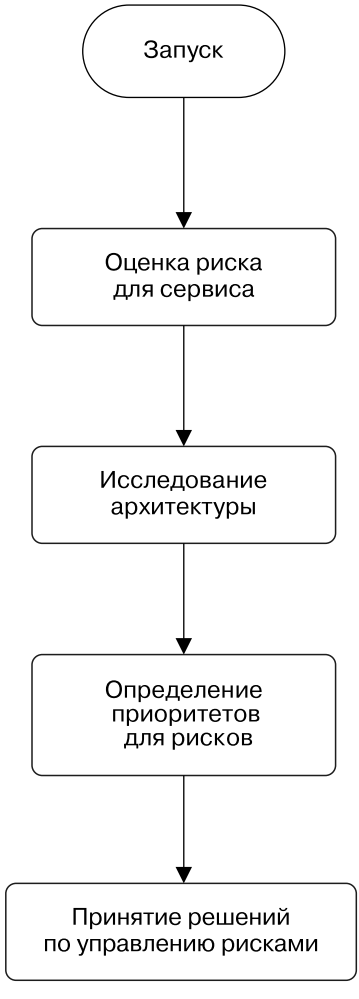


Рис. 3.1. Запуск процесса управления рисками

Кроме того, мы должны учитывать наихудшие сценарии, которые могут угрожать жизнеспособности сервиса в долгосрочной перспективе. Вам — бравому инженеру БД — следует помнить, что мы сейчас не пытаемся охватить все возможные риски. Пока что наша цель — составить начальный список рисков, которые мы будем пытаться смягчить или устранить в первую очередь, и план, позволяющий наиболее полно и с наибольшей отдачей использовать имеющиеся ресурсы.

Оценка риска для сервиса

Вооружившись списком поддерживаемых сервисов и микросервисов, вы должны сесть за стол вместе с владельцами продукта и оценить допустимость рисков для каждого из этих сервисов. Вам необходимо ответить на следующие вопросы.

- Каковы SLO доступности и времени ожидания, определенные для данного сервиса?
- Что представляют собой время простоя или недопустимая задержка для этой услуги, если:
  - затронуты все клиенты;
  - затронута часть клиентов;
  - сервис работает в режиме ограниченной функциональности (только чтение, отключены некоторые функции и т. п.);
  - снижена производительность сервиса?
- Какова цена простоя сервиса:
  - какова потеря дохода;
  - каково влияние на лояльность клиентов:
    - это бесплатный или платный сервис;
    - есть ли конкуренты, к которым клиент может легко перейти;
  - есть ли последствия простоя, способные подорвать работу всей компании:
    - потеря данных;
    - нарушение конфиденциальности;
    - простои во время праздника, иного особого события;
    - большая длительность простоя?

Рассмотрим пример. Компания UberPony специализируется на услуге предоставления клиентам пони по вызову, и ею предлагаются следующие шесть сервисов.

1. Регистрация нового клиента.
2. Прием и выполнение заказа на предоставление «пони по требованию».
3. Регистрация водителя пони.
4. Логистика для водителей пони.
5. Выплаты водителю пони.
6. Внутренняя аналитика.

Рассмотрим сервисы «Регистрация клиентов» (табл. 3.1) и «Заказ и выполнение заказов» (табл. 3.2).

Таблица 3.1. Регистрация клиентов UberPony

SLO доступности	99,90 %
SLO задержки	1 секунда
Количество новых клиентов в день	5000
SLO допустимых ошибок	5
Затраты на инфраструктуру в сутки	13 698 долларов
Затраты на инфраструктуру на 1 доллар дохода	0,003 доллара
Показатель ценности клиента[1]	1000 долларов
Показатель ценности новых клиентов за день (суммарно)	5 000 000 долларов
Пиковое количество обращений клиентов в минуту	100
Отсев клиентов после ошибки	60 %
Пиковое значение потерь за минуту	60 000 долларов

Таблица 3.2. Заказ и выполнение заказов в UberPony

SLO доступности	99,90 %
SLO задержки	1 секунда
Количество текущих заказов в день	500 000
SLO допустимых ошибок	500
Затраты на инфраструктуру в сутки	30 000 долларов
Затраты на инфраструктуру в пересчете на 1 доллар дохода	0,006 доллара
Средний доход от одного заказа	10 долларов
Ежедневный доход	5 000 000 долларов
Пиковое количество заказов в минуту	1000
Отмены заказов после ошибки	25 %
Отсев клиентов после ошибки	1 %
Пиковое значение потерь дохода за минуту	2500 долларов
Снижение суммарного показателя ценности клиентов (перспективные потери от отсева клиентов) за минуту	10 000 долларов
Всего потерь за минуту	12 500 долларов

Таким образом, похоже, что проблемы в сервисе регистрации клиентов могут обойтись нам в 4,8 раза дороже по сравнению с сервисом приема и выполнения заказов. Получается, 75 % клиентов будут повторять попытку заказа после ошиб-

[1] Англ. customer lifetime value — ожидаемая прибыль от одного клиента в течение всего жизненного цикла. — *Примеч. науч. ред.*

ки, но только 40 % вернутся, если не смогут зарегистрироваться. Видимо, вместо этого они с радостью уйдут на UberDonkey. Обратите внимание: мы попытались учесть такие переменные, как потеря клиента после ошибки заказа и количество повторных попыток клиентов зарегистрироваться или сделать заказ после ошибки. Это будет непросто без хорошей бизнес-аналитики, но при недостатке данных можно довольствоваться хотя бы приближенными оценками. Во всяком случае, это лучше, чем ничего!

Со временем данные будут изменяться и дополняться, поэтому обязательно вовремя обновляйте показатели. Например, если UberDonkey станет более конкурентоспособным и UberPony начнет терять 5 % клиентов из-за ошибки в заказе, то наши потери за минуту простоя сервиса приема и выполнения заказов внезапно вырастут до 52 500 долларов, что значительно повысит его приоритет. Тогда нам будет важнее переключить внимание на этот сервис.

Ревизия архитектуры

Теперь, определив сферу деятельности, проведем ревизию систем и сред, входящих в сферу нашей ответственность:

- центры обработки данных (ЦОД);
- архитектурные компоненты/уровни (СУБД MySQL, балансировщики нагрузки Nginx, экземпляры приложений J2EE, сеть, брандмауэр, Hadoop/HDFS, сеть доставки контента (CDN));
- распределение ролей внутри этого компонента (Writer/Primary, Replica);
- пути и порядок взаимодействия между сервисами (запросы из приложения в MySQL, из балансировщика нагрузки в приложение, публикация приложения в Redis);
- задания (извлечение, преобразование и загрузка (ETL), загрузка CDN, обновление кэша, управление конфигурацией, оркестрация (общая координация), резервное копирование и восстановление, агрегация журналов).

В табл. 3.3 представлен упрощенный перечень первоочередных сервисов.

Таблица 3.3. Регистрация клиентов UberPony

Компонент	**ЦОД 1**	**ЦОД 2**
Фронтальные балансировщики нагрузки	2	2
Веб-серверы	20	20
Балансировщики нагрузки Java	2	2
Java-серверы	10	10

Продолжение ⇘

Таблица 3.3 *(продолжение)*

Компонент	ЦОД 1	ЦОД 2
Прокси базы данных	2	2
CloudFront CDN	Сервис	Сервис
Кэш-серверы Redis	4	4
Серверы записи кластера MySQL	1	0
Серверы чтения кластера MySQL	2	2
Репликация MySQL	Сервис	Сервис
Обновление CDN	Задание	Задание
Обновление кэша Redis	Задание	Задание
Резервные копии MySQL	Задание	Не определено
Процесс ETL	Задание	Не определено
Хранилище данных RedShift	Сервис	Не определено

Наш следующий шаг — оценить риски в этой архитектуре, которые могут повлиять на работу сервиса.

Расстановка приоритетов

Как выявить риски, способные привести к нарушению целей SLO, и расставить их приоритеты? В области управления рисками используется определение риска как произведение вероятности[1] возникновения фактора, приводящего к неблагоприятному исходу, и величины последствий этого исхода.

Пример спектра возможных оценок приведен в табл. 3.4.

Таблица 3.4. Спектр оценки

Вероятность/ последствия	Тяжелые	Значительные	Умеренные	Незначительные	Ничтожно малые
Почти наверняка	Неприемлемый	Неприемлемый	Высокий	Средний	Приемлемый
Очень вероятно	Неприемлемый	Высокий	Высокий	Средний	Приемлемый
Возможно	Неприемлемый	Высокий	Средний	Средний	Приемлемый
Маловероятно	Высокий	Средний	Средний	Приемлемый	Приемлемый
Редко	Высокий	Средний	Приемлемый	Приемлемый	Приемлемый

[1] В оригинале здесь likelihood; ниже имеется замечание о сложности толкования понятий likelihood и probability. — *Примеч. науч. ред.*

Для устранения двусмысленности важно определить возможные значения для вероятности и цены исхода. Цена может различаться в зависимости от конкретной предметной области и задачи. Что касается неоднозначности понятий вероятности, ожидаемой вероятности, частоты, то мы предлагаем вам изучить публикацию *Describing probability: The limitations of natural language* («Описание вероятности: ограничения естественного языка», http://www.risk-doctor.com/pdf-files/emeamay05.pdf).

Разделим все вероятности следующим образом (табл. 3.5).

Таблица 3.5. Вероятности

Качественная оценка	Количественная оценка, %
Почти наверняка	> 50
Очень вероятно	26–50
Возможно	11–25
Маловероятно	5–10
Редко	< 5

Будем считать эти проценты вероятностью нарушения SLO в течение определенного периода времени, например за неделю. Что же касается последствий, то классифицируем их, оценивая влияние на наши SLO в случае реализации нарушающих нормальный процесс факторов, включая повреждение данных, утрату конфиденциальности и нарушения системы безопасности. Большинство этих проблем попадают в категории сильного или значительного воздействия. Опять же это только примеры.

Тяжелые последствия (немедленное нарушение SLO)

Тяжелые последствия означают следующее.

- Недоступность всего сервиса или ограничение его функциональности на срок в общей сложности от 100 миллисекунд до 10 минут и более для 5 % пользователей и более. (В семидневной неделе 10 080 минут. Десять минут простоя поглощают 99,9 % SLO.)
- Состоявшееся или неизбежное в будущем раскрытие данных клиента другим клиентам.
- Предоставление посторонним лицам доступа к рабочим (production) системам и/или реальным данным.
- Повреждение данных транзакций.

Любое из перечисленных событий переводит последствия в категорию тяжелых.

Значительные последствия (неминуемое нарушение SLO)

Значительные последствия означают следующее.

- Недоступность всего сервиса или ограничение его функциональности на срок от 100 миллисекунд до 3–5 минут для 5 % пользователей и более (до 50 % бюджета времени доступности).
- Уровень доступных мощностей системы снизился до 100 % от необходимого вместо запланированного значения 200 %.

Любое из перечисленных событий переводит последствия в категорию значительных.

Умеренные последствия (возможно нарушение SLO в сочетании с другими инцидентами, произошедшими в тот же период)

Умеренные последствия означают следующее.

- Недоступность всего сервиса или ограничение его функциональности на срок от 100 миллисекунд до 1–3 минут для 5 % пользователей и более (до 33 % бюджета времени доступности).
- Уровень доступных мощностей системы снизился до 125 % от необходимого вместо запланированного значения 200 %.

Любое из перечисленных событий переводит последствия в категорию умеренных.

Незначительные последствия

Незначительные последствия означают следующее.

- Недоступность сервиса или ограничение его функциональности на срок от 100 миллисекунд до 1 минуты для 5 % пользователей и более (до 10 % бюджета времени доступности).
- Уровень доступных мощностей системы снизился до 150 % от необходимого вместо запланированных 200 %.

Любое из перечисленных событий переводит последствия в категорию незначительных.

Напомним: мы не станем пытаться охватить все потенциальные риски. Вы будете пополнять их портфель изо дня в день в рамках текущих процессов управления инцидентами и рисками. То, что мы сейчас делаем, называется *фреймингом*: мы ограничиваем нашу работу некими разумными рамками. В данном случае мы

включаем в эти рамки сценарии, *наиболее вероятные* и имеющие *наибольшие последствия*.

Например, мы знаем, что сбои компонентов и сбои экземпляров — это рядовые события в общедоступных облачных средах, наподобие той, которую UberPony использует в качестве хоста. Другими словами, там малое среднее время между отказами (MTBF). Мы будем классифицировать эти сбои как вероятные для групп веб-экземпляров и групп экземпляров Java, поскольку они будут встречаться нам в умеренном количестве в любое время (от 20 и 10 соответственно). Однако отказ одного веб-экземпляра означает, что будут затронуты 5 % клиентов. При сбое экземпляра Java окажутся затронуты 10 % клиентов. Это является нарушением SLO, и поскольку запуск новой копии данного экземпляра может занять от 3 до 5 минут, это повлечет *значительные* последствия для сервиса. При вероятности, оцениваемой как *«очень вероятно»*, и *значительных* последствиях риск *высокий*. После того как мы включим автоматическое восстановление после сбоев (вывод аварийного экземпляра из эксплуатации и запуск вместо него нового), этот процесс будет занимать, согласно проведенным тестам, в среднем 5 секунд. Это изменит последствия на *незначительные*, а риск соответственно на *умеренный*.

Если в нашем исследовании рассматривать сбой на уровне сервиса или экземпляра, то можно составить примерно такую таблицу (табл. 3.6).

Таблица 3.6. Сервис регистрации клиентов UberPony

Компонент	**Вероятность**	**Последствия**	**Риск**
Балансировщик клиентской нагрузки	Возможно	Тяжелые	Неприемлемый
Веб-серверы	Вероятно	Значительные	Высокий
Балансировщики нагрузки Java	Возможно	Значительные	Высокий
Java-серверы	Вероятно	Значительные	Высокий
Прокси базы данных	Возможно	Значительные	Высокий
Cloudfront CDN	Редко	Значительные	Умеренный
Кэш-серверы Redis	Возможно	Значительные	Умеренный
Серверы записи MySQL	Маловероятно	Тяжелые	Высокий
Серверы чтения MySQL	Возможно	Значительные	Высокий
MySQL-репликация	Возможно	Значительные	Высокий
Обновление CDN	Маловероятно	Незначительные	Приемлемый
Redis-обновление кэша	Маловероятно	Незначительные	Приемлемый
Резервные копии MySQL	Маловероятно	Значительные	Приемлемый
Процесс ETL	Маловероятно	Незначительные	Приемлемый
Хранилище данных RedShift	Редко	Незначительные	Приемлемый

Используя фрейминг, мы сначала погрузимся в ситуации неприемлемого или высокого риска, затем можно заняться случаями умеренного риска и т. д. В разделе, посвященном базам данных, мы выполним более детальную оценку рисков для БД. Наша цель — помочь вам разобраться в процессе. Еще одна особенность заключается в том, что необходимо учитывать также все риски на уровне центра обработки данных. Они встречаются редко, но относятся к той же категории, что и нарушение конфиденциальности, потеря данных и другие риски, требующие рассмотрения в связи с потенциальными последствиями для бизнеса.

Управление рисками и принятие решений

Теперь, когда у нас есть список рисков для оценки с расставленными приоритетами, рассмотрим методы принятия решений по управлению рисками для их смягчения и по возможности устранения. Мы уже начали делать это в предыдущем разделе для веб-серверов и серверов Java, автоматизировав замену аварийных компонентов, чтобы сократить среднее время восстановления (MTTR) после сбоя. Помните, что мы должны позаботиться в первую очередь о быстром восстановлении и сокращении MTTR, а не об устранении сбоев. Наша цель — устойчивость, а не нестабильно высокая доступность!

Почему MTTR важнее, чем MTBF

Создавая систему, которая редко ломается, вы делаете ее изначально неустойчивой. Будет ли ваша команда готова выполнить ремонт, когда система выйдет из строя? Будет ли она вообще знать, что делать? Если же в системе часто происходят отказы, последствия которых смягчаются и остаются незначительными, то команда знает, что делать, когда все идет не так, как надо. Процессы хорошо документированы и отточены, и автоматическое восстановление становится полезным и действует по-настоящему, а не скрывается в темных углах вашей системы.

Для каждого потенциального риска команда может выбрать один из трех подходов:

- избегание (найти способ устранить риск);
- сокращение (найти способ уменьшить последствия риска, когда он реализуется);
- принятие (обозначить риск как допустимый и соответствующим образом запланировать ответные действия).

В сфере управления рисками существует и четвертый подход, называемый разделением рисков, предусматривающий аутсорсинг, страхование и другие способы передачи или делегирования рисков. Однако все это не относится к риску в ИТ, поэтому анализировать его не будем.

Для каждого компонента мы рассмотрим типы сбоев, их последствия и несколько приемов управления, направленных на автоматизацию восстановления, сокращение времени восстановления и снижение частоты сбоев. Это управление потребует определенных затрат и усилий. Сравнивая затраты с возможными потерями при простоях, мы сможем принять правильное решение о выборе мер по снижению рисков.

Выявление рисков

Оценивая риски для компании UberPony, мы определили риски на уровне сервиса хранения данных MySQL как высокие. Это очень типично для уровня базы данных. Итак, посмотрим, что можно сделать, чтобы уменьшить эти риски. В этом сервисе выявлены четыре ключевые точки возможных отказов:

- сбой модуля записи данных;
- сбой модуля чтения данных;
- сбой модуля репликации;
- сбой модуля резервного копирования.

Все это типичные точки отказа для хранилищ данных.

Оценка

Чтобы оценить риск ошибок записи, команда эксплуатации UberPony садится и определяет свои возможности по автоматизации восстановления после сбоя в модуле записи в MySQL. При аварии модуля записи наш сервис «Регистрация клиентов» не может создавать или изменять какие-либо данные. Это означает: никаких новых клиентов и никакой возможности изменять данные ни для самих клиентов, ни для UberPony. Мы уже определили, что неработоспособность сервиса регистрации на пике нагрузки может приводить к потере до 60 000 долларов «стоимости» новых клиентов каждую минуту. Понимать это очень важно! Это означает, что *принятие риска* в данном случае не подходит.

Смягчение рисков и управление ими

Некоторые меры по *устранению риска* у нас уже приняты. У нас есть RAID-массив с десятью дисками, который обеспечивает избыточность хранения данных, поэтому сбой отдельного диска не приводит к сбою всей базы данных. Аналогично избыточность предусмотрена и в среде выполнения. Другой подход к *устранению риска* — замена базового ядра СУБД MySQL на Galera — архитектуру, позволяющую выполнять запись в любой узел кластера MySQL. Это потребует значительных изменений архитектуры, и никто из команды не имеет соответствующего

опыта. Обсудив риски, вносимые новой системой, мы пришли к выводу, что они перевешивают выгоды от нового подхода.

Если приложение спроектировано правильно, то даже при аварии модуля записи клиенты все равно могли бы входить в сервис и просматривать свои данные через модуль чтения. Это и есть *смягчение риска*. Пообщавшись с командой разработчиков программного обеспечения, мы узнаем, что у них это предусмотрено. Но в таком режиме ограниченной функциональности новые клиенты регистрироваться по-прежнему не могут, поэтому ценность этой функции оказывается не очень высокой (однако другие функции, которые пока не разрабатываются, высоко ценятся на конкурентном рынке).

В итоге команда решает реализовать автоматическое восстановление — в данном случае автоматическое переключение на другой мастер-диск. Было выбрано восстановление в автоматическом, а не ручном режиме, потому что за десять минут простоя, допускаемых согласно SLO, оператор просто не успевает войти в систему и выполнить необходимые действия. И поскольку вероятность потери данных при записи все же остается, этот процесс должен быть очень надежным.

Реализация

В качестве технологии для автоматического восстановления при сбоях команда выбрала MySQL MHA. MySQL MHA (MySQL High Availability) — это программное обеспечение для управления восстановлением, переключением аварийных компонентов и соответствующими изменениями в топологии репликации. Прежде чем реализовать столь важный процесс, команда разработала план интенсивного тестирования. Такие тесты выполняются поэтапно, сначала в тестовой среде без трафика, затем в тестовой среде с моделируемым трафиком и, наконец, в реальной производственной среде, сопровождаясь тщательным мониторингом. Эти тесты проводят много раз, чтобы убедиться, что они охватывают не только отдельные частные случаи. Тесты включают в себя:

- корректное отключение основной базы данных в тестовой среде;
- принудительное отключение процесса MySQL в тестовой среде;
- принудительное отключение экземпляра сервера, на котором работает MySQL, в тестовой среде;
- моделирование фрагментации сети на изолированные сегменты.

После каждого теста команда должна:

- записать время, которое потребовалось на восстановление работоспособности;
- записать величину задержки для моделируемого и реального трафика с целью оценки влияния на производительность;

- убедиться в отсутствии повреждений таблиц;
- убедиться в отсутствии потерь данных;
- проверить журналы ошибок на стороне клиентов, чтобы увидеть, как это повлияло на них.

После того как команда убедилась, что система будет работать в соответствии с заданными SLO, она начинает думать, как включить этот процесс в другие ежедневные процессы, чтобы обеспечить его регулярное выполнение, качественное документирование и отсутствие ошибок. Первоначально его решают включить в процесс развертывания, используя процесс восстановления после отказа для выполнения внесения изменений в базы данных, не влияя на однопоточные процессы репликации MySQL. К моменту завершения этих работ время восстановления после отказа сократилось до 30 секунд и менее.

Поскольку процессы аварийного восстановления актуальны и для команды разработки программного обеспечения, они тоже понимают, что в течение этого 30-секундного интервала также возможна потеря данных. Поэтому разработчики выполняют двойную запись для своих приложений, отправляя все вставки, обновления и удаления брокеру событий на случай необходимости восстановления утерянных данных. Это еще одна мера смягчения последствий в случае сбоя модуля записи.

Все эти приемы управления хороши на ранних стадиях построения системы. Важно помнить, что они не обязательно должны быть идеальными. Это лишь самое начало, когда мы решаем в первую очередь наиболее приоритетные и наиболее значимые задачи.

Процесс запуска системы завершен — вы прошли долгий путь, охватив наиболее распространенные случаи рисков. С этого момента начинается постепенное совершенствование.

Постепенное совершенствование

После завершения процесса запуска приоритеты по устранению рисков и смягчению их последствий включаются в общий технологический процесс (architectural pipeline) проектирования, сборки и текущего обслуживания. Ранее мы упоминали, что управление рисками — это непрерывный процесс; таким образом, нет необходимости реализовывать все с самого начала: постепенно развиваясь, процессы дополнят портфель рисков, обеспечивая все более широкий охват (рис. 3.2). Итак, что это за процессы?

- Периодическая ревизия сервисов (service delivery reviews).
- Управление инцидентами.
- Организация технологического процесса.

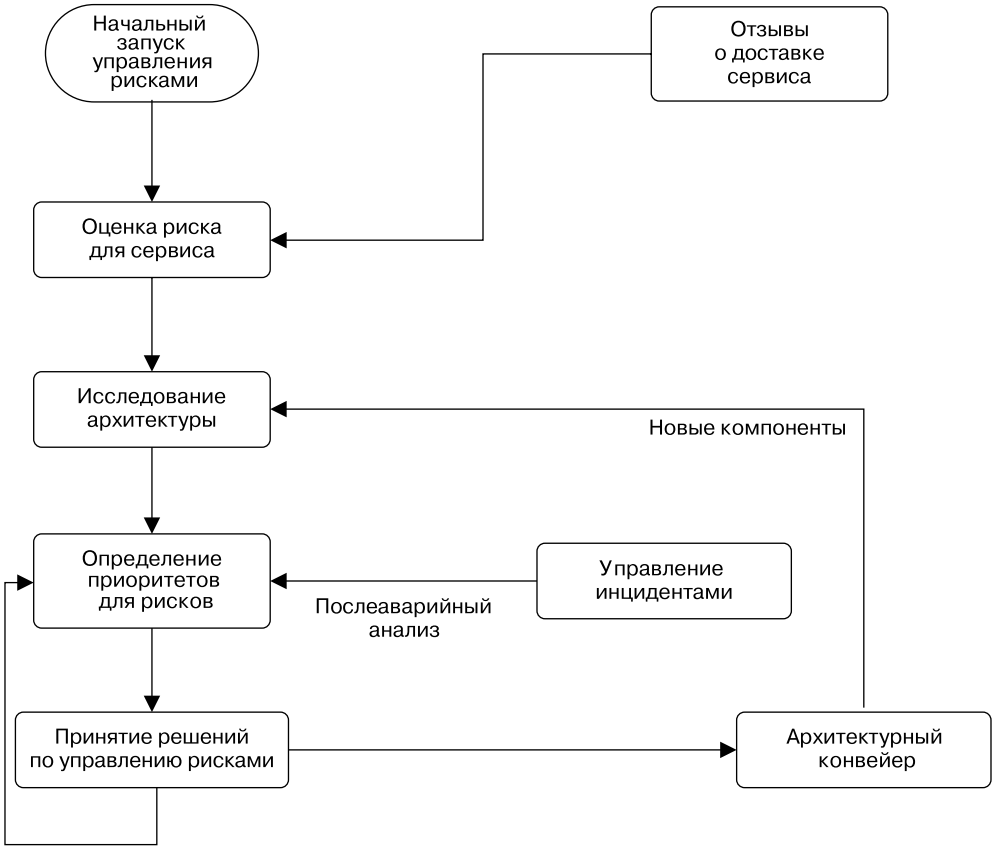


Рис. 3.2. Развивающийся жизненный цикл и его вклад в процесс управления рисками

Ревизия сервисов — это периодический возврат к доработкам сервиса с учетом изменений в допустимости рисков и их последствий, в доходах и пользовательской базе. Если изменения этих параметров существенны, предыдущие классификации рисков и решения для их устранения или смягчения их последствий должны пересматриваться, чтобы оставаться адекватными.

Процессы управления инцидентами также создают предпосылки для изменения приоритетов рисков. По мере выявления новых уязвимостей при анализе аварий необходимо изучать эти уязвимости и добавлять их в список с соответствующими приоритетами. Наконец, в уже построенный технологический процесс необходимо внедрять новые компоненты управления рисками, чтобы выявлять те риски, которые могли быть упущены на этапе проектирования.

Резюме

Итак, вы уже узнали о важности включения управления рисками в повседневные процессы ИТ. Вы ознакомились с некоторыми соображениями и факторами, которые могут повлиять на это, и рассмотрели реалистичный пример процессов как запуска новой системы, так и повседневных, позволяющих поддерживать последовательное развитие системы управления рисками.

Даже понимая наши обязательства в отношении качества уровня обслуживания и потенциальных рисков, угрожающих их выполнению, мы все еще упускаем жизненно важную составляющую — оперативный контроль (operational visibility). Для своевременного выявления проблем и принятия решений о направлении развития наших систем нам необходимо ясное представление о текущей ситуации, равно как знания об изменениях производительности и других характеристик систем с течением времени.

4 Оперативный контроль

Оперативный контроль, часто называемый мониторингом, — это краеугольный камень искусства проектирования надежных баз данных. Оперативный контроль позволяет знать все о рабочих характеристиках сервиса БД благодаря регулярному измерению параметров и сбору сведений о различных компонентах. Почему это так важно? Зачем нужен оперативный контроль? Вот лишь некоторые причины.

- *Исправления и изменения в связи с неисправностями.* Нам нужно знать, когда что-то повреждено или вот-вот будет повреждено, чтобы вовремя это исправить, не допуская нарушения целевых показателей качества обслуживания (SLO).
- *Анализ производительности и поведения.* Важно понимать, каковы задержки в наших приложениях и где они возникают, а также видеть их изменения со временем, включая пиковые значения и всплески. Эти данные играют решающую роль в понимании того, как влияет использование новых функций, эксперименты и оптимизация.
- *Планирование мощностей.* Возможность соотносить поведение пользователя и эффективность приложений с реальными ресурсами — процессором, сетью, хранилищем данных, пропускной способностью и памятью — крайне важна для предотвращения нехватки ресурсов в критический для бизнеса момент.
- *Отладка и послеаварийный анализ.* Чем интенсивнее изменения, тем чаще что-нибудь оказывается повреждено. Хороший оперативный контроль дает возможность быстро находить точки сбоя и точки оптимизации, позволяющие снизить риск в будущем. Человеческие ошибки никогда не были основной причиной поломок, но системы всегда можно улучшить и сделать их более устойчивыми.

ЧЕЛОВЕЧЕСКАЯ ОШИБКА ДЕЙСТВИТЕЛЬНО НИКОГДА НЕ ЯВЛЯЕТСЯ ПЕРВОПРИЧИНОЙ АВАРИИ?

При анализе инцидента или проблемы может возникнуть соблазн назвать их основной причиной человеческую ошибку. Однако, копнув глубже, можно обнаружить, что то, что казалось человеческой ошибкой, на самом деле проявление более фундаментальных проблем процесса или среды. Как такое может быть? Вот несколько причин:

- нестабильная, плохо оснащенная или слишком сложная система способствует совершению ошибок людьми;
- процесс, который не учитывает человеческие потребности, такие как сон, необходимое рабочее окружение или наличие определенных навыков, также может провоцировать людей совершать ошибки;
- в результате нарушения процедур найма операторов и их обучения в рабочую среду могут попадать неподходящие и недостаточно подготовленные операторы.

Кроме того, понятие «первопричины» само по себе неверно, поскольку ошибки и инциденты редко бывают вызваны одной-единственной причиной. Чем сложнее система, тем сложнее сбои, и участие людей лишь усложняет ситуацию. Вместо того чтобы искать первопричину сбоя, мы предлагаем рассмотреть *список* факторов, ему способствующих, с расстановкой приоритетов рисков и воздействий.

- *Бизнес-анализ*. Понимание того, как используется функционал вашей системы, может сыграть ведущую роль в выявлении проблем. Но не менее важно видеть, как люди применяют ваши функции и как это влияет на соотношение цены и эффективности[1].
- *Причинно-следственные связи*. Благодаря тому что события в инфраструктуре и приложениях регистрируются и журналируются в стеке оперативного контроля, можно быстро соотнести изменения рабочей нагрузки, поведения и доступности. Примерами таких событий являются развертывание приложений, изменение инфраструктуры и изменение схемы баз данных.

Практически каждый аспект вашей организации требует истинного оперативного контроля — *OpViz*. Эта глава призвана помочь вам понять, что такое наблюдаемость в архитектурах, с которыми вы будете работать. Мы не описываем здесь какой-либо конкретный набор инструментов, но существуют общие принципы, общая классификация и шаблоны использования, которые следует изучить. Мы продемонстрируем их на многочисленных примерах конкретных ситуаций. Сначала

[1] В тексте оригинала value versus cost. Предположительно, под value здесь имелась в виду сумма возможностей системы, общая отдача от ее использования. — *Примеч. науч. ред.*

рассмотрим эволюцию OpViz от традиционных подходов к тому, что используется в настоящее время.

ТРАДИЦИОННЫЙ МОНИТОРИНГ

Традиционные системы мониторинга обычно характеризуются следующим:

- хостами чаще всего являются реальные серверы, а не виртуальные экземпляры или контейнеры, и они имеют длительный срок службы, измеряемый месяцами, а иногда и годами;
- сетевые и аппаратные адреса стабильны;
- основное внимание уделяется системам, а не сервисам;
- в первую очередь контролируются уровень нагрузки и статические пороги (симптомы), а не «пользовательские» показатели (показатели качества обслуживания);
- разные структурные компоненты (например, сеть и база данных) обслуживаются разными системами;
- предусмотрена низкая детализация (с интервалом 1 или 5 минут);
- больше внимания уделяется сбору данных (опросу) и их представлению, а не анализу;
- большие затраты на администрирование и зачастую меньшая, по сравнению с контролируемым сервисом, стабильность.

Подведем итоги с позиции традиционного мониторинга. В традиционных средах администраторы БД в основном задавались такими вопросами: «Работает ли моя база данных?» и «Устойчиво ли она работает?». Они не учитывали, как поведение БД влияло на задержки для пользователей, изучая гистограммы задержек только с точки зрения распределения и потенциальных выбросов. Зачастую они бы и хотели, чтобы это стало возможным, но у них не было инструментов (bit.ly/2zkfkhc).

Оперативный контроль очень важен! Поэтому нам необходимы определенные правила проектирования, построения и использования этого критически важного процесса.

Новые правила оперативного контроля

Современный оперативный контроль исходит из того, что хранилища данных распределенные и часто очень масштабные, при этом анализ данных более важен, чем просто их сбор и представление. В процессе оперативного контроля мы всегда задаем два вопроса (и, хотелось бы надеяться, быстро отвечаем на них): «Как это влияет на SLO?» и «Каким образом это нарушается и почему?». Другими

словами, вместо того чтобы рассматривать стек OpViz как набор утилит, которые будут переданы команде эксплуатации, вы должны спроектировать, построить и поддерживать его как платформу бизнес-аналитики (Business Intelligence, BI). К оперативному контролю следует относиться так же, как к хранилищу данных или платформе для обработки больших данных. Чтобы это учесть, приходится изменить правила игры.

Относитесь к системам OpViz как к системам BI

Разработка BI-системы начинается с изучения того, какого рода вопросы будут задавать ваши пользователи, и система строится так, чтобы соответствовать этим вопросам. Необходимо учесть потребности пользователей, касающиеся времени доступа к данным («Как быстро можно получить данные?»), их подробности («Насколько глубоко можно изучать данные?») и доступности вообще. Другими словами, вы определяете SLO для вашего сервиса OpViz (см. главу 2).

Отличительной чертой зрелой платформы OpViz является то, что она способна сообщать не только о состоянии инфраструктуры, в которой выполняется приложение, но и о поведении приложений, работающих в этой инфраструктуре. В конечном счете она должна показывать, как функционирует бизнес-логика и как на нее влияют инфраструктура и другие приложения, на которые это функционирование опирается. С учетом всего этого платформа OpViz и должна обеспечивать поддержку инженерам по эксплуатации и базам данных, разработчикам программного обеспечения, бизнес-аналитикам и руководителям.

Тенденции в эфемерных распределенных средах

Мы уже говорили, что после внедрения виртуальных инфраструктур жизненный цикл экземпляров БД сокращается. И хотя его длительность все равно остается намного больше, чем у других компонентов инфраструктуры, нам нужны агрегированные показатели для сервисов, состоящих из короткоживущих компонентов, а не для отдельных экземпляров баз данных.

На рис. 4.1 показана более или менее стабильная конфигурация «мастер/реплика» реляционного хранилища данных, где в течение одного дня выполняется множество разнообразных действий. К концу дня конфигурация может заметно измениться (рис. 4.2).

Такая динамическая инфраструктура требует хранения показателей на основе ролей (функций), а не на основе имен хостов или IP-адресов. Поэтому вместо сохранения набора показателей конкретно для DB01 мы бы добавили показатели к роли «мастер», что позволило бы отслеживать поведение главной базы даже после перехода этой роли к новому экземпляру. Это обеспечивают системы по-

иска и обнаружения сервисов, выполняющие большую работу по поддержанию абстракции над динамическими частями инфраструктуры.

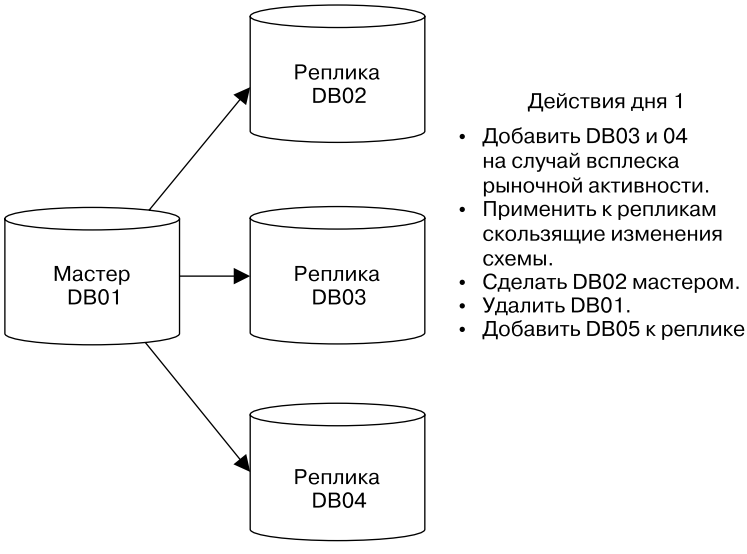


Рис. 4.1. Типичная конфигурация «мастер/реплика»

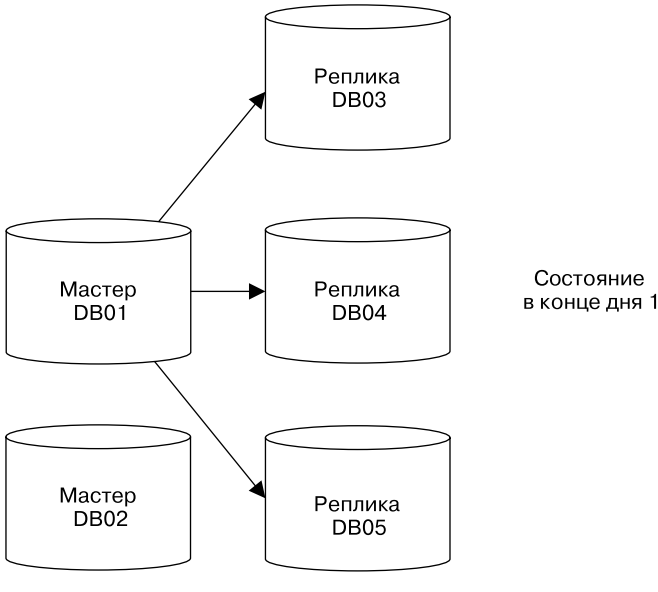


Рис. 4.2. Она же в конце дня

Хранение ключевых показателей с высокой детализацией

Как было показано в главе 2, высокая степень детализации имеет решающее значение для понимания уровня загруженности приложений. Все, что связано с SLO, следует записывать с периодичностью 1 секунда или менее, чтобы действительно понимать, что происходит в системе. Хорошее эмпирическое правило состоит в том, чтобы определить, может ли показатель за время от 1 до 10 секунд измениться настолько, чтобы повлиять на SLO, и на основании этого устанавливать степень детализации.

Например, если вы контролируете ограниченный ресурс, скажем процессор, то имеет смысл собирать эти данные с периодичностью 1 секунда или меньше, учитывая, что очереди приложений в ожидании процессора могут быстро заполняться и освобождаться. При SLO задержки, измеряемых в миллисекундах, этого должно быть достаточно, чтобы понять, является ли причиной задержки перегрузка процессора. Высокой частоты выборок требует также и мониторинг очередей запросов к базе данных.

И наоборот, для медленно меняющихся показателей, таких как дисковое пространство или доступность сервиса, можно проводить измерения с периодичностью 1 минута или даже реже без риска потери данных. При повышении частоты выборок растет и расход ресурсов, поэтому такой выбор должен быть обоснованным. Также, вероятно, стоит использовать не более пяти различных частот выборок, чтобы платформа OpViz оставалась простой и структурированной.

В качестве примера недостаточной частоты выборок рассмотрим график на рис. 4.3.

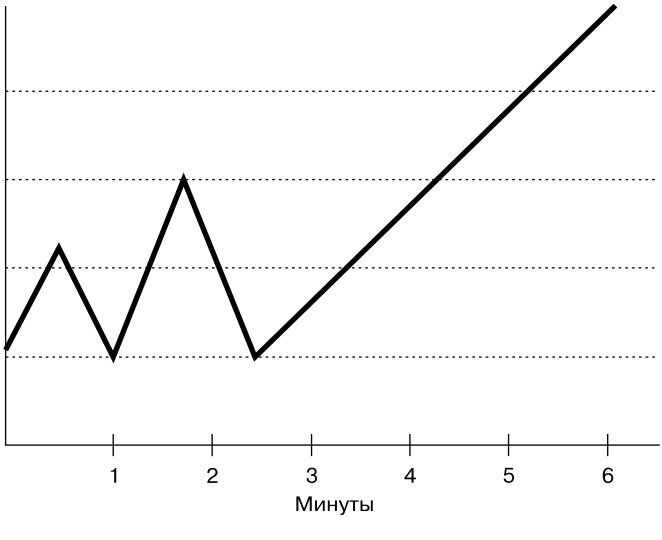


Рис. 4.3. Реальная нагрузка с пиками

Здесь имеются два пика, за которыми следует долгое возрастание. Если же снимать значения этого показателя с периодом 1 минута, то график будет выглядеть так, как представлено на рис. 4.4.

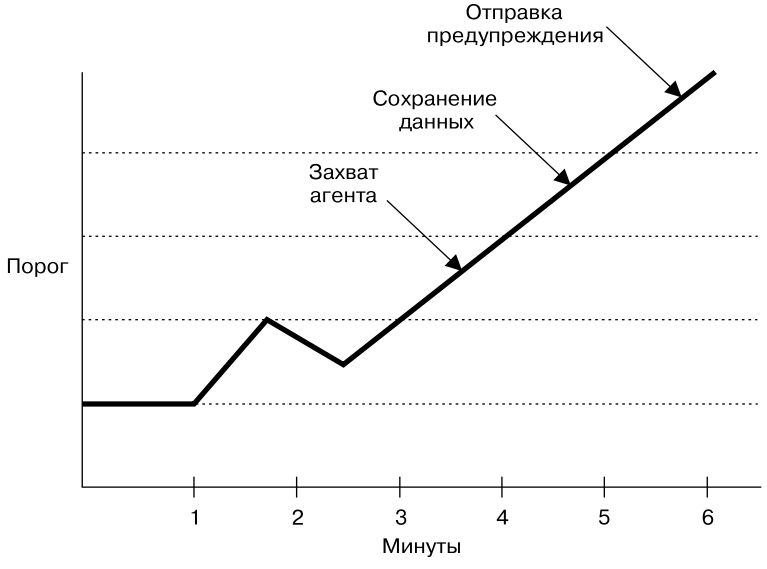


Рис. 4.4. Рабочая нагрузка визуализируется с периодом 1 минута

Обратите внимание, что на втором графике не видно ни одного пика, превосходящего пороговое значение, и он выглядит намного более плавным. На самом деле до третьей минуты мы даже не превысили порог оповещения. Предполагая, что сохранение показателя и проверка правил оповещения выполняются раз в минуту, мы даже не отправим уведомление оператору до тех пор, пока после реального превышения ограничения не пройдет 7,5 минуты!

Сохраняйте простоту архитектуры

Для развивающейся платформы обычная ситуация — 10 000 и более показателей, проверяемых на различных уровнях детализации для инфраструктуры с любым количеством экземпляров и серверов, которые в любой момент выходят из строя и снова включаются. Ваша цель состоит в том, чтобы иметь возможность быстро ответить на упомянутые ранее вопросы, а это значит, что приходится постоянно добиваться снижения отношения «сигнал/шум». То есть нужно быть безжалостными к объему данных, которые вы пропускаете в свою систему, особенно в точках взаимодействия с человеком, таких как представление информации и оповещение.

Довольно много времени прошло под лозунгом «мониторинга всего», и это было закономерным следствием ситуации, когда мониторинг был редким и выбороч-

ным. Сейчас реальность такова, что распределенные системы и мультисервисные приложения порождают слишком много показателей. У молодых организаций нет ни денег, ни времени, чтобы управлять таким количеством данных мониторинга. Крупные организации должны хорошо знать, что критически важно для их систем, чтобы сосредоточиться именно на этом.

ЦЕЛЕНАПРАВЛЕННОСТЬ ПОКАЗАТЕЛЕЙ

Сначала сосредоточьтесь на показателях, непосредственно связанных с вашими SLO. Это можно назвать основным направлением контроля. Опираясь на сведения из главы 2, ответьте, какие показатели должны быть приоритетными для сбора вашей платформой OpViz? Давайте посмотрим.

- *Задержка*. Клиент обращается к вашему сервису. Как долго ему придется ждать?
- *Доступность*. Сколько обращений завершаются ошибками?
- *Частота обращений*. Как часто происходят обращения к вашему сервису?
- *Загруженность*. Наблюдая за сервисом, вы должны знать, насколько полно загружены критически важные ресурсы, — это необходимо для обеспечения качества обслуживания и пропускной способности.

Разумеется, как инженер по обеспечению надежности БД, вы должны немедленно начать разносить эти показатели по подсистемам хранилища данных. Это имеет смысл, и это естественное развитие процесса, которое будет обсуждаться позже в главе.

Упрощению и выделению «полезного сигнала» служит также и стандартизация. Здесь имеется в виду стандартизация типовых решений, уровней детализации, способов сохранения и всех остальных функций и характеристик, доступных инженерам. Обеспечив все это, вы можете быть уверены, что система проста для понимания и, следовательно, удобна для получения ответов на запросы и для выявления возможных проблем.

Запомнив эти четыре правила, вы сможете разрабатывать и строить максимально полезные системы мониторинга. Если же обнаружите, что нарушаете их, спросите себя почему. Если у вас не окажется действительно хорошего ответа на этот вопрос, подумайте, как вернуться в исходное состояние.

Структура OpViz

На эту тему мы могли бы написать отдельную книгу. Когда вы начнете собирать и подготавливать данные, необходимые для перехода на OpViz, то вам понадобится уметь распознавать хорошую платформу и отстаивать применение более эффективной платформы. Научить этому — цель данного раздела.

Будем рассматривать платформу OpViz как очень большое устройство распределенного ввода-вывода. Данные передаются, маршрутизируются, структурируются и в итоге выводятся в каком-либо полезном виде, что помогает лучше понимать ваши системы, распознавать поведение, вызванное компонентами, которые вышли или скоро выйдут из строя, и соответствовать заданным SLO. Рассмотрим этот процесс подробнее.

- Данные генерируются агентами[1] на стороне клиента и отправляются в центральный коллектор этого типа данных (показатели в Sensu или CollectD, события в Graphite, журналы в Logstash или Splunk):
  - в некоторых случаях централизованная система мониторинга, например Sensu или Nagios, выполняет проверку методом извлечения в дополнение к упомянутому ранее методу проталкивания;
  - преимущество распределенной проверки — когда приложение генерирует проверки и передает их — в том, что управлять конфигурацией проще и легче, чем в тесно связанных системах, таких как Nagios, требующих совместной настройки и агента, и сервера.
- Коллекторы хранят данные (в таких системах, как Graphite, InfluxDB или ElasticSearch) или пересылают их на маршрутизаторы/процессоры событий, такие как Riemann.
- Маршрутизаторы событий на основании эвристик перенаправляют данные соответствующим получателям.
- Данные могут направляться в долговременное хранилище, представляться в виде графиков, инициировать оповещения, задания (тикеты, ticket) и обращения к другим компонентам.

Именно эти выходные данные и составляют для нас истинную ценность стека OpViz.

Входные данные

Чтобы получить хорошие выходные данные, нам нужны хорошие входные данные. По возможности используйте те, что созданы вашими средами, а не дополнительными искусственными зондами. Подход, при котором вы моделируете пользователя путем отправки запросов в систему, называется *мониторингом черного ящика*. При мониторинге черного ящика привлекаются пользователи-«канарейки»[2] или

[1] Строго говоря, агенты не порождают данные сами по себе, а считывают некоторые показатели функционирования компонентов и передают их в виде данных указанного типа. — *Примеч. науч. ред.*

[2] Их так называют по аналогии с канарейками, которых использовали в шахтах в качестве живых индикаторов утечки газа. — *Примеч. науч. ред.*

анализируются входящие и исходящие данные в интернет-шлюзе. Мониторинг методом черного ящика может быть эффективным для периодов с небольшим трафиком или для компонентов, которые запускаются слишком редко для обычного мониторинга. Но если вы генерируете достаточное количество данных, то получение реальных показателей, то есть *мониторинг белого ящика*, становится неизмеримо более привлекательным. *Тестирование методом белого ящика* подразумевает обширные знания о приложении и в предельном случае наличие программных инструментов, включаемых внутрь приложения. Для этого существуют такие прекрасные инструменты, как AppDynamics, NewRelic и Honeycomb. С их помощью можно отследить поток данных одного конкретного пользователя через все приложение вплоть до базы данных.

ТЕСТИРОВАНИЕ МЕТОДОМ ЧЕРНОГО ЯЩИКА И ТЕОРИЯ МАССОВОГО ОБСЛУЖИВАНИЯ

Учитывая важность величины задержки, мы даже при тестировании методом черного ящика можем использовать теорию массового обслуживания и информацию об объеме трафика и задержках исполнения вызовов, чтобы определить, насколько загружена система. Подробнее о теории массового обслуживания читайте в публикации VividCortex *The Essential Guide to Queueing Theory* (https://www.vividcortex.com/resources/queueing-theory) и на странице университета Нью-Мексико (http://ece-research.unm.edu/jimp/611/slides/chap6_3.html).

Одним из преимуществ этого подхода является то, что все, что создает данные, становится агентом. Проблемы централизованного монолитного решения, которое само генерирует все проверки и зонды, будут расти по мере роста системы. А при тестировании методом белого ящика эта работа распределяется по всей архитектуре. Подобная архитектура также позволяет легко регистрировать и удалять новые сервисы и компоненты на уровне сбора данных, что соответствует правилам OpViz. Вместе с тем бывает полезным и наличие системы мониторинга, способной выполнять удаленные измерения, например, для проверки того, запущен ли сервис, выполняется ли репликация, активирован ли важный параметр хоста базы данных.

ОТДЕЛЕНИЕ СИГНАЛА ОТ ШУМА

Управляя распределенными системами, мы все чаще обращаемся ко все большим наборам данных. Мы уже пришли к тому, что для управления всеми данными, которые собирают наши приложения, и поддерживающей их инфраструктурой необходимо строить системы обработки больших данных. Как уже говорилось в этой главе, до сих пор большой проблемой стеков наблюдаемости остается использование анализа

данных, статистической обработки и прочей «высшей математики». Для эффективного выделения сигнала среди шума необходимо опираться на машинные методы отделения сигнала от шума.

Эта область все еще остается в значительной степени теоретической, и большинство попыток улучшить обнаружение аномалий оказались бесполезными при мультимодальных рабочих нагрузках и постоянных их изменениях из-за той быстроты, с которой разрабатываются новые функции, и из-за переменного состава пользователей. Хорошая система обнаружения аномалий помогает выявить активность, которая не соответствует норме, тем самым сразу направляя внимание персонала на возникшие проблемы, что помогает сократить среднее время восстановления (MTTR) за счет лучшего соотношения «сигнал/шум».

Вот несколько систем, на которые стоит обратить внимание:

- Reimann;
- Anomaly.io;
- VividCortex;
- Dynatrace;
- Circonus;
- Prometheus.

Теперь мы намерены отправить все эти ценные сведения в нашу платформу OpViz. О каких именно данных идет речь?

Телеметрия/показатели

Ах, эти показатели! Такие разнообразные. Такие вездесущие. Показатель — это результат измерения характеристики, которой обладает приложение или компонент инфраструктуры. Показатели измеряются периодически, создавая временные ряды, каждый элемент которых содержит значение одной или нескольких характеристик, временную метку и значение. Характеристики могут относиться к отдельному хосту, сервису или центру обработки данных. Истинная ценность этих данных заключается в том, что можно исследовать их изменения с течением времени с помощью визуализации, например, в виде графиков.

Обычно применяются следующие четыре формы хранения и представления показателей.

- *Счетчики*. Это накопительные показатели, которые показывают, сколько раз произошло конкретное событие.

- *Датчики.* Это показатели, которые могут как увеличиваться, так и уменьшаться, и показывают текущее значение, например температуру, количество заданий в очереди или активных блокировок.
- *Гистограммы.* На них множество событий разбито на сегменты определенной конфигурации, чтобы показать их распределение.
- *Сводные графики.* Похожи на гистограммы, но отражают результаты проверки счетчиков в скользящих временных окнах.

К показателям часто применяются математические функции, помогающие извлекать из них ценную информацию. Эти функции очень полезны, но важно помнить, что результат их вычисления — вторичная информация, поэтому важны также и первичные (необработанные) данные. Если вы контролируете средние значения за 1 минуту, но не располагаете исходными данными, то не сможете получить средние значения для более длительных интервалов, таких как часы или дни. Далее приведены некоторые из функций:

- счетчик;
- сумма;
- среднее значение (математическое ожидание);
- медиана;
- процентили;
- стандартное (среднеквадратичное) отклонение;
- быстрота изменения;
- закон распределения.

Визуализация распределений

Визуализация распределения очень важна, когда надо сориентироваться в тех данных, которые обычно генерируются в веб-архитектурах. Такие данные редко распределены по нормальному закону и часто имеют длинные «хвосты». Это бывает сложно увидеть на обычных графиках. Но благодаря возможности генерировать карты распределения за заданные периоды времени становятся доступными новые способы визуализации, такие как временные гистограммы и тепловые карты, которые действительно могут помочь оператору-человеку наглядно показать рабочие нагрузки, возникающие в ваших системах[1].

Показатели служат основой для выявления симптомов и стоящих за ними скрытых первопричин. Таким образом, они играют ключевую роль в раннем выявлении и быстром решения всевозможных проблем, которые могут повлиять на SLO.

[1] Тепловые карты величин задержки.

События

Событие — это отдельное действие, которое произошло в среде. Изменение в конфигурации — это событие. Развертывание кода — тоже событие. Восстановление после отказа основного экземпляра базы данных — еще одно событие. Каждое из них может послужить ключом для соотнесения симптомов с их причинами (первичными проблемами).

Журнал событий

События заносятся в журнал (log), и мы можем рассматривать журнал как подмножество событий. Операционные системы, базы данных и приложения — все они протоколируют соответствующие события, занося их в журналы. Журналы могут представлять дополнительные, не отражаемые показателями сведения в контексте того, что происходило в системе. Например, запись в журнале о запросе к базе данных может сообщить, когда был сделан запрос, важные параметры этого запроса и даже пользователя базы данных, который его сделал.

Выходные данные

Итак, данные поступают в системы, и это хорошо, но это не помогает нам отвечать на вопросы или обеспечивать соответствие SLO, не так ли? Что нам нужно создать в нашей среде OpViz? Разберем данный вопрос более пристально.

- *Оповещения*. Оповещения прерывают работу человека, требуя отвлечься от того, что он делает, и разбираться с нарушением правил, которое и привело к оповещению. Это дорогостоящая операция, и ее следует использовать только тогда, когда имеется непосредственная угроза нарушения SLO.
- *Задания*. Задания генерируются в тех случаях, когда нужно выполнить некую работу, но неизбежной катастрофы пока нет. Результатом мониторинга должны быть задания, которые поступают в очередь и которые инженеры должны обработать.
- *Уведомления*. Иногда нужно просто отметить, что событие произошло, чтобы держать в курсе тех людей, чья работа связана с ним, например запротоколировать события развертывания кода. Уведомления часто отправляются в дискуссионную группу, в вики или в систему поддержки совместной работы, где они будут видны, но не прервут рабочий процесс.
- *Автоматизация (автоматическая обработка)*. Бывает, что данные мониторинга, особенно коэффициент использования системы, указывают на необходимость увеличить или уменьшить количество предоставляемых ресурсов.

В таких случаях можно обратиться к автоматическому масштабированию для изменения пулов ресурсов. Это всего лишь один пример автоматизации, инициируемой мониторингом.

- *Визуализация*. Одной из наиболее общих и часто используемых форм результатов OpViz являются графики. Они объединяются в информационные панели, соответствующие потребностям конкретного сообщества пользователей, и служат ключевым инструментом, с помощью которого выявляются зависимости в данных.

Первоначальный запуск мониторинга

Если вы подобны большинству людей разумных, то, возможно, чувствуете себя подавленными из-за того, сколько всего должно происходить. Это нормально! Сейчас самое время напомнить, что все, что мы здесь строим, — постепенный процесс. Начните с малого, позвольте событиям развиваться и добавляйте их по мере необходимости. Это особенно верно при запуске среды.

Как при любом начальном запуске, вы начинаете с нуля. Никаких показателей, никаких оповещений, никаких результатов мониторинга — только группка программистов, которые строчат не в меру оптимистичный код. Многие стартапы доводятся так или иначе до состояния экземпляра, годящегося в качестве прототипа или тестового стенда, где-нибудь в публичном облаке, а потом он каким-то образом превращается в работающую «промышленную» базу данных. Голова? Ударьте ею об стол!

Возможно, вы только что пришли в молодой стартап в качестве первого (и пока единственного) инженера по эксплуатации и базам данных, и вы взираете на то, что разработчики сделали в направлении мониторинга или контроля. И это фактически... ничего.

Выглядит знакомо? Если у вас есть опыт работы со стартапами, то вы понимаете, что так и должно быть. Здесь нечего стыдиться! Именно так и строится стартап. Было бы глупо начинать его с построения сложной системы оперативного контроля, прежде чем появятся реальные потребности в ней. Чтобы стартап стал успешным, нужно сосредоточиться на основном продукте, быстро и настойчиво разыскивая клиентов, реагируя на их отзывы и реалии производства и принимая непростые решения о том, на что направить драгоценные инженерные ресурсы. Стартап станет успешным, если сложные системы контроля производительности будут внедряться только тогда, когда потребуются, но не раньше. Многие стартапы терпят неудачу, но обычно не потому, что инженеры не смогли спрогнозировать все возможные параметры хранилища. Для начала нужно установить *минимальный необходимый набор мониторинга* (Minimum Viable Monitoring Set).

ПОТОКИ ДАННЫХ В БАЗАХ

Вы можете рассматривать данные как поток, поступающий от клиентов к базе данных. На самом высоком уровне БД существует для приема, хранения и обработки данных:

- в памяти клиента;
- передаваемых по линиям связи между клиентом и хранилищем данных;
- в структурах базы данных;
- в ОС и структурах дисковых массивов;
- на ваших дисках;
- в резервных копиях и архиве.

Все возможные измерения в базах данных можно свести к поиску ответов на следующие вопросы.

- Сколько времени занимает получение данных и почему так много?
- Сколько времени занимает ввод данных и почему так много?
- Безопасно ли хранятся данные и как именно они хранятся?
- Доступны ли данные в дублирующих хранилищах на случай сбоя основной базы?

Это, конечно, довольно упрощенное представление, но оно неплохо отражает реальную картину при взгляде сверху, и ее стоит иметь в виду, когда мы углубимся в следующие разделы.

Существует бесконечно много показателей базы данных, системы, хранилища и различных приложений прикладного уровня, которые можно контролировать. Условно говоря, на уровне «физиологические потребности» пирамиды Маслоу можно определить, работает база данных или нет. Перейдя на уровень «уважение», вы начинаете заниматься другими обнаруженными симптомами, связанными с реальными проблемами, такими как количество подключений или процент блокировок. В целом иерархия выглядит так.

1. Мониторинг того, работают базы данных или отключены (pull-проверки).
2. Мониторинг общих показателей задержек/ошибок и сквозной мониторинг работоспособности (push-проверки).
3. Инструментарий прикладного уровня для измерения задержек/ошибок для каждого вызова базы данных (push-проверки).
4. Сбор как можно большего количества показателей, относящихся к уровням системы, хранилища, базы данных и приложений, независимо от того, будут они, по вашему мнению, полезны или нет. Для большинства операционных систем, сервисов и БД это более или менее обеспечивается подключаемыми модулями.

5. Специальные проверки для отдельных известных проблем. Например, проверки поведения при отключении *x* процентов узлов базы данных или слишком высокого процента глобальных блокировок (это следует делать непрерывно и превентивно, см. главу 3).

Иногда можно быстрее перейти на уровень «уважение», подключив сторонние сервисы мониторинга, такие как VividCortex, Circonus, HoneyComb или NewRelic. Но если вы храните эти показатели базы данных в системе отдельно от прочих результатов мониторинга, то это своего рода жульничество. Хранение данных в разнородных системах усложняет соотнесение симптомов, полученных от нескольких платформ мониторинга. Мы не хотим сказать, что это плохо или этого не следует делать, — случается, что элегантные уловки и хаки живут очень долго! Но помните, что уровень «самореализация» обычно включает в себя объединение всех каналов мониторинга в единый источник, показывающий истинную картину.

Итак, теперь, защитив компанию от банкротства в случае потери диска или механической ошибки инженера, можно начать думать о состоянии сервиса. Вот ключевые вопросы, которые следует задать себе с самого начала: «Надежно ли хранятся данные?», «Работает ли сервис?», «Испытывают ли клиенты трудности?». Это минимальный жизненно необходимый набор вопросов для мониторинга продукта.

Безопасны ли данные?

Работая с любыми критически важными данными, сохранность которых вас действительно волнует, необходимо избегать оставаться с менее чем тремя «живыми» экземплярами базы: одним основным и не менее чем двумя запасными, если речь идет о дублирующихся хранилищах данных на MySQL или MongoDB. При использовании распределенных хранилищ данных, таких как Cassandra или Hadoop, коэффициент репликации должен быть равен 3. Ведь вы никогда, ни за что, ни в коем случае не захотите оказаться в ситуации, когда есть только одна копия всех интересующих вас данных. А это означает, что вы должны быть готовы потерять один экземпляр, сохранив при этом избыточность. Поэтому минимальное количество копий не две, а три. Даже если ваш стартап только-только запустился и вы считаете каждую копейку, подводя баланс каждый месяц, критически важные данные не та статья расходов, за счет которой можно пытаться сэкономить (подробнее о проектировании архитектуры и обеспечении доступности читайте в главе 5).

Но не все данные одинаково полезны! Если вы можете позволить себе потерять некоторые данные или способны при необходимости воссоздать их из неизменяемых журналов, то запуск с $n + 1$ экземплярами (где n — количество узлов, необходимое для нормальной работы) — это нормально. Это ваш личный выбор — только вам известно, насколько важен и незаменим каждый набор данных для компании и насколько ограниченны ее финансы. Вам также нужны резервные копии, и необходимо регулярно проверять, что они работоспособны и что процесс резервного

копирования завершается успешно. Если не контролировать, в порядке ли резервные копии, нельзя быть уверенными, что данные в безопасности.

Примеры мониторинга безопасности

Вот несколько примеров проверок безопасности, которые нужно включить в программу мониторинга:

- число действующих узлов не менее трех;
- функционируют потоки, ответственные за репликацию[1];
- как минимум на одном узле репликация выполнялась не более 1 секунды назад;
- последнее резервное копирование было успешным;
- последнее автоматическое восстановление реплик из резервных копий выполнено успешно.

Работает ли сервис?

Сквозные проверки — самый мощный инструмент из вашего арсенала, поскольку они наиболее точно отражают ваш опыт работы с клиентами. У вас должна выполняться проверка работоспособности верхнего уровня, которая проверяет не только работоспособность на уровне веб-сервисов и приложений, но и все соединения с базой данных, участвующие в критических путях. Если данные распределены между несколькими хостами, проверка должна выбирать объекты в каждом из разделов, при этом автоматически определяя полный список разделов и сегментов, чтобы не приходилось вручную добавлять новые проверки каждый раз при масштабировании.

Однако — и это важно! — у вас должна быть и более простая проверка работоспособности для балансировщиков нагрузки, которая занимала бы не все соединения с базой данных. В противном случае проверка способна легко убить систему.

Чрезмерная проверка жизнеспособности

Однажды Черити работала над системой, где в качестве проверки балансировщика HAProxy использовался простой запрос SELECT LIMIT 1 для таблицы mysql. Как-то раз в системе удвоили емкость некоторых сервисов, не имеющих хранения результатов запросов, удвоив таким образом количество прокси-серверов, выполняющих проверки работоспособности. Увеличение емкости других систем внезапно вывело из строя весь сайт, так как серверы БД оказались перегружены этими проверками. Более 95 % запросов к базе данных оказались лишь проверками работоспособности. Не делайте так!

[1] Имеются в виду вычислительные потоки (threads) сервисов и/или приложений. — *Примеч. науч. ред.*

Кстати об уроках, полученных столь нелегким путем: у вас всегда должен быть какой-то локальный мониторинг — на худой конец хотя бы внешняя проверка работоспособности самого сервиса мониторинга. Не имеет значения, насколько совершенной и надежной будет локальная система мониторинга, если центр обработки данных или облако выйдет из строя и потянет за собой весь аппарат мониторинга. Хорошей практикой является внешняя проверка каждого основного продукта или сервиса, а также проверка работоспособности самого сервиса мониторинга.

Примеры объектов мониторинга доступности БД

Вот несколько способов проверки того, доступна ли система, или как скоро она перестанет быть доступной:

- проверка работоспособности на уровне приложения, которая запрашивает все внешние хранилища данных;
- выполнение запроса для каждого раздела в каждом элементе хранилища данных для всех хранилищ данных;
- проверка приближающихся проблем с емкостью и пропускной способностью:
  - емкость диска;
  - соединения с базой данных;
- проверка по журналу ошибок:
  - перезапуск БД (быстрее, чем выполняется мониторинг!);
  - проверка повреждений.

Испытывают ли клиенты трудности?

Итак, мониторинг показывает, что ваш сервис жив. Сердце пациента бьется. Мы молодцы!

Но что, если задержка удваивается или утраивается или 10 % запросов заканчиваются ошибкой, из-за чего проверка жизнеспособности не включается? Что, если ваша база данных доступна только для чтения, но не для записи или репликация запаздывает, из-за чего зависает большинство операций записи? Что, если в RAID-массиве потерян том и массив работает в ограниченном режиме, а в это время происходит построение индекса или выполняются горячие обновления отдельной записи?

Именно поэтому так интересно проектировать системы, в частности базы данных. Существует множество способов выхода системы из строя, и всего около 5 % из них вы, скорее всего, сможете предвидеть заранее. Круто!

Вот почему следует постепенно разрабатывать библиотеку всесторонних высокоуровневых показателей состояния сервиса: проверки работоспособности, частоты ошибок, задержек. Все, что существенно влияет на него и ухудшает качество обслуживания клиентов. А что потом? Переключиться на что-нибудь еще и смотреть, что еще повреждено.

Мы почти не шутим. Как говорилось в главе 3, вы едва ли много выиграете, пытаясь заранее угадать, как будет разрушен ваш сервис. У вас просто еще нет для этого данных. За это время вы можете создать множество разных вещей, подождать, пока что-то прекратит работать, а затем уделить внимание тому, что действительно начнет выходить из строя.

Теперь, когда мы рассказали вам о методах начального запуска развития, разберем, что следует измерять, сделав акцент на том, что из этого нужно DBR-инженеру.

Оснащение приложения

Начинать следует именно с приложений. Большинство измерений можно выполнить на уровне хранилища данных, но первыми, основными индикаторами проблем должны быть изменения поведения пользователей и приложений. Благодаря выполняемому вашими инженерами оснащению приложений инструментами и наличию решений для управления производительностью приложений (APM), таким как New Relic и AppDynamics, можно получить огромный объем данных, полезных для всей организации.

- К этому времени вы уже должны измерять и регистрировать все запросы и отклики на страницах и в конечных точках API.
- Вы должны также делать это для всех внешних сервисов, включая базы данных, поисковые индексы и кэши.
- Аналогичным образом должны контролироваться любые задания и независимые рабочие процессы.
- Точно так же должен быть оснащен любой независимый многократно используемый код, такой как метод или функция, который взаимодействует с базами данных, кэшами и другими хранилищами.
- Необходимо отслеживать, сколько обращений к базе данных выполняется каждой конечной точкой, страницей, функцией или методом.

Отслеживание поведения кода для доступа к данным (например, вызовов SQL), выполняемого при каждой операции, позволяет быстро создавать перекрестные ссылки на более подробные журналы запросов к базе данных. Это может оказаться сложной задачей для систем объектно-реляционного отображения (Object-Relational Mapping Systems, ORM), где SQL генерируется динамически.

Комментарии в SQL

При настройке SQL большой проблемой является установление соответствия между SQL-кодом, выполняемым в базе данных, и конкретным местом в базе кода, откуда он вызывается. Во многих системах управления базами данных можно вставлять комментарии с дополнительной информацией. Они будут отображаться в журналах запросов к БД. Комментарии — отличное место для вставки информации о локализации в базе кода.

Контроль выполнения в распределенных системах

Контроль производительности на всех ступенях, от приложения до хранилища данных, имеет важнейшее значение при решении задачи оптимизации в случае длительных задержек, что бывает очень непросто. Системы New Relic или Zipkin (с открытым исходным кодом) позволяют выполнять трассировку в распределенных системах от вызовов в приложениях до внешних сервисов, таких как базы данных. Полная трассировка транзакции от приложения до хранилища данных в идеале должна давать информацию о времени для всех вызовов внешних сервисов, а не только для запроса к базе данных.

Полная трассировка базы данных может стать мощным средством для обучения инженеров-программистов (SWE) и достижения автономности и самодостаточности. Вам не придется указывать разработчикам, на что им следует обратить внимание, они сами смогут получить эту информацию. Как сказал Аарон Мортон (Aaron Morton) в своем выступлении «Заменить Cassandra Tracing на Zipkin» на Last Pickle, «предсказать заранее, какие инструменты создадут столь позитивные культурные сдвиги, в принципе невозможно, но я использовал Git и его практики запросов на внесение изменений и стабильных мастер-веток, а также Grafana, Kibana и Zipkin» (подробнее об этом читайте в блоге The Last Pickle по адресу https://thelastpickle.com/blog/2015/12/07/using-zipkin-for-full-stack-tracing-including-cassandra.html).

Выполнение вызова состоит из многих стадий, и они могут представлять интерес для DBRE. Список включает (но не ограничивается ими!) следующие:

- установление соединения с базой данных или с прокси базы данных;
- постановку в очередь на соединение с базой данных в пуле соединений;
- запись показателя или события в сервис очередей или сообщений;
- создание идентификатора пользователя из централизованного сервиса UUID;
- выбор сегмента базы в зависимости от значения переменной (например, ID пользователя);

- поиск, кэширование или сброс кэширования на уровне кэша;
- сжатие или шифрование данных на прикладном уровне;
- запрос уровня поиска.

ТРАДИЦИОННЫЙ SQL-АНАЛИЗ

Привет, это Лейн. Не счесть сколько раз, занимаясь консультированием, я приходила в магазин, где не было мониторинга, который выявлял бы соответствие между мониторингом производительности приложений и базы данных. Для создания представления (view) из базы данных мне неизменно приходилось собирать данные из журналов TCP и SQL-запросов. Затем, составив список приоритетных SQL-запросов для оптимизации, я возвращалась к SWE, и они не знали, в каком месте находится код, который нужно исправить. Поиск в базах кода может занять неделю драгоценного времени или даже больше.

У вас, как у DBRE, есть прекрасная возможность работать бок о бок с SWE, чтобы гарантировать, что все классы, методы, функции и задания имеют прямые соответствия с вызываемым SQL-запросом. Когда SWE и DBRE используют одни и те же инструменты, DBR-инженеры могут указать на основные критические точки и вскоре вы обнаружите, что программисты выполняют работу за вас!

Если у транзакции есть бюджет производительности и известны требования к задержкам, то сотрудники, ответственные за каждый компонент, будут работать совместно, чтобы определить главные источники затрат, сделать соответствующие вложения и достигнуть компромиссов, чтобы удовлетворить всем требованиям.

События и журналы

Само собой разумеется, что все журналы приложений должны собираться и храниться, включая трассировки стека вызовов. Кроме того, будут происходить многочисленные события, весьма полезные для OpViz, которые должны быть запротоколированы. В их числе:

- развертывание кода;
- измеренное время развертывания;
- ошибки, возникшие при развертывании.

Мониторинг приложений важен как первый шаг к реалистичному взгляду на поведение приложения с позиции пользователя, что напрямую связано с SLO задержки. Эти симптомы дают ключ к пониманию ошибок и деградации системы. Теперь мы посмотрим, какие еще данные помогут нам выполнить анализ и выявить первопричины. Это данные хоста.

Оснащение серверов и экземпляров баз данных

Теперь займемся отдельным хостом, реальным или виртуальным, на котором находится экземпляр базы данных. Именно здесь можно получить все данные об операционной системе и физических ресурсах, предназначенных для работы наших БД. Несмотря на то что эта информация не связана с конкретным приложением или сервисом, она бывает полезна, если наблюдаются такие симптомы, как задержка или ошибки на уровне приложения.

При использовании этой информации для выявления причин аномалий в приложениях цель состоит в том, чтобы найти, какие из ресурсов используются слишком часто или слишком редко, недостаточно загружены или выдают ошибки (Underutilized, Saturated or throwing Errors — USE, как определил Брендан Грегг (Brendan Gregg) в своей методологии: http://www.brendangregg.com/usemethod.html). Эта информация также имеет решающее значение при планировании мощностей для масштабирования и оптимизации производительности. Выявление узких мест и ограничений позволяет расставить приоритеты при оптимизации, чтобы добиться максимальной отдачи.

Агрегирование для распределенных систем

Имейте в виду, что сама по себе информация о хосте не особенно полезна, разве что как показатель того, что с хостом что-то не так и его нужно отключить. Вместо этого подумайте об агрегировании сведений о коэффициенте использования, загрузке и ошибках, получаемых для пула хостов, реализующих одну и ту же функцию. Другими словами, если есть 20 хостов Cassandra, то вас больше всего будут интересовать общий коэффициент использования пула, время ожидания (загруженность) и возникающие ошибки. Если ошибки локализованы на одном хосте, надо просто удалить его из пула и заменить новым.

В операционной системе Linux хорошими контрольными точками для мониторинга ресурсов являются:

- процессор;
- память;
- сетевые интерфейсы;
- вход/выход хранилища данных;
- емкость хранилища данных;
- контроллеры хранилища данных;
- сетевые контроллеры;
- интерфейсы между процессорами;
- интерфейсы между блоками памяти;
- интерфейсы между хранилищами данных.

Роль операционной системы

Невозможно переоценить важность глубокого изучения рабочих характеристик операционной системы. Многие специалисты по базам данных предоставляют это системным администраторам, однако между уровнем сервисов базы данных и операционной системой существует слишком тесная связь, чтобы оставить ее без внимания. Прекрасным примером является то, как Linux заполняет весь страничный кэш и, таким образом, количество свободной памяти в качестве показателя для контроля ее использования оказывается практически бесполезным. В этом случае количество просмотров страниц в секунду становится гораздо более полезным показателем, что неочевидно, если нет более глубокого понимания того, как работает управление памятью в Linux.

Кроме мониторинга аппаратных ресурсов, в операционной системе есть еще несколько позиций для контроля:

- блокировки на уровне ядра;
- пользовательские блокировки;
- пул выполняющихся задач (процессов);
- дескрипторы файлов.

Если вы плохо знакомы с этой областью, предлагаем обратиться к странице Брендана Грегга *USE page for Linux* (http://www.brendangregg.com/usemethod.html), где весьма детально описана организация контроля этих показателей. Очевидно, что вам придется затратить немало времени и усилий на усвоение того, что там представлено.

События и журналы

В дополнение к показателям следует отправлять в соответствующую систему обработки событий, такую как *RSyslog* или *Logstash*, и все записи журналов. Это журналы ядра, планировщика, аутентификации, почты и обычных сообщений, а также журналы, относящиеся к процессам и приложениям, например MySQL или nginx.

Процессы, управляющие конфигурацией и обеспечивающие планирование, также должны протоколировать в журналах критические события в стеке OpViz. Вот неплохой для начала список таких событий:

- выход хоста из строя;
- изменение конфигурации;
- перезагрузка хоста;
- перезапуск сервиса;
- сбой хоста;
- сбой сервиса.

ОБЛАЧНЫЕ И ВИРТУАЛИЗИРОВАННЫЕ СИСТЕМЫ

В подобных средах необходимо учитывать несколько дополнительных моментов.

Стоимость! В этих средах вы тратите деньги по факту, а не авансом, как, возможно, привыкли в центре обработки данных. Здесь рентабельность и эффективность имеют решающее значение.

При мониторинге процессора следите за временем недоступности. Это время, когда виртуальный процессор ожидает реального процессора, который задействован в других задачах. Длительное время недоступности (10 % и более в течение продолжительных периодов) является показателем того, что в среде есть шумный сосед! Если время недоступности одинаково для всех ваших хостов, то причина, вероятно, заключается в вас и, скорее всего, придется выделить больше ресурсов и/или изменить распределение нагрузки.

Если время недоступности велико только на одном или нескольких хостах, значит, другой арендатор крадет ваше время! Лучше всего удалить этот хост и запустить новый. И надеяться, что он будет развернут в другом месте и станет работать намного лучше.

Если вы сможете внедрить описанное ранее в свой стек OpViz, то получите отличную возможность понять, что происходит на уровне хоста и операционной системы. Теперь рассмотрим сами базы данных.

Оснащение хранилища данных

Что требует мониторинга, что следует отслеживать в базах данных и почему? Отчасти это зависит от типа хранилища. Здесь мы обратим внимание на группы, достаточно общие, чтобы быть универсальными, но при этом достаточно специфичные, чтобы помочь вам разобраться с собственными базами данных. Мы можем разбить все параметры на следующие четыре группы:

- уровень соединения с хранилищем данных;
- контроль внутри базы данных;
- объекты базы данных;
- вызовы/запросы к базе данных.

Каждой из этих групп мы посвятим свой раздел и начнем с уровня подключения к хранилищу данных.

Уровень соединения с хранилищем данных

Мы уже обсудили важность контроля времени, которое требуется для подключения к внутреннему хранилищу данных в рамках транзакции. Кроме того, система мониторинга должна быть в состоянии выделить время обмена данными с прокси и время прохождения данных от прокси до серверной части. Вы можете периодически получать эти значения с помощью утилиты TCPdump и Tshark/Wireshark, если у вас нет чего-то вроде Zipkin. Этот процесс можно автоматизировать для произвольных выборок или запускать по мере надобности.

В случае задержек и/или ошибок между приложением и соединением с базой данных потребуются дополнительные показатели, которые помогут определить причины. Взяв для примера рекомендованный нами ранее метод USE, посмотрим, какие еще показатели могут нам помочь.

Коэффициент использования

Базы данных позволяют поддерживать только конечное число соединений. Максимальное количество соединений ограничено для всех локаций базы. В соответствии с параметрами конфигурации база данных сможет принимать только определенное количество соединений, устанавливая искусственную верхнюю границу для минимизации риска перегрузки хоста. Контроль этого максимума, как и фактического количества соединений, имеет решающее значение, поскольку по умолчанию он может быть установлен произвольно низким.

Соединения открывают доступ к ресурсам также и на уровне операционной системы. Например, в PostgreSQL для каждого соединения используется один процесс Unix, в MySQL, Cassandra и MongoDB — один поток (thread). Всем им требуются память и файловые дескрипторы. Итак, вот несколько моментов, на которые стоит обратить внимание, чтобы понять поведение соединений:

- верхняя граница числа соединений и их реальное количество;
- состояние соединения (работает, спит, прервано и др.);
- коэффициент использования открытого файла на уровне ядра;
- максимальный коэффициент использования процессов на уровне ядра;
- коэффициент использования памяти;
- показатели пула потоков, такие как коэффициенты использования кэша таблиц MySQL или пула потоков MongoDB;
- коэффициент использования пропускной способности сети.

Эти данные должны сообщать вам о дефиците ресурсов или о перегрузке на уровне соединений. Если окажется, что коэффициент использования составляет 100 %

и загруженность тоже высока, то это хороший показатель. Но низкая загрузка в сочетании с высокой загруженностью говорит о том, что где-то есть узкое место. Высокий, но не стопроцентный коэффициент использования ресурсов также часто серьезно влияет на величину задержки и может быть ее причиной.

Загруженность

Уровень загрузки бывает наиболее полезен в сочетании с коэффициентом использования. Если у ресурсов, задействованных на 100 %, большое время ожидания, это указывает на очевидную проблему с емкостью. Но если есть задержки и перегрузка без полного использования, это может означать, что где-то есть узкое место, которое вызывает коллизии. Загруженность может быть измерена в таких типичных точках, как:

- журнал TCP-соединений;
- очередь соединений для конкретной базы данных, например, `back_log` в MySQL;
- превышение времени ожидания соединения;
- ожидание потоков в пулах соединений;
- подкачка страниц памяти;
- блокировки процессов в базе данных.

Для распознавания перегрузок решающее значение имеют длина очереди и время ожидания. Каждое обнаруженное ожидание соединения или процесса — это индикатор потенциального узкого места.

Ошибки

Благодаря коэффициенту использования и показателю загруженности мы можем определить, влияют ли ограничения ресурсов и узкие места на задержку на уровне соединения с базой данных. Это отличная информация, позволяющая принять решение о том, нужно ли расширять доступные ресурсы, снимать искусственные ограничения конфигурации или вносить изменения в архитектуру. Необходимо также отслеживать ошибки и использовать сведения о них для выявления и устранения неисправностей и/или проблем конфигурации. Ошибки можно зафиксировать различными способами.

- Журналы базы данных предоставляют коды ошибок, возникающих при сбоях на уровне базы данных. Иногда встречаются конфигурации с различной степенью детализации. Убедитесь, что у вас включен режим достаточно подробного ведения журнала, чтобы идентифицировать ошибки подключения, но не забудьте о дополнительных затратах на протоколирование, особенно если для

журналов используются те же хранилище данных и ресурсы ввода-вывода, что и для базы данных.

- Журналы приложений и прокси-серверов также предоставляют информацию о различных ошибках.
- Могут пригодиться и ошибки хоста, описанные в предыдущем разделе.

К ошибкам относят ошибки сети, превышения тайм-аутов при соединении, ошибки аутентификации, разрывы соединений и многое другое. Они могут указывать на такие проблемы, как поврежденные таблицы, проблемы с DNS, взаимные блокировки, изменения аутентификационных данных и т. д.

Используя показатели задержек/ошибок в приложениях, трассировку и соответствующую телеметрию коэффициента использования, сведения об уровне перегрузки и отдельных специфических ошибках, вы получите информацию для выявления нарушений или ограничения работоспособности на уровне соединения с базой данных. В следующем разделе мы рассмотрим, что можно измерять внутри соединений.

УСТРАНЕНИЕ ПРОБЛЕМ СКОРОСТИ СОЕДИНЕНИЯ ДЛЯ POSTGRESQL

Instagram — одна из компаний, которые выбрали PostgreSQL в качестве реляционной базы данных. Она решила использовать пул соединений PGBouncer, чтобы увеличить количество соединений для приложений, которые могут подключаться к базам данных. Это проверенный механизм масштабирования для увеличения количества соединений с хранилищем данных, и, учитывая, что PostgreSQL порождает новый процесс Unix для каждого соединения, создание дополнительных соединений оказывается медленным и дорого обходится.

Используя драйвер Python psycopg2, компания работала со стандартным значением `autocommit = FALSE`. Это означает, что даже для запросов, требующих только чтения, применялись явные `BEGIN` и `COMMIT`. Изменив параметр `autocommit` на `TRUE`, компания сократила время выполнения запросов, что уменьшило также длину очереди к пулу соединений.

Первоначально могло показаться, что задержки в приложении вызваны стопроцентным использованием пула, что приводит и к увеличению очередей. Из анализа показателей на уровне соединения и мониторинга пулов pgbouncer стало ясно, что пул *ожидания* увеличивается вследствие перегрузки, и пул *активных* процессов большую часть времени полностью загружен. Не имея других показателей, отражающих высокий коэффициент использования, загруженность и явные ошибки, нужно было посмотреть, что происходит внутри соединения, в котором наблюдается замедление запросов. Мы обсудим это в следующем подразделе.

Контроль внутри базы данных

Заглянув внутрь базы данных, мы увидим, что количество «шестеренок», различных показателей и вообще сложность базы существенно возросли. Другими словами, именно здесь происходит встреча с реальностью! Кроме того, не забываем о USE. Наша цель — найти узкие места, которые могут влиять на задержки, ограничивать запросы или вызывать ошибки.

Важно иметь возможность рассматривать это как с точки зрения отдельного хоста, так и с агрегацией по ролям. Некоторые базы данных, такие как MySQL, PostgreSQL, ElasticSearch и MongoDB, предусматривают отдельные роли мастера и реплики. В Cassandra и Riak нет конкретных ролей, но они часто бывают распределены по регионам или зонам. Это тоже имеет значение при сборе данных.

Показатели пропускной способности и задержки

Сколько операций и каких именно выполняется в хранилищах данных? Эти сведения позволяют составить полное представление о работе базы. По мере того как разработчики будут добавлять новые функции, рабочие нагрузки станут смещаться, и вы получите хорошие показатели их изменения. Вот несколько примеров показателей, которые необходимо собирать, чтобы понять, каковы рабочие нагрузки:

- операции чтения;
- операции записи:
  - вставки;
  - модификации;
  - удаления;
- другие операции:
  - фиксация (подтверждение) изменений;
  - откаты изменений;
  - использование DDL-операторов;
  - другие административные задачи.

Говоря о задержке, здесь мы имеем в виду только совокупную задержку и усредненные значения. Позже в этом подразделе мы рассмотрим детальный и более информативный мониторинг запросов. Пока же в таком представлении данных вы не получите выбросов — лишь основную информацию о рабочей нагрузке.

Фиксация изменений, откат операций и журналирование

Конкретные реализации зависят от хранилища данных, но почти всегда присутствует набор операций ввода-вывода, связанных с выгрузкой данных на диск. В системе управления базами данных InnoDB, MySQL и PostgreSQL операции записи производят изменения только в буфере (в памяти), а сами записываются в журнал откатов (или журнал упреждающей записи в PostgreSQL). Затем эти данные выгружаются на диск в фоновом режиме с сохранением контрольных точек для восстановления. В Cassandra данные хранятся в memtable (в памяти), к ним добавляется журнал фиксации изменений. Таблицы memtable периодически выгружаются в SSTable. Таблицы SSTable также время от времени уплотняются. Далее приведены показатели, которые можно контролировать:

- «грязные» буферы (MySQL);
- «возраст» контрольной точки (MySQL);
- ожидающие и завершенные задачи уплотнения (Cassandra);
- отслеживаемые «грязные» байты (MongoDB);
- вытесненные модифицированные и немодифицированные страницы (MongoDB);
- конфигурация `log_checkpoints` (PostgreSQL);
- представление `pg_stat_bgwriter` (PostgreSQL).

Все использования контрольных точек, очистка и уплотнение — это операции, которые серьезно сказываются на производительности базы данных. Иногда их влияние состоит в увеличении количества операций ввода-вывода, а иногда может стать причиной полной остановки всех операций записи во время выполнения основной операции. Здесь сбор показателей позволяет настраивать конкретные конфигурируемые параметры, чтобы свести к минимуму эффекты, которые будут возникать во время таких операций. Таким образом, в данном случае, заметив увеличение задержки и увидев, какие показатели, связанные с выгрузками данных, показывают чрезмерную фоновую активность, мы сможем понять, настройки каких операций могут быть связаны с этими процессами.

Состояние репликации

Репликация — это копирование данных на нескольких узлах таким образом, чтобы данные на одном узле сделать идентичными данным на других узлах. Репликация — краеугольный камень обеспечения доступности данных и масштабирования чтения, а также часть восстановительных мероприятий после аварий и одна из мер по обеспечению безопасности данных. Однако при репликациях возможны три си-

туации, которые могут вызвать большие проблемы, если их вовремя не отслеживать и не предотвращать. Мы подробно обсудим репликацию в главе 10.

Задержка репликации — первая из возможных проблем. Иногда применение изменений к другим узлам может замедляться. Это может быть результатом загруженности сети, использования однопоточных приложений, которые не успевают за темпом репликаций[1], или по ряду других причин. Бывает, что во время пиковой активности репликация вообще не запускается, из-за чего данные реплик оказываются не согласованы несколько часов. Это опасно, так как в обработку могут попасть устаревшие данные, и если придется переключиться на эту реплику как на резервную, то более свежие данные могут быть потеряны.

В большинстве систем управления базами данных есть легко отслеживаемые показатели задержки репликации. Кроме того, вы можете увидеть разницу между временными метками на главном сервере и на реплике. В таких системах, как Cassandra, при использовании *модели отложенной целостности* (eventually consistent model) можно обратиться к журналу невыполненных операций для синхронизации реплик по окончании периода недоступности. Например, в Cassandra это hinted handoff — «направленная отправка», механизм сохранения, задействующий соседние узлы.

Вторая проблема — прерванная репликация. Процессы, реализующие репликацию данных, просто прерываются из-за любых возникших ошибок. Решать эту проблему нужно быстро, чему способствует мониторинг, с последующим устранением причины ошибок и возобновлением репликации, позволяющей наверстать упущенное. В этом случае можно отслеживать состояние потоков репликации.

И наконец, последняя проблема — дрейф репликации — наиболее коварна. В этом случае данные становятся несогласованными, что делает репликацию бесполезной и потенциально опасной. Обнаружение дрейфа репликации для больших наборов данных может оказаться сложной задачей, решение которой зависит от рабочих нагрузок и вида сохраняемых данных.

Например, если данные относительно неизменны и нормальными для них операциями являются вставка и чтение, можно запустить подсчет контрольных сумм для массивов данных в разных репликах, а затем сравнить контрольные суммы, чтобы убедиться в их идентичности. Это можно сделать при развертывании после репликации, что позволяет легко проверить безопасность за счет дополнительной загрузки процессора на хостах базы данных. Однако, если выполняется много изменений, это может оказаться сложнее, потому что приходится либо периодически вычислять контрольные суммы для уже проверенных данных, либо просто делать случайные выборки.

---

[1] Вероятно, имеются в виду приложения, выполняющие собственно репликацию. — *Примеч. науч. ред.*

Структуры памяти

Хранилища данных постоянно имеют дело с разнообразными структурами памяти. Одной из самых распространенных структур для БД является кэш данных. Он может называться по-разному, но его назначение состоит в том, чтобы хранить часто используемые данные в памяти, а не на диске. Встречаются и другие кэши, в том числе кэши для анализируемого SQL, кэши соединений, кэши результатов запросов и др.

При мониторинге этих структур применяются следующие типичные показатели.

- *Коэффициент использования*. Общий объем выделенного пространства, которое используется на протяжении заданного времени.
- *Коэффициент оттока*. Частота, с которой кэшируемые объекты удаляются, чтобы освободить место для других объектов или потому что исходные данные были объявлены недействительными.
- *Частота попаданий*. Частота, с которой удается воспользоваться кэшированными данными вместо некэшированных. Это может помочь при оптимизации производительности.
- *Конкурентность*. Часто эти структуры имеют собственные средства обеспечения правильной последовательности выполнения, такие как мьютексы, которые могут стать узкими местами. Понимание того, какова загруженность этих компонентов, также может помочь при оптимизации.

Некоторые системы, такие как Cassandra, используют для управления памятью виртуальные машины Java (JVM), что открывает совершенно новые области для мониторинга. Кроме того, в таких средах решающее значение имеют сборка мусора и применение различных куч (heap) для хранения объектов.

Коллизии параллельного доступа и блокировки

В реляционных базах активно используются блокировки для поддержки одновременного доступа из нескольких сеансов. Блокировки позволяют выполнять операции чтения и модификации, гарантируя при этом, что другие процессы не смогут в это время ничего изменить в данных. Это исключительно полезно, однако может приводить к задержкам, поскольку образуется очередь процессов, ожидающих снятия блокировки. В некоторых случаях возможны превышения тайм-аутов процессов из-за тупиков (deadlock, взаимная блокировка), для которых ситуации просто не могут быть разрешены иначе, чем откатом к предыдущему состоянию. Подробнее о реализации блокировок читайте в главе 11.

Мониторинг блокировок включает в себя мониторинг времени, проведенного в ожидании снятия блокировок хранилища данных. Это можно считать показателем перегрузки, и рост очередей может указывать на проблемы с приложениями

и с блокировками или на какие-то первичные неполадки (первопричины), влияющие на задержку, когда сеансы с блокировками требуют больше времени для завершения. Важен также мониторинг откатов изменений и тупиков, поскольку это еще один показатель того, что приложения не отменяют блокировки корректно, что приводит к превышению тайм-аутов ожидания и откату. Откаты изменений могут быть частью обычных транзакций с нормальным поведением, но часто бывают основным показателем того, что транзакциям мешают какие-то скрытые на более низком уровне операции.

Как обсуждалось ранее в разделе, посвященном структурам памяти, в базе данных есть множество точек, которые функционируют как примитивы синхронизации, предназначенные для безопасного управления конкурентным (параллельным) доступом. Обычно это мьютексы или семафоры. Мьютекс (Mutex, Mutually Exclusive Lock — взаимно исключающая блокировка, взаимное исключение) — механизм блокировки, используемый для синхронизации доступа к ресурсу, такому как запись в кэше. Владеть мьютексом (захватить мьютекс) может только одна задача. Это означает, что если существует объект, связанный с мьютексами, то только его владелец может снять блокировку (освободить мьютекс). Это и обеспечивает защиту данных от повреждений.

Семафоры ограничивают количество пользователей, одновременно получающих доступ к общему ресурсу, некоторым максимально допустимым значением. Потоки могут запрашивать доступ к ресурсу (уменьшая семафор) и сигнализировать, что они закончили использовать ресурс (увеличивая семафор). В табл. 4.1 перечислены примеры применения мьютексов и семафоров для мониторинга системы управления базой данных InnoDB в MySQL.

Таблица 4.1. Показатели активности семафоров InnoDB

Название	Описание
Mutex Os Waits (Delta)	Количество семафоров/мьютексов InnoDB, ожидаемых операционной системой
Mutex Rounds (Delta)	Количество циклов семафоров/мьютексов InnoDB во внутреннем массиве синхронизации
Mutex Spin Waits (Delta)	Количество потоков, ожидающих семафоры/мьютексы InnoDB во внутреннем массиве синхронизации
Os Reservation Count (Delta)	Счетчик операций ожидания семафора/мьютекса InnoDB, добавленных во внутренний массив синхронизации
Os Signal Count (Delta)	Счетчик случаев извещения потоков InnoDB о событиях семафоров/мьютексов во внутреннем массиве синхронизации
Rw Excl Os Waits (Delta)	Счетчик операций ожидания семафоров исключительного доступа (запись) со стороны InnoDB

Продолжение ⇩

Таблица 4.1 *(продолжение)*

Название	Описание
Rw Excl Rounds (Delta)	Счетчик циклов ожидания исключительного доступа (запись) для семафоров в массиве синхронизации InnoDB
Rw Excl Spins (Delta)	Счетчик операций ожидания исключительного доступа (запись) для семафоров в массиве синхронизации InnoDB
Rw Shared Os Waits (Delta)	Счетчик операций ожидания семафоров совместного доступа (чтение) со стороны InnoDB
RW Shared Rounds (Delta)	Счетчик циклов ожидания совместного доступа (чтение) для семафоров в массиве синхронизации InnoDB
RW Shared Spins (Delta)	Счетчик операций ожидания совместного доступа (чтение) для семафоров в массиве синхронизации InnoDB
Spins Per Wait Mutex (Delta)	Средняя длительность ожидания в циклах для семафоров/мьютексов InnoDB во внутреннем массиве синхронизации
Spins Per Wait RW Excl (Delta)	Средняя длительность ожидания исключительного доступа (запись) в циклах для семафоров/мьютексов InnoDB к периодам ожидания во внутреннем массиве синхронизации
Spins Per Wait RW Shared (Delta)	Средняя длительность ожидания совместного доступа (чтение) в циклах для семафоров/мьютексов InnoDB во внутреннем массиве синхронизации

Увеличение этих значений может указывать на то, что в некоторых местах кодовой базы достигается предельное допустимое количество одновременных обращений к хранилищу. Решением этой проблемы может быть настройка параметров и/или масштабирование, что должно обеспечить обоснованный уровень параллелизма при доступе к хранилищу данных для удовлетворения требований трафика.

Коллизии доступа и блокировки действительно способны убить даже самые высокопроизводительные запросы, когда достигается критическая точка в масштабировании. Выполняя мониторинг этих показателей в ходе нагрузочного тестирования и в производственных условиях, можно выяснить ограничения программного обеспечения базы данных и определить пути оптимизации приложений для увеличения количества одновременно работающих пользователей.

Объекты базы данных

Очень важно понимать, что представляет собой ваша база данных и как в ней хранятся данные. В простейшем случае это понимание того, сколько памяти занимают каждый объект базы данных и связанные с ним ключи и индексы. Как и в случае с хранилищем на основе файловой системы, понимание скорости роста и времени, за которое будет достигнута верхняя граница, здесь столь же важно, если не важнее, чем знание текущего варианта хранилища.

Полезно не только понимать, как хранятся данные и как увеличивается хранилище, но и контролировать характер распределения критически важных данных. Например, знать верхние и нижние границы, средние значения и область значений для элементов данных полезно для понимания производительности операций индексации и поиска. Это особенно важно для целочисленных типов и символьно-ориентированных данных с малым разнообразием значений. Если эти данные находятся на контроле у программистов, это позволяет вам и им оптимально выбирать типы данных и индексации.

Если набор данных сегментирован с использованием диапазонов значений ключа или списков, то понимание того, как происходит распределение между сегментами, поможет обеспечить максимальную производительность для каждого узла. Эти методики сегментирования позволяют учесть «горячие точки», где распределение нарушает порядок, заданный хеш-функциями или «модулярным» методом[1]. Выявление этого позволит вам и вашей команде сформулировать рекомендации о необходимости изменения балансировки или пересмотра моделей сегментирования.

Запросы к базе данных

В зависимости от системы управления базой данных, с которой вы работаете, доступ к данным и манипуляции с ними на деле могут как сопровождаться многочисленными средствами контроля, так и обходиться вовсе без них. Попытка палить из пушки по воробьям, подробно протоколируя все запросы в высоконагруженной системе, может привести к критическим проблемам с задержками и доступностью для системы и ее пользователей. Однако нет более ценных данных, чем эти. Некоторые решения, такие как Vivid Cortex и Circonus, ориентированы на TCP и протоколы проводной передачи данных для получения необходимой информации, что радикально снижает негативный эффект от протоколирования запросов. Другие методы предусматривают выполнение выборок на менее загруженной реплике и включение журналирования только в фиксированные периоды времени или только для медленно выполняемых операций.

Независимо от этого вы, очевидно, захотите хранить как можно больше информации о производительности и коэффициенте использования базы данных. Это и потребление ресурсов процессора и ввода-вывода, и количество прочитанных и записанных строк, и детализированное время выполнения и время ожидания, а также счетчик выполненных операций. Решающее значение для оптимизации имеет понимание способов оптимизации, применяемых индексов и статистики по объединению, сортировке и агрегированию.

---

[1] Вероятно, имеется в виду использование целочисленного деления и остатков. — *Примеч. науч. ред.*

Проверки и события базы данных

Журналы базы данных и клиентских приложений — богатый источник информации, особенно о проверках и ошибках. Эти журналы могут дать важные сведения, которые нельзя получить иными способами, например, следующие:

- попытки соединения и неудачные соединения;
- предупреждения и ошибки из-за повреждений данных;
- перезапуски базы данных;
- изменения конфигурации;
- возникновение тупиков;
- дампы ядра и трассировки стека.

Часть этих данных вы можете агрегировать и направить в свои системы сбора показателей. Остальные стоит рассматривать как события, которые нужно отслеживать и протоколировать, чтобы использовать для установления соответствий.

Резюме

Ну что ж, наверное, после всего этого нам нужен перерыв! Прочитав главу, вы хорошо поняли важность оперативного контроля, узнали, как запустить программу OpViz, как создавать и развивать архитектуру OpViz. У вас никогда не будет достаточно много информации о системах, которые вы строите и используете. Но вы сможете быстро найти средства, созданные для контроля тех сервисов, за которые вы будете отвечать в ходе эксплуатации! Они заслуживают не меньшего внимания, чем любой другой компонент инфраструктуры.

5 Инжиниринг инфраструктуры

Начнем процесс актуализации кластеров БД, результатами которого будут пользоваться приложения и аналитики. Мы уже обсудили большую часть подготовительной работы: ожидания уровня качества обслуживания, анализ рисков и, конечно же, оперативный контроль. В следующих двух главах мы рассмотрим методы и паттерны проектирования и построения сред.

В этой главе мы изучим различные хосты, на которых может работать хранилище данных, включая варианты *внесерверной обработки данных* и *базы данных как сервиса*, а также обсудим разные варианты хранения, доступные для этих хранилищ данных.

Хосты

Как говорилось ранее, хранилища данных существуют не в вакууме. Они всегда запускаются как процессы на каком-либо хосте. Традиционно узлы базы данных были физическими серверами. За последнее десятилетие появились новые возможности и инструменты, включая виртуальные хосты, контейнеры и даже абстрактные сервисы. Рассмотрим каждый из них, обсудим достоинства и недостатки их использования для БД и некоторые детали реализации.

Физические серверы

В данном контексте физический сервер — хост с операционной системой, предназначенный исключительно для запуска сервисов непосредственно из этой операционной системы. В незрелых средах на физическом сервере можно запускать много сервисов, если трафик низкий, а ресурсов много. Один из первых шагов, которые предпримет инженер по обеспечению надежности БД, — выделение собственных серверов для хранилищ данных. Для хранилищ данных характерна значительная загрузка процессоров, оперативной памяти и подсистемы ввода-вывода. Некоторым приложениям требуется больше ресурсов процессора или устройств ввода-вывода, и вы не хотите, чтобы они конкурировали за эти ресурсы с другими приложениями.

Настройка рабочих нагрузок также довольно специфична и, следовательно, требует правильной изоляции.

Если база данных запускается на выделенном физическом хосте, она будет использовать множество компонентов (мы вскоре кратко их обсудим) и взаимодействовать с ними. В рамках этого обсуждения предположим, что на хосте установлена операционная система Linux или Unix. Несмотря на то что многое из сказанного применимо и к Windows, различия довольно существенны, поэтому в примерах мы не будем использовать эту ОС.

Мы постарались привести здесь множество рекомендуемых методов, чтобы продемонстрировать глубину знаний об ОС и оборудовании, которые DBR-инженер может использовать для решения проблем, способных значительно повысить доступность и производительность БД. По мере повышения уровня абстракции и перехода к виртуальным хостам и контейнерам будем добавлять соответствующую информацию там, где это уместно.

Работа на уровне системы и ядра

Вы, как DBRE, должны напрямую сотрудничать с инженерами по обеспечению надежности информационных систем (Software Reliability Engineers, SRE), чтобы определить конфигурации ядра для хостов базы данных. Ваши договоренности должны стать теми эталонами («золотыми стандартами»), которые будут автоматически развертываться одновременно с двоичными файлами базы данных и другими конфигурациями. Для большинства СУБД имеются конкретные требования и рекомендации поставщика, которые следует изучить и применять, где это возможно. Интересно, что существуют разные подходы к этому в зависимости от типа используемой БД. Итак, рассмотрим и обсудим несколько категорий высокого уровня.

Ограничения пользовательских ресурсов

Существует ряд ресурсов, которые базы данных задействуют гораздо интенсивнее, чем обычные серверы. К ним относятся файловые дескрипторы, семафоры и пользовательские процессы.

Планировщик ввода/вывода

Планировщик ввода/вывода (I/O scheduler) — это инструмент, применяемый операционными системами для определения последовательности, в которой будут выполняться низкоуровневые операции ввода-вывода, то есть обмен с блочными устройствами, входящими в хранилище данных. По умолчанию планировщики предполагают, что они работают с вращающимися дисками с большой задержкой поиска. Поэтому обычно используется *алгоритм лифта (elevator algorithm)*, со-

гласно которому запросы упорядочиваются по местоположению, а время поиска минимизируется. В Linux можно увидеть следующие доступные опции:

```
wtf@host:~$ cat /sys/block/sda/queue/scheduler
[noop] anticipatory deadline cfq
```

Если блочное устройство, в отношении которого осуществляется операция, представляет собой массив SSD с контроллером, выполняющим оптимизацию ввода-вывода, то правильным решением будет выбрать планировщик `noop`. Все запросы ввода-вывода обрабатываются одинаково, поскольку время поиска на твердотельных накопителях (SSD) довольно стабильно. Планировщик по сроку завершения (дедлайна, Deadline Scheduler) будет выполнять оптимизацию, стремясь минимизировать задержку ввода-вывода, задавая крайние сроки, чтобы предотвратить зависание процесса, и устанавливая приоритеты операций чтения выше, чем операций записи. Подобные планировщики продемонстрировали особенно высокую производительность в многопоточных средах с высокой степенью параллелизма, таких как загрузка базы данных.

Распределение и фрагментация памяти

Никто не спорит с тем, что базы данных относятся к самым ресурсоемким приложениям, которые могут выполняться на сервере. Понимание способов распределения памяти и управления ею имеет решающее значение для ее максимально эффективного использования. Для компиляции двоичных файлов базы данных применяются различные библиотеки распределения памяти. Вот некоторые примеры таких библиотек.

- В MySQL InnoDB, начиная с версии 5.5, задействуется пользовательская библиотека, которая упаковывает `glibc mal loc`. Говорят, что сервису GitHub удалось сократить задержки на 30 %, переключившись на `tcmalloc`, а Facebook использует `jemalloc`.
- В PostgreSQL также есть собственная пользовательская библиотека размещения на основе `malloc`. В отличие от большинства других хранилищ данных PostgreSQL выделяет память очень большими порциями, называемыми *контекстами памяти* (memory contexts).
- Apache Cassandra начиная с версии 2.1 при размещении вне кучи вместо родного распределения памяти использует `jemalloc`.
- В реализации MongoDB версии 3.2 по умолчанию задействуется `malloc`, но эту СУБД можно настроить на `tcmalloc` или `jemalloc`.
- В Redis, начиная с версии 2.4, применяется `jemalloc`.

Доказано, что и `jemalloc`, и `tcmalloc` существенно повышают параллелизм для большинства рабочих нагрузок базы данных, значительно улучшая производительность по сравнению с исходным `glibc malloc`, а также снижая фрагментацию.

По умолчанию память выделяется в виде страниц размером 4 Кбайт. Таким образом, в 1 Гбайт памяти хранится 262 144 страницы. Процессоры используют *таблицу страниц*, содержащую список этих страниц, в котором на каждую страницу ссылается *запись таблицы страниц. Буфер быстрого преобразования адреса* (Translation Lookaside Buffer, TLB) — это кэш памяти, в котором для более быстрого поиска хранятся последние преобразования виртуальной памяти в физические адреса. *Неудача* (miss) TLB — это ситуация, когда в TLB отсутствуют данные преобразования виртуальной страницы в физический адрес. При неудаче TLB подключение происходит медленнее, чем обнаруживается соответствие, и может потребоваться просмотр страницы с несколькими загрузками. При больших объемах памяти и, следовательно, больших страницах неудачи TLB могут привести к перезагрузке кэша.

Transparent Huge Pages (THP) — система управления памятью Linux, позволяющая сократить издержки при поиске в TLB на машинах с большим объемом памяти, используя большие страницы памяти и тем самым уменьшая количество требуемых записей. THP — это блоки памяти размером 2 Мбайт и 1 Гбайт. Таблицы, применяемые для страниц объемом 2 Мбайт, годятся для памяти объемом несколько гигабайт, тогда как страницы размером 1 Гбайт лучше подходят для терабайтной памяти. Однако дефрагментация таких больших страниц может привести к значительной перегрузке процессора, что наблюдается, в частности, в рабочих нагрузках Hadoop, Cassandra, Oracle и MySQL. Чтобы это смягчить, может потребоваться отключить дефрагментацию и потерять из-за этого до 10 % памяти.

Linux не особенно оптимизирована для нагрузок базы данных, требующей низких задержек и высокой степени распараллеливания. Невозможно предсказать, когда ядро перейдет в режим восстановления, и одна из лучших рекомендаций, которые мы можем дать, — просто никогда не использовать физическую память полностью, а зарезервировать ее часть, чтобы избежать зависаний и значительных задержек. Чтобы зарезервировать память, вы можете не выделять ее при настройке ОС.

Подкачка

В системах Linux/UNIX *подкачка* — это процесс хранения и извлечения данных, которые больше не помещаются в память и должны записываться на диск, чтобы можно было уменьшить нагрузку на ресурсы памяти. Эта операция очень медленная — на несколько порядков медленнее по сравнению с доступом к памяти, следовательно, ее нужно рассматривать как крайний метод.

Принято считать, что базам данных следует избегать подкачки, поскольку при этом задержка сразу возрастает, превышая допустимый уровень. Тем не менее, если отключить подкачку, компонент операционной системы Out of Memory Killer (OOM Killer) завершит процесс базы данных.

Есть две точки зрения на это явление. Первый, более традиционный подход заключается в том, что лучше пускай база данных работает медленно, чем не работает

вообще. Второй подход, который лучше согласуется с философией DBRE, заключается в том, что задержки — это так же плохо, как и падение производительности; таким образом, подкачка с диска вообще не должна допускаться.

Конфигурации базы данных обычно имеют два верхних предела использования памяти: реалистичный и теоретический. Фиксированные структуры памяти, такие как буферные пулы и кэши, потребляют фиксированное количество памяти, которое легко спрогнозировать. Однако на уровне соединения все может стать более запутанным. Существует теоретический предел, основанный на максимальном количестве соединений и максимальном размере всех структур памяти каждого соединения, таких как буферы сортировки и стеки потоков. Применяя пулы соединений и опираясь на некоторые разумные предположения, вы должны быть в состоянии предсказать безопасный порог памяти, который позволил бы избежать подкачки.

В этом процессе учитываются такие аномалии, как неправильная конфигурация, неконтролируемые процессы и другие нежелательные события, способные превысить пределы памяти, что приводит к отключению сервера в случае события OOM. Но это на самом деле хорошо! Разработав эффективные стратегии обеспечения контроля, производительности и отработки отказов, вы фактически исключите потенциальные отклонения от целевого уровня качества обслуживания (SLO) в отношении задержек.

Отключение подкачки

Это следует делать только в том случае, если у вас есть надежные процессы восстановления после отказа. Иначе такая операция обязательно повлияет на доступность приложений.

Если вы разрешите подкачку в своей среде, то сможете уменьшить вероятность того, что операционная система будет выполнять подкачку памяти базы данных для файлового кэша, что, как правило, бесполезно. Вы также можете настроить оценки OOM для процессов базы данных, чтобы уменьшить вероятность того, что профилировщик памяти ядра убьет процесс базы данных, стараясь освободить память для других задач.

Неравномерный доступ к памяти

В ранних реализациях поддержки нескольких процессоров применялась архитектура Symmetric Multiprocessing (SMP), обеспечивавшая *равный доступ к памяти* для каждого процессора через общую шину, соединяющую процессоры и банки памяти. В современных многопроцессорных системах используется *неравномерный доступ к памяти* (Non-Uniform Memory Access, NUMA), который выделяет локальный банк памяти для каждого процессора. Доступ к памяти, принадлежащей другим процессорам, по-прежнему реализуется через общую шину. Таким образом,

одни обращения к памяти (локальные) имеют значительно меньшую задержку, чем другие (удаленные).

В Linux процессор и его ядра составляют *узел*. Операционная система присоединяет банки памяти к их локальным узлам и вычисляет затраты на обращения к памяти между узлами на основе расстояния между ними. Процессу и его потокам будет выделен *предпочтительный* узел для использования памяти. Это можно временно изменить с помощью планировщиков, но привязка всегда будет приводить к предпочтительному узлу. Кроме того, после окончания процедуры выделения памяти последняя не будет передаваться другому узлу.

Это означает, что в среде, где имеются большие структуры памяти, такие как пулы буферов базы данных, память будет тесно привязываться к предпочтительному узлу. Такой дисбаланс приведет к тому, что при отсутствии доступной памяти предпочтительный узел быстро заполнится. Но даже если использовать меньше памяти, чем физически доступно на сервере, все равно не удастся избежать подкачки.

РЕШЕНИЕ ПРОБЛЕМЫ NUMA И MYSQL В «ТВИТТЕРЕ»

Джереми Коул (Jeremy Cole) подготовил два впечатляющих поста об этой проблеме и о том, как он ее решил в «Твиттере». Первоначально использовался подход, предусматривающий принудительное выделение памяти с чередованием адресов посредством `numactl --interleave = all`.

Благодаря чередованию адресов распределение памяти может эффективно распространяться на все узлы. Но это не было эффективным на 100 %, поскольку буферный кэш операционной системы может довольно быстро заполниться, если процесс MySQL, некоторое время выполнявшийся в производственной среде, будет перезапущен. Добавив к решению еще два пункта, удалось построить повторяемый и надежный процесс.

- Очистка буферных кэшей Linux непосредственно перед запуском `mysqld` с помощью команды `sysctl -q -w vm.drop_caches = 3`. Это позволило обеспечить справедливое распределение даже при перезапуске демона, когда значительные объемы данных находятся в буферном кэше операционной системы.
- Принудительное выделение операционной системой пула буферов InnoDB сразу после запуска с помощью `MAP_POPULATE`, там, где это поддерживается (в Linux 2.6.23+), в противном случае — откат к `memset`. Это приводит к немедленному принятию решения о распределении памяти между узлами NUMA, в то время как буферный кэш остается свободным и не задействованным в описанной ранее очистке.

Это прекрасный пример того, насколько DBRE может быть полезен очень большой группе SWE и SRE. В данном случае решение проблемы чрезмерной подкачки потребовало вникнуть в управление памятью операционной системы. В сочетании с глубокими знаниями об управлении памятью в MySQL это позволило решить проблему, а в дальнейшем послужило основой для модификации самого MySQL.

На этом этапе для большинства запросов СУБД устанавливается расслоение для NUMA в ядре. Подобные вопросы широко обсуждаются в отношении PostgreSQL, Redis, Cassandra, MongoDB, MySQL и ElasticSearch.

Сеть

В этой книге мы исходим из предположения, что все хранилища данных являются распределенными. Сетевой трафик имеет решающее значение для производительности и доступности БД. Весь сетевой трафик можно разбить на следующие категории:

- межузловой обмен данными;
- трафик приложений;
- административный трафик;
- трафик резервного копирования и восстановления.

Межузловой обмен данными включает в себя репликацию данных, работу согласованных протоколов и протоколов мессенджеров, а также управление кластером. Эти данные позволяют кластеру знать о своем состоянии и поддерживать репликацию данных в заданных количествах. Трафик приложений — это трафик, поступающий с серверов приложений или прокси-серверов. Он поддерживает состояние приложений и позволяет им создавать, изменять и удалять данные.

Административный трафик характеризует обмен данными между системами управления, операторами и кластерами. Сюда входят запуск и остановка сервисов, развертывание двоичных файлов, а также внесение изменений в базу данных и конфигурацию. Если где-то в другом месте дела идут плохо, этот трафик — единственная надежда, он позволяет восстановить систему вручную или автоматически. Что такое трафик резервного копирования и восстановления, понятно из названия. Это трафик, создаваемый при архивировании и копировании данных, перемещении данных между системами или восстановлении данных из резервных копий.

Изоляция трафика — один из первых шагов к построению правильной сети для БД. Это можно сделать с помощью физических *сетевых адаптеров* (Network Interface Card, NIC) или путем совместного использования одного NIC. Современные серверные сетевые адаптеры обычно обеспечивают скорость 1 или 10 Гбит/с, и их можно объединить в пару, чтобы обеспечить избыточность и распределение нагрузки. Избыточность увеличит среднее время между отказами (MTBF), однако это пример устойчивости, а не отказоустойчивости.

Базы данных нуждаются в компактной и экономичной реализации транспортного уровня, способной эффективно справляться с требуемыми нагрузками. Частые

и быстрые соединения, короткие обратные вызовы и запросы, чувствительные к задержке, требуют особой настройки. Ее можно разбить на три части:

- обеспечение большого количества соединений за счет увеличения количества доступных портов TCP/IP;
- сокращение времени, необходимого для сброса и перезапуска сокетов, чтобы избежать большого количества соединений, ожидающих закрытия (`TIME_WAIT`, см. описание протокола TCP), — уже неработоспособных, но еще недоступных для повторного открытия;
- увеличение очереди запросов на установление TCP-соединения (backlog, см. описание протокола TCP), чтобы избежать отказов из-за ее переполнения.

Протокол TCP/IP станет вашим лучшим другом при устранении проблем с задержками и доступностью. Мы настоятельно рекомендуем вам изучить его. Для этого хорошо подходит том 1 книги Дугласа Э. Комера (Пирсона) (Douglas E. Comer (Pearson)) *Internetworking with TCP/IP*. В 2014 году она была переиздана.

Хранилище

Описание хранилища БД заслуживает отдельной книги. Вам придется изучить отдельные диски, сгруппированную конфигурацию дисков, контроллеры, обеспечивающие доступ к дискам, программное обеспечение для управления томами и установленные поверх всего этого файловые системы. С любой из этих тем вполне реально самостоятельно разобраться за несколько дней, поэтому мы сосредоточимся на общей картине.

На рис. 5.1 показаны способы распространения данных в хранилище. При чтении данных из файла вы переходите из пользовательского буфера в кэш страниц, затем к контроллеру диска и, наконец, к конкретной «пластине» диска для извлечения данных, а затем возвращаетесь назад, чтобы доставить данные пользователю.

Этот сложный каскад буферов, планировщиков, очередей и кэшей используется для смягчения того факта, что диски работают очень медленно по сравнению с памятью — *10 миллисекунд вместо 100 наносекунд*.

Для баз данных определены пять основных параметров (целей):

- емкость;
- пропускная способность (количество операций ввода-вывода в секунду — IO per second, IOPS);
- время отклика, или задержка;
- доступность;
- устойчивость.

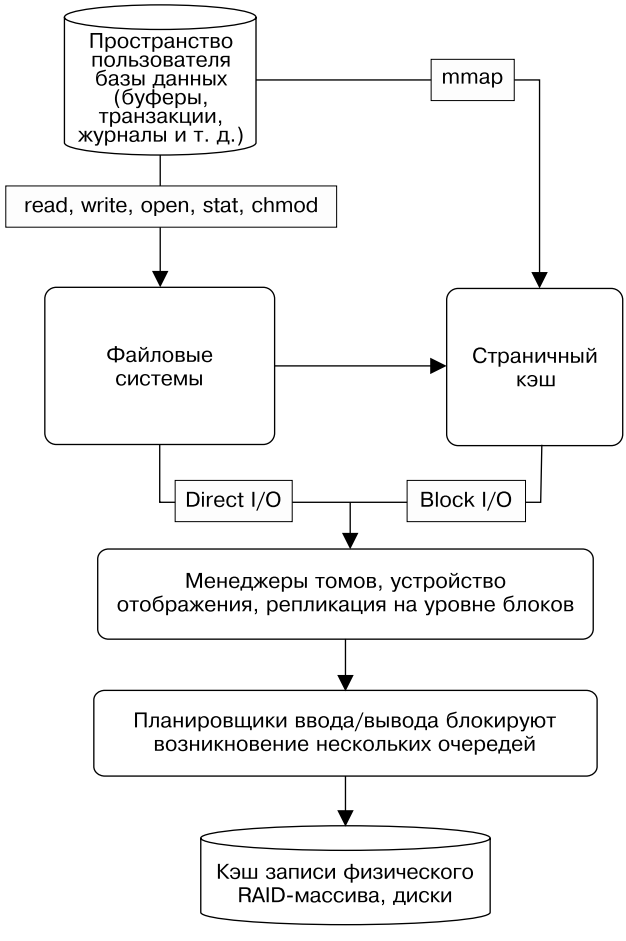


Рис. 5.1. Стек хранения Linux

Емкость хранилища

Емкость — это объем пространства, доступного для хранения данных и журналов в вашей базе данных. Хранилище может размещаться на больших дисках, на нескольких дисках с чередованием (RAID 0) или на нескольких дисках, функционирующих как отдельные точки монтирования, известные как JBOD (just a bunch of disks — «просто несколько жестких дисков»). Каждое из этих решений имеет свою модель отказов. Одиночные большие диски являются единственными точками отказа, если они не зеркальные (RAID 1). Для RAID 0 значение MTBF уменьшено в N раз, где N — количество дисков в массиве с чередованием. У JBOD сбои будут более частыми, но здесь, в отличие от RAID 0, окажутся доступными

N – 1 других дисков. Некоторые базы данных могут использовать их и сохранять работоспособность, пускай и ограниченную, пока не будет приобретен и установлен запасной диск.

Понимание того, какова общая потребность базы данных в хранилище, — лишь часть общей картины. Если вам требуется хранилище объемом 10 Тбайт, можете создать массив дисков с чередованием общим объемом 10 Тбайт или смонтировать JBOD из десяти дисков по 1 Тбайт и распределить файлы данных между ними. Однако теперь у вас есть база данных емкостью 10 Тбайт для резервного копирования за один раз, и в случае сбоя придется восстановить 10 Тбайт, что займет много времени. В течение этого времени будет снижена производительность и доступность всей системы. Одновременно необходимо учесть, способны ли ПО базы данных, операционная система и аппаратное обеспечение управлять параллельными рабочими нагрузками для чтения из этого монолитного хранилища данных и записи в него. Разбиение системы на более мелкие базы данных повысит отказоустойчивость, емкость и производительность приложений, систем резервного копирования/восстановления и средств копирования набора данных.

Пропускная способность хранилища

IOPS — это стандартная единица измерения, количество операций ввода-вывода в секунду на данном устройстве хранения. Такие операции включают в себя чтение и запись. При учете потребностей следует рассматривать IOPS не для средней, а для пиковой рабочей нагрузки базы данных. Для правильного планирования новой системы необходимо оценить количество операций ввода-вывода в секунду, требуемое для каждой транзакции, и пиковое количество транзакций. Эти значения, очевидно, будут сильно различаться в зависимости от приложения. Можно ожидать, что приложение, которое выполняет постоянные операции вставки и чтения одной строки, будет производить четыре или пять операций ввода-вывода в секунду на транзакцию. Сложные, состоящие из нескольких запросов транзакции запросто могут включать 20–30 операций ввода-вывода.

Рабочие нагрузки базы данных обычно бывают смешанными, предполагая операции как чтения, так и записи, и не последовательными, а случайными. Есть ряд исключений, таких как схемы записи только для добавления (например, SSTables Cassandra), которые выполняют последовательную запись. Для жестких дисков (HDD) значения IOPS для случайного доступа зависят главным образом от времени поиска (позиционирования) устройства хранения. Для твердотельных накопителей значения IOPS случайного доступа ограничены быстродействием внутреннего контроллера и интерфейса памяти. Это объясняет значительный прирост IOPS, наблюдаемый в SSD. Значения IOPS для последовательного доступа соответствуют максимальной пропускной способности диска. Зачастую значения IOPS последовательного доступа измеряются в мегабайтах в секунду и указывают на то, с какой скоростью может выполняться чтение больших массивов или последовательная запись.

Присматриваясь к твердотельным накопителям, не забудьте о шине. Рассмотрите возможность установки флеш-решения для шины PCIe, такого как FusionIO, с пропускной способностью 6 Гбит/с и задержкой несколько микросекунд. Однако на момент написания этой книги 10 Тбайт обошлись бы вам примерно в 45 000 долларов.

Традиционно значение IOPS было ограничивающим фактором для емкости хранилища. Это особенно верно для операций записи, которые нельзя оптимизировать с помощью кэширования, в отличие от операций чтения. Увеличение числа дисков и их объединение с помощью технологий RAID 0 (объединение с чередованием, striping) или JBOD приведет к росту не только емкости хранилища, но и IOPS. RAID 0 обеспечит однообразную задержку и устранит «горячие точки» (выбросы), которые иногда появляются в JBOD, но за счет уменьшения MTBF, в зависимости от количества дисков в наборе.

Время отклика хранилища

Время отклика, или задержка (latency), — это суммарная длительность операции ввода-вывода для клиента. Другими словами, это время, прошедшее между запросом на операцию ввода-вывода в хранилище и получением подтверждения о завершении чтения или записи. Как и для большинства ресурсов, здесь существуют очереди для ожидающих запросов, которые могут появляться в периоды загруженности. Очереди — это не так уж плохо — до определенной степени. Фактически многие контроллеры предназначены именно для оптимизации глубины очереди. Если рабочая нагрузка не создает достаточного количества запросов ввода-вывода для полного использования доступной производительности, то хранилище может не обеспечивать ожидаемую пропускную способность.

Приложения транзакционных БД чувствительны к увеличению задержки ввода-вывода, и для них оправдано использование твердотельных накопителей. Вы можете сохранять высокий уровень IOPS и одновременно малое время отклика, устанавливая для тома небольшую длину очереди и высокий лимит количества операций ввода-вывода. Постоянное применение к тому большего количества операций ввода-вывода в секунду, чем позволяет данный том, систематическое превышение обеспечиваемого томом лимита IOPS могут увеличить задержку ввода-вывода.

Приложения, требующие высокой пропускной способности, например масштабные запросы MapReduce, менее чувствительны к увеличению задержки ввода-вывода и хорошо подходят для жестких дисков. Вы можете обеспечивать высокую пропускную способность для жестких дисков, поддерживая большую длину очереди при выполнении последовательного ввода-вывода значительного объема данных.

Еще одним узким местом, увеличивающим время отклика, является страничный кэш Linux. Используя прямой ввод-вывод (`O_DIRECT`), можно обойти кэширование страниц и избежать влияния его задержки, составляющей несколько миллисекунд.

Доступность хранилища

Производительность и емкость являются критически важными показателями, но необходимо учитывать также надежность хранения. В 2007 году компания Google провела обширное исследование частоты отказов жестких дисков, которое называлось *Failure Trends in a Large Disk Drive Population* («Тенденции отказов большого количества дисковых накопителей») (https://static.googleusercontent.com/media/research.google.com/ru//archive/disk_failures.pdf) (рис. 5.2). Исходя из его результатов, можно ожидать, что почти 3 из 100 дисков выйдут из строя в течение первых трех месяцев работы. Из тех дисков, что выйдут из строя в течение первых шести месяцев, примерно 1 из 50 сбоев случится за период от шести месяцев до одного года эксплуатации. Это не так уж много, но если у вас есть шесть серверов БД с восемью дисками на каждом, то можно ожидать сбоя диска в течение указанного периода.

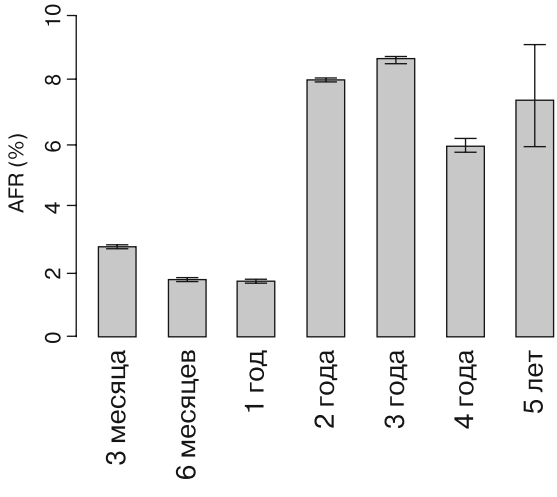


Рис. 5.2. Частота отказов жесткого диска, по данным Google

Именно поэтому инженеры начали выполнять зеркалирование дисков или создавать RAID-массивы, такие как RAID 1, и устранять недостатки массивов дисков с чередованием с помощью контроля четности. В этой книге мы лишь вкратце упомянем массивы с контролем по четности (например, RAID 5), потому что у них очень велики затраты на запись. Современные подсистемы хранения обычно организованы как JBOD или сгруппированы с чередованием (RAID 0), зеркалированием (RAID 1) или с зеркалированием и чередованием (RAID 10). С учетом этого легко предсказать, что RAID 1 и RAID 10 наиболее устойчивы к сбоям одного диска, в то время как хранилища RAID 0 наиболее склонны к полному отказу в обслуживании (и вероятность тем выше, чем из большего числа дисков с чередованием состоит хранилище). JBOD демонстрирует способность выдерживать сбои, продолжая обслуживать остальную часть хранилища.

Доступность — это не только среднее время между отказами (MTBF) для ваших томов. Это также то время, которое потребуется для восстановления после сбоя, или среднее время восстановления (MTTR). Выбор между RAID 10 и RAID 0 зависит от возможности простого развертывания сменных хостов базы данных во время сбоя. Иногда кажется заманчивым остановиться на RAID 0 по соображениям производительности. Для RAID 1 или 10 требуется вдвое больше IOPS для записи, а при использовании высокопроизводительных дисков дублирование оборудования может оказаться очень дорогим. RAID 0 производителен и предсказуем, но несколько нестабилен. В конце концов, можно ожидать, что в наборе жестких дисков с пятью накопителями за первый год случится около 10 % отказов. При четырех хостах можно ожидать сбоя каждые год-два.

Если есть зеркало, можно заменить диск без переорганизации базы данных. При работе с базой данных объемом 2 Тбайт на довольно быстро вращающихся дисках время резервного копирования базы измеряется в часах, и вам, вероятно, потребуется больше времени на репликацию БД, чтобы синхронизировать изменения из последней резервной копии. Придется подумать, хватит ли имеющейся емкости для поддержки пикового трафика на тот случай, если во время восстановления откажет один узел или даже два. В сегментированной среде, где каждый набор данных обслуживает лишь часть пользователей, это может быть вполне приемлемым, поэтому можно задействовать менее стабильные тома данных для экономии затрат и/или задержки записи.

Надежность хранения

И наконец, есть еще надежность (постоянство) хранения. Когда база данных хочет гарантированно сохранить данные на физическом диске, она явно обращается к операционной системе с вызовом `fsync()`, вместо того чтобы полагаться на автоматическую отложенную выгрузку страничного кэша. Примером этого может служить ситуация, когда создается журнал обновлений или журнал с упреждающей записью и его действительно нужно записать на диск, чтобы обеспечить возможность восстановления БД. Для обеспечения высокой производительности операций записи многие диски и контроллеры используют кэш в памяти. Функция `fsync` указывает подсистеме хранения, что такие кэши должны быть сброшены (выгружены) на диск. Если кэш записи питается от аккумулятора и способен пережить отключение питания, будет гораздо эффективнее не очищать этот кэш. Важно убедиться, что ваша структура действительно записывает данные в стабильное хранилище, будь то энергонезависимое устройство записи (NVRAM), кэш-память операций записи или сами дисковые накопители. Это можно сделать с помощью такой утилиты, как `diskchecker.pl` (https://gist.github.com/bradfitz/3172656) от Брэда Фитцпатрика (Brad Fitzpatrick).

Выполнение операций файловой системы также может привести к повреждениям и несогласованности во время сбоев, например при аварийных отказах. Однако использование файловых систем с журналированием, например XFS и EXT4, значительно уменьшает вероятность наступления таких событий.

Сети хранения данных

Данные можно хранить не только на локально подключенном накопителе, но и в сети хранения данных (Storage Area Network, SAN), подключаясь к ней через внешний интерфейс, обычно Fibre Channel. Использование сетей SAN значительно дороже, чем локальное хранение, зато централизация хранилища позволяет сократить расходы на управление и обеспечивает значительную гибкость.

Используя современные сети хранения, вы получаете гораздо больше буферов для кэширования данных. Кроме того, они предоставляют множество функций, которые могут быть полезны для больших наборов данных. Так, невероятно полезна возможность делать снимки для резервных копий и копий данных. На практике создание моментальных снимков и перемещение данных относятся к самым удобным функциям современной инфраструктуры, притом что твердотельные накопители обеспечивают лучшие показатели ввода/вывода, чем традиционные SAN.

Преимущества физических серверов

Использование физических серверов — самый простой вариант размещения БД. Нет никаких абстракций, которые бы скрывали детали реализации и особенности выполнения или увеличивали сложность. В большинстве случаев вы можете контролировать ОС так плотно, как только возможно. Благодаря этому функционирование системы оказывается максимально прозрачным.

Недостатки физических серверов

Тем не менее при использовании физических серверов возможны некоторые проблемы. Прежде всего, есть вероятность впустую потратить ресурсы, выделенные для конкретных серверов. Кроме того, развертывание таких систем может занять довольно много времени и иногда бывает трудно гарантировать идентичность всех серверов с точки зрения аппаратного и программного обеспечения. С учетом этого обсудим виртуализацию.

Виртуализация

При виртуализации применяется программное обеспечение, которое создает различные требуемые ресурсы, разделяющие и совместно использующие имеющуюся физическую инфраструктуру. Это программное обеспечение позволяет работать на одном сервере нескольким операционным системам, в которых выполняется по несколько приложений. Например, с помощью виртуальных машин (Virtual Machine, VM) можно поочередно запускать на одном сервере четыре экземпляра Linux, каждый со своими выделенными ресурсами процессора, памятью, сетевыми подключениями и хранилищем.

Виртуализация позволяет объединять ресурсы инфраструктуры, включая процессорные ресурсы, хранилище и сетевые соединения, и создавать пулы, которые можно распределять между виртуальными серверами. Это часто называют облачными вычислениями. Именно с ними вы имеете дело, если работаете в общедоступной облачной инфраструктуре, такой как Amazon Web Services (AWS). Но облачные вычисления можно осуществлять и в ваших собственных центрах обработки данных.

По сути, в любом решении, общедоступном, частном или смешанном, с помощью кода можно определить, как будут выглядеть ресурсы сервера и связанные с ними операционные системы. Это обеспечивает согласованное развертывание систем управления базами данных, что означает: DBR-инженеры смогут создавать собственные кластеры БД, настроенные в соответствии со стандартами, установленными DBRE. В эти стандарты входят:

- ОС;
- версия программного обеспечения базы данных;
- конфигурации ОС и базы данных;
- сведения о безопасности и полномочиях;
- пакеты и библиотеки программного обеспечения;
- сценарии управления.

Все это замечательно, но если добавить уровень абстракции поверх физических ресурсов, это создаст ряд дополнительных сложностей в управлении. Рассмотрим некоторые из них.

Гипервизор

Гипервизор, или монитор виртуальной машины (Virtual Machine Monitor, VMM), может представлять собой программное обеспечение — обычное или встроенное — или оборудование. Гипервизор создает и запускает виртуальные машины. Компьютер, на котором гипервизор запускает одну виртуальную машину или несколько, называется *хост-компьютером*, а каждая виртуальная машина — *гостевым компьютером*. Гипервизор представляет гостевые операционные системы с виртуальной операционной платформой и управляет работой гостевых ОС.

Параллелизм

Для баз данных, работающих под гипервизором, характерны более жесткие ограничения распараллеливания, чем для тех же программ на чистом «железе». При проектировании таких виртуализированных сред основное внимание следует уделять горизонтальному масштабированию, при котором распараллеливание внутри узлов сводится к минимуму.

Хранилище

Надежность и производительность — это не то, чего следует ожидать от виртуализированного хранилища. Между страничным кэшем виртуальной машины и физическим контроллером находятся виртуальный контроллер, гипервизор и страничный кэш хоста. Это означает увеличенную задержку операций ввода/вывода. У гипервизоров обычно не в почете вызовы `fsync` при записи данных — для повышения производительности. Это означает, что запись данных на диск в случае сбоя гарантировать невозможно.

Кроме того, даже если вы сможете легко запустить виртуальную машину за 10 минут или быстрее, это еще не означает создание данных, необходимых для работы существующей БД. Например, при развертывании новой реплики нужно будет откуда-то загрузить для нее данные.

Есть две основные категории хранилищ в виртуализированных средах: локальные и постоянные блочные. Локальное хранилище эфемерно. Его данные существуют, пока работает его виртуальная машина. Постоянное блочное хранилище можно подключить к любой виртуальной машине и использовать там. Если виртуальная машина отключится, то другая виртуальная машина сможет подключиться к этому хранилищу. Внешнее постоянное хранилище идеально подходит для БД. Такое блочное хранилище часто позволяет делать моментальные снимки, чтобы можно было легко перемещать данные.

Блочное хранилище гораздо сильнее зависит от сети, чем традиционные физические диски, и переполнение может быстро привести к снижению производительности.

Примеры использования

Приняв во внимание все эти предостережения, DBR-инженерам необходимо тщательно изучить особенности виртуальных и облачных ресурсов, если планируется использовать их для инфраструктуры БД. При проектировании этих инфраструктур следует учитывать все упомянутые факторы. Кратко перечислим их снова.

- Ненадежность хранения означает, что потеря данных неизбежна.
- Нестабильность копий означает, что автоматизация и восстановление после отказов должны быть очень надежными.
- Горизонтальное масштабирование требует автоматизации для управления значительным количеством серверов.
- Приложения должны справляться с периодическими задержками.

Даже с учетом всего этого виртуальные и облачные инфраструктуры могут иметь огромное значение для БД. Возможность создавать платформы самообслуживания, которые пользователи могли бы сами строить, чтобы потом работать с ними, серьезно увеличивает потенциал ресурсов DBRE. Это позволяет передавать знания и внедрять рекомендованные методики, даже имея в штате лишь несколько DBR-инженеров.

Быстрое развертывание также позволяет выполнять всестороннее тестирование приложений и прототипирование. Команды разработчиков становятся намного более эффективными, не создавая узкие места для DBRE при развертывании и конфигурировании. Это также означает, что разработчики с меньшей вероятностью окажутся предоставленными сами себе при развертывании уровней хранения данных новых приложений.

Контейнеры

Контейнеры располагаются поверх физического сервера и его операционной системы. Все контейнеры совместно используют ядро ОС, двоичные файлы и библиотеки. Эти разделяемые компоненты доступны только для чтения, а запись осуществляется в смонтированные уникальные его экземпляры. Контейнеры намного легче, чем виртуальные машины. На практике их размер составляет всего несколько мегабайт. Если для запуска виртуальной машины может потребоваться около 10 минут, запуск контейнера занимает считаные секунды.

Однако для хранилищ данных преимущества быстрого запуска в Docker часто нивелируются необходимостью присоединять, загружать и синхронизировать данные. Кроме того, настройки уровня ядра, большое количество ввода/вывода и перегрузка сети часто усложняют модель совместно используемой ОС или хоста. При этом Docker — отличный инструмент для быстрого выполнения развертываний в тестовых средах и средах разработки, и DBR-инженеры обязательно найдут ему применение, включив в свой инструментарий.

База данных как сервис

Компании все чаще ищут сторонние решения для своих виртуализационных и облачных сервисов. Взяв за основу модель самообслуживания, к которой мы вернемся в главе 6, вы получаете сторонние управляемые платформы БД. Их предлагают все провайдеры общедоступных облачных сервисов, наиболее известным из которых является Amazon Relational Database Service (RDS). Он предлагает MySQL, PostgreSQL, Aurora, SQL Server и Oracle. В этих системах вам предоставляется

возможность выбрать полностью развернутые среды БД для размещения в инфраструктуре.

Базы данных как сервис (Database as a Service, DBaaS) получили значительное распространение благодаря тому, что автоматизация многих наиболее рутинных действий службы эксплуатации освобождает время для ценных инженерных ресурсов. Вот типичные функции:

- развертывание;
- использование мастера восстановления после отказа;
- добавление исправлений (патчей) и обновлений;
- резервное копирование и восстановление из копий;
- просмотр показателей;
- повышение производительности благодаря «особому соусу», такому как Aurora от Amazon.

Все это высвобождает время, однако инженерам-программистам может показаться, что специалисты по базам данных не нужны. Это большое заблуждение! Абстрагированные сервисы привносят свои проблемы, но, что важнее, позволяют сосредоточиться на том, в чем ваши специализированные знания могут принести наибольшую пользу.

Проблемы DBaaS

Одной из самых больших проблем является отсутствие контроля. Не имея доступа к ОС, сетевым устройствам и оборудованию, вы не сможете диагностировать многие существенные проблемы.

> **DBAAS И NETWORK TIME PROTOCOL (NTP)**
>
> Мы работали с известным сервисом DBaaS, и наш клиент решил одним из первых воспользоваться новой версией базы данных, предоставляемой поставщиком. Мы умоляли клиента не делать этого, но соблазн был слишком велик, и мы углубились в бета-тестирование. В этот раз наш специалист из команды эксплуатации лихо проскочил мимо синхронизации сетевого протокола времени (Network Time Protocol, NTP) между всеми хостами БД. После нескольких часов устранения необъяснимой несогласованности репликаций мы связались со службой поддержки, чтобы выяснить, что именно происходит. Это была очень горячая ночь!

Сегодня многие системы мониторинга собирают данные базы данных SQL на уровне TCP для управления масштабированным сбором данных, однако они должны ис-

пользовать журналы или внутренние снимки, например *схему производительности* MySQL для данных. Кроме того, в этом случае недоступны такие проверенные временем инструменты трассировки и мониторинга, как `top`, `dtrace` и `vmstat`.

Проблемы надежности здесь аналогичны наблюдаемым в других виртуализированных средах, а реализация важных функций, таких как репликация и резервное копирование, часто оказывается работой с черным ящиком — приходится полагаться на то, что поставщик все сделал правильно.

DBRE и DBaaS

В мире маркетинга платформы DBaaS часто продаются в виде средства, избавляющего от потребности в дорогих специалистах по базам данных, которых трудно найти и потом удержать. Платформа DBaaS позволяет быстрее внедрить надежную инфраструктуру БД, что помогает отсрочить необходимость найма такого специалиста или привлечения его со стороны. Однако это не значит, что в итоге его все же не придется нанять.

Во всяком случае, несмотря на то, что DBaaS дает возможность абстрагироваться от рутинного труда и простых проблем, вы рискуете, что возникнет труднорешаемая проблема, прежде чем в вашей компании появятся специалисты по базам данных. Кроме того, есть ключевые решения, которые необходимо принять на ранней стадии, что требует наличия человека, обладающего глубоким знанием выбранного механизма управления базами данных. В частности, нужно определить следующее:

- какой механизм базы данных будет использоваться;
- как моделируются данные;
- какова соответствующая структура доступа к данным;
- какие решения по безопасности базы данных будут приняты;
- как происходит управление данными и каковы планы роста/мощности.

Таким образом, хоть благодаря DBaaS у разработчиков программного обеспечения и будет больше возможностей, вам, как DBR-инженеру, придется работать усерднее, чем когда-либо, чтобы помочь сделать правильный выбор и убедиться, что программисты понимают, где ваш опыт может повлиять на успешное или неудачное развертывание DBaaS.

DBaaS может быть очень привлекательной для организации, особенно в первое время, когда каждая минута инженерного времени невероятно ценна. DBR-инженерам настоятельно рекомендуется рассмотреть пути миграции и аварийного восстановления в инфраструктуре. Все, что может делать ваша система DBaaS, вы и ваша команда эксплуатации сможете в свое время автоматизировать, сохраняя в любых ситуациях полный контроль над хранилищами данных и доступ к ним.

Резюме

В этой главе мы рассмотрели различные комбинации хостов, которые вам могут встретиться: физические, виртуальные, контейнеры и сервисы. Обсудили, как влияют на работу системы доступные ресурсы процессора, памяти, сети и хранилищ, а также к каким последствиям могут привести выделение недостаточного количества ресурсов и неправильная конфигурация.

В главе 6 мы поговорим о том, как управлять этими инфраструктурами БД с помощью соответствующих инструментов и какие процессы обеспечивают масштабирование, управление рисками и сбоями. Рассмотрим настройку конфигурации, оркестрацию, автоматизацию и обнаружение сервисов, а также управление ими.

6 Управление инфраструктурой

В главе 5 мы рассмотрели различные компоненты инфраструктуры и парадигмы, с помощью которых можно управлять хранилищами данных. Здесь обсудим, как управлять ими при необходимости масштабирования. Начнем с самого маленького компонента — поговорим о конфигурации и определении отдельного хоста. Затем мы увеличим масштаб до развертывания хостов и оркестрации между компонентами. После этого пойдем дальше, к динамическому определению текущего состояния инфраструктуры и публикации этих данных, также известным как обнаружение сервисов. Наконец, перейдем к стеку для среды разработки и рассмотрим создание сред разработки, аналогичных большим стекам сред промышленной эксплуатации.

Прошли те времена, когда можно было обойтись одной или двумя стойками, легко и относительно стабильно управляемыми вручную. Сегодня мы должны быть готовы поддерживать крупные сложные инфраструктуры силами всего нескольких специалистов. Автоматизация имеет решающее значение для обеспечения возможности многократно надежно развертывать хранилища данных. От этого зависят стабильность и доступность приложений, а также скорость развертывания новых функций. Наши цели должны состоять в том, чтобы исключить повторяющиеся и/или ручные процессы и создать легко воспроизводимые инфраструктуры с помощью стандартизированных процессов и автоматизации.

Какие возможности для этого существуют?

- Установка программного обеспечения, включая ОС, базу данных, а также связанные пакеты и утилиты.
- Настройка программного обеспечения и ОС, гарантирующих желаемое поведение и рабочие нагрузки.
- Загрузка данных в новые базы данных.
- Установка сопутствующих инструментов, таких как агенты мониторинга, утилиты резервного копирования и наборы инструментов оператора.
- Тестирование инфраструктуры, обеспечивающей соответствующие настройки и поведение.
- Статическое и динамическое тестирование на соответствие.

Если нужно свести все это к минимуму, то необходимо гарантировать возможность последовательно создавать и/или воспроизводить любой компонент инфраструктуры БД, а также знать текущее и предыдущее состояния любого из компонентов в процессе устранения неисправностей и тестирования.

Честно сказать, это довольно общий обзор. Если хотите копнуть глубже, то мы рекомендуем вам прочесть книгу Кифа Морриса (Kief Morris) *Infrastructure as Code* (O'Reilly). Цель главы — продемонстрировать различные компоненты управления инфраструктурой с помощью кода, в который вам придется внести свой вклад, и указать, как это может облегчить вам жизнь как инженеру по обеспечению надежности БД.

Контроль версий

Чтобы достичь перечисленных ранее целей, необходимо использовать контроль версий для всех применяемых в процессе компонентов. В число этих компонентов входят:

- исходный код и сценарии;
- библиотеки и пакеты, которые функционируют как зависимости;
- конфигурационные файлы и другая конфигурационная информация;
- версии образов ОС и двоичных файлов базы данных.

Система контроля версий (version control system, VCS) является основой любого рабочего процесса разработки программного обеспечения. Инженеры по базам данных и системам, вместе с инженерами-программистами работающие над созданием и развертыванием приложений и инфраструктур и управлением ими, действуют совместно в рамках VCS. Раньше инженеры систем и БД обычно не использовали VCS. Если же они все-таки делали это, то отдельно от VCS команды программистов, из-за чего невозможно было сопоставлять версии инфраструктуры с версиями кода.

Вот некоторые примеры популярных платформ VCS:

- GitHub;
- Bitbucket;
- Git;
- Microsoft Team Foundation Server;
- Subversion.

Система контроля версий должна быть источником истины в последней инстанции для всех элементов инфраструктуры. К ним относятся сценарии, файлы конфи-

гурации и файлы определений, которые будут использоваться для установления кластера базы данных. Любые изменения нужно вносить в VCS. Когда нужно что-то изменить, вы получаете копию из VCS, вносите правки и затем снова фиксируете их в VCS. После фиксации изменений можно выполнить проверку, тестирование и, наконец, развертывание. Стоит отметить, что перед сохранением в VCS пароли нужно замаскировать.

Определение конфигурации

Чтобы определить конфигурацию и структуру кластера базы данных, вы будете применять различные компоненты. В вашем приложении для управления конфигурацией будет использоваться *предметно-ориентированный язык* (domain-specific language, DSL), но можно также работать со сценариями на Windows PowerShell, Python, Shell Scripts, Perl и других языках. Далее приведены некоторые популярные приложения для управления конфигурацией:

- Chef;
- Puppet;
- Ansible;
- SaltStack;
- CFEngine.

Определяя конфигурацию, а не описывая ее в виде сценария, вы создаете легкочитаемые компоненты, которые сможете повторно использовать в своей инфраструктуре. Это обеспечивает согласованность и часто уменьшает объем работы, необходимой для добавления нового компонента в управление конфигурацией.

В этих приложениях есть примитивы, которые в Chef называются рецептами, а в Puppet — манифестами. Они сгруппированы в «книги рецептов», или «методички». Эти сборники включают в себя готовые «рецепты» с атрибутами, которые можно задействовать для переопределения значений, используемых по умолчанию в различных случаях, таких как тестирование или промышленная эксплуатация, а также схемы распространения файлов и расширения, например библиотеки и шаблоны. Конечный результат выполнения инструкций «методички» — это код, который создает определенный компонент инфраструктуры, например позволяет установить MySQL или Network Time Protocol (NTP).

Эти файлы бывают довольно сложны сами по себе и должны иметь определенные атрибуты. Они должны быть параметризованы, чтобы можно было запускать одно и то же определение в разных средах — для разработки, тестирования и эксплуатации — на основе входных данных. Действия, полученные из этих определений, и их применение также должны быть *идемпотентными*.

Идемпотентность

Идемпотентное действие — это действие, которое может выполняться неоднократно с одними и теми же результатами. Идемпотентная операция принимает на вход желаемое состояние и делает все необходимое для приведения компонента в это состояние независимо от его текущего статуса. Например, при обновлении файла конфигурации, чтобы установить размер кэша буфера, можно исходить из предположения, что такая запись в файле уже существует, что было бы наивно и чревато ошибками. Вместо этого можно вставить строку в файл. Если вставлять ее автоматически, то может оказаться, что она уже существует, и тогда строка продублируется. Таким образом, это действие не является идемпотентным.

Вместо этого можно посмотреть сценарий, чтобы проверить, существует ли такая строка. Затем, если она существует, можно ее изменить, если не существует — вставить. Это идемпотентный подход.

Пример определения конфигурации, которой требует распределенная система наподобие Cassandra, можно разбить на следующие разделы:

- основные атрибуты:
  - способ установки, местоположение, хеши;
  - имя и версия кластера;
  - групповые и пользовательские полномочия;
  - размер кучи;
  - конфигурация JVM;
  - структура каталога;
  - конфигурация сервиса;
  - конфигурация JMX;
  - виртуальные узлы;
- конфигурация и структура JBOD;
- поведение системы сборки мусора;
- определение так называемых сидов (seed);
- конфигурация файла YAML;
- конфигурации ресурсов ОС;
- внешние сервисы:
  - PRIAM;
  - JAMM (показатели Java);
  - журналирование;
  - OpsCenter;
- центр обработки данных и схема стойки.

Кроме проверки на идемпотентность и параметризацию, каждый из этих компонентов должен проходить соответствующие предварительные и заключительные тесты, а также интегрироваться в среду мониторинга и журналирования для управления ошибками и постоянного улучшения.

Предварительные и заключительные тесты включают в себя проверку того, что состояние в начале и в конце такое, как ожидается. Дополнительные тесты могут фокусироваться на использовании функций, которые могут включаться или отключаться в результате изменения, чтобы увидеть, является ли функциональное поведение ожидаемым. Мы предполагаем, что прежде, чем будет определена необходимость этих изменений, будут выполнены эксплуатационные тесты, а также тесты производительности, емкости и масштабирования. Это означает, что тестирование будет направлено на проверку работоспособности реализации и желаемого поведения. Пример реального практического примера с идемпотентностью вы найдете в блоге Salesforce Developers по адресу http://sforce.co/2zxYrjG.

Сборка из конфигурации

После того как будут определены спецификации сервера, приемочные тесты и модули для автоматизации и сборки, у вас окажется весь исходный код, необходимый для создания БД. Существует два подхода к тому, как это должно быть реализовано, они называются *«жаркой»* и *«выпечкой»*. «Выпечка», «жарка», «рецепты», «шеф-повара»... еще не проголодались? Загляните в презентацию Джона Уиллиса (John Willis) *DevOps and Immutable Delivery* (https://www.nginx.com/blog/devops-and-immutable-delivery/).

«Жарка» подразумевает динамическую настройку во время развертывания хоста. Сначала подготавливаются хосты, развертываются операционные системы и только потом выполняется настройка. Все приложения для управления конфигурацией, упомянутые в предыдущем разделе, позволяют создавать и развертывать инфраструктуру с помощью «жарки».

Например, при запуске MySQL Galera Cluster можно увидеть следующее:

- серверное оборудование подготовлено (три узла);
- операционные системы установлены;
- клиент «шеф-повар» и его «нож» (CLI) установлены, «книги рецептов» загружены;
- применены «книги рецептов» для настройки прав на уровне ОС и применения соответствующих конфигураций;
- применены «книги рецептов» для установки пакетов по умолчанию;
- создан/загружен пакет данных, который будет использоваться для атрибутов уровня кластера (IP, узел инициализации, имена пакетов);

- применены роли узла (Galera Node):
  - установлены двоичные файлы MySQL/Galera;
  - установлены пакеты и скрипты MySQL;
  - выполнена настройка базовой конфигурации;
  - выполнены тесты;
  - запущены и выключены сервисы;
  - создан кластер/настроен основной узел;
  - настроены остальные узлы;
  - выполнены тесты;
- зарегистрирован кластер в сервисах инфраструктуры.

«Выпечка» означает получение базового образа и его настройку во время сборки. Это значит, что будет создан своего рода «золотой стандарт», который станет эталоном для всех хостов, добавленных для одной и той же роли. Затем делается моментальный снимок этого образа, который сохраняется для дальнейшего использования. Примерами артефактов, полученных в результате «выпечки», являются образы Amazon AMI или виртуальных машин. В этом сценарии нет ничего динамического.

Packer — это инструмент производства Hashicorp, который создает образы. Интересно, что Packer позволяет создавать из одной конфигурации образы для разных сред, например, образы Amazon EC2 или VMWare. Большинство утилит управления конфигурацией также могут создавать «выпеченные» образы.

Поддержка конфигурации

В идеальном мире управление конфигурацией должно смягчать и потенциально даже устранять дрейф конфигурации. Дрейф конфигурации — это то, что происходит после «жарки» или «выпечки» и развертывания сервера. Сначала все экземпляры этого компонента могут быть одинаковыми, но затем люди начнут регистрироваться в системе и обязательно что-то настроят, установят, проведут несколько экспериментов и оставят после себя те или иные следы.

Неизменяемая инфраструктура — это инфраструктура, которая не может мутировать, то есть изменяться после развертывания. Если необходимо сделать изменения, то их вносят в определение конфигурации, хранящееся в системе управления версиями, и сервис развертывается повторно. Неизменяемые инфраструктуры привлекательны тем, что обеспечивают такие качества, как:

- *простота* — запрещая изменения, вы резко ограничиваете число возможных комбинаций состояний в инфраструктуре;

- *предсказуемость* — состояние всегда известно. Это означает, что все исследования выполняются намного быстрее, а при устранении неполадок все легко воспроизводится;
- *восстанавливаемость* — состояние может быть легко восстановлено путем повторного использования эталонных образов. Это значительно сокращает среднее время восстановления (MTTR). Образы известны, протестированы и готовы к развертыванию в любой момент.

Несмотря на это, неизменяемые инфраструктуры иногда сопровождаются довольно высокими издержками. Например, если у вас есть MySQL Cluster, работающий на 20 узлах, и вы хотите изменить один параметр, то после того, как изменение будет зарегистрировано, придется повторно развернуть все узлы этого кластера, чтобы добавить нововведения.

Можно выбрать золотую середину — в среде разрешить некоторые изменения, если они выполняются часто, автоматизированы и предсказуемы. Внесение изменений вручную по-прежнему запрещено, благодаря чему в значительной степени сохраняются предсказуемость и восстанавливаемость при *минимизации эксплуатационных расходов* (http://chadfowler.com/2013/06/23/immutable-deployments.html).

Применение определений конфигурации

Как же реализуются эти принципы?

Синхронизация конфигурации

Многие из рассмотренных инструментов управления конфигурацией обеспечивают синхронизацию. Это означает, что любые возникающие изменения по расписанию перезаписываются, поскольку конфигурация принудительно возвращается в стандартное состояние. Однако для этого требуется, чтобы синхронизированное состояние было как можно более полным, иначе некоторые области будут пропущены и, таким образом, конфигурация окажется подвержена дрейфу.

Перераспределение компонентов

Выявлять различия и выполнять чистое развертывание, чтобы их устранить, можно с помощью соответствующих инструментов. В некоторых средах может происходить постоянное повторное развертывание или повторное развертывание после входа в систему вручную либо какого-то взаимодействия с компонентом. Как правило, это более привлекательно для решения, созданного с помощью «выпечки», в котором устранены издержки на настройку после развертывания.

Использование определения конфигурации и управления может помочь гарантировать, что отдельные серверы или экземпляры будут правильно скомпонованы и останутся такими в процессе эксплуатации. Но есть более высокий уровень абстракции в процессе развертывания — определение межсервисных инфраструктур и оркестрация развертываний.

Определение и оркестрация инфраструктуры

Теперь, когда мы рассмотрели конфигурацию и развертывание отдельных хостов, будь то серверы, виртуальные машины или облачные экземпляры, пойдем дальше и рассмотрим группы хостов. В конце концов, мы нечасто будем управлять отдельным экземпляром базы данных, изолированным от других. Предполагая, что всегда работаем с распределенным хранилищем данных, мы должны иметь возможность создавать, развертывать и эксплуатировать несколько систем одновременно.

Инструменты оркестрации и управления для предоставления инфраструктур интегрируются с приложениями развертывания («жарки» или «выпечки»). Это позволяет создать полную инфраструктуру, включая сервисы, которые могут не использовать хост: конфигурации виртуальных ресурсов или платформы как сервиса. В идеале эти инструменты создадут единое законченное решение, которое систематизирует процесс формирования всего центра обработки данных или сервиса, предоставляя разработчикам и специалистам по эксплуатации возможность строить, интегрировать и запускать инфраструктуры от начала до конца.

Благодаря абстрагированию конфигураций инфраструктуры в виде архивируемого кода, хранящегося в системе управления версиями, эти инструменты могут интегрироваться с приложениями управления конфигурациями для автоматизации поддержки хостов и приложений. Они обеспечивают управление всеми базовыми ресурсами инфраструктуры и сервисов, необходимыми инструментам автоматизации для эффективного выполнения своих задач.

Обсуждая определения инфраструктуры, мы часто используем понятие стека. Возможно, вы слышали о стеках LAMP (Linux, Apache, MySQL, PHP) или MEAN (MongoDB, Express.js, Angular.js, Node.js). Конкретный стек может быть ориентирован на конкретное приложение или группу приложений. Стек приобретает еще более конкретное значение, если говорить об описании инфраструктур для инструментов автоматизации и оркестрации. Такое описание мы и будем иметь в виду в данном случае.

Структура стеков значительно влияет на то, как вы будете выполнять свои обязанности в качестве DBR-инженера в команде. Обсудим эти изменения и их влияние на роль DBRE.

Определение монолитной инфраструктуры

В этом стеке все приложения и сервисы, которыми владеет организация, описаны вместе, в одном большом определении. Другими словами, все кластеры базы данных определены в одних и тех же файлах. В такой среде любое количество приложений и сервисов обычно использует одну БД или несколько. В одном и том же определении вполне могут оказаться пять разных приложений со связанными базами данных.

В действительности в определении монолитной инфраструктуры нет никаких преимуществ, но есть много недостатков. С точки зрения общей оркестрации или инфраструктуры как кода все проблемы могут быть решены следующим образом.

- Чтобы внести изменения в определение, нужно его полностью протестировать. Это означает, что тесты будут медленными и неустойчивыми. В результате люди будут избегать вносить изменения, из-за чего инфраструктура станет застывшей и нестабильной.
- Изменения скорее все нарушат, чем окажутся изолированными для одного компонента инфраструктуры.
- Создание среды тестирования или разработки означает, что придется собрать вместе весь продукт, а не отдельную его часть, на которой можно было бы сосредоточиться.
- Изменения часто будет вносить небольшая группа лиц, которые знают весь стек. Из-за этого изменений может оказаться недостаточно, что замедлит скорость.

Команды могут прийти к монолитно определенному стеку, если добавят новый инструмент, такой как Terraform, и просто переведут в него всю свою инфраструктуру. При рассмотрении определения инфраструктуры выделяют горизонтальные факторы — различные уровни в пределах одного стека, а также вертикальные факторы, которые функционально разбивают стек таким образом, чтобы каждый сервис помещался в свой стек, а не в один общий.

Разделение по вертикали

Разбивая определения так, чтобы каждое из них соответствовало отдельному сервису (рис. 6.1), можно уменьшить размер и сложность определений. Таким образом то, что когда-то было одним файлом определения, может превратиться в два файла, по одному для каждого сервиса. При внесении изменений это ограничивает область отказов только одним сервисом. В результате вдвое сократится объем тестирования и пропорционально уменьшится размер среды разработки и тестирования.

Если у вас несколько приложений, использующих один и тот же уровень базы данных (что довольно распространено), это окажется более сложной задачей.

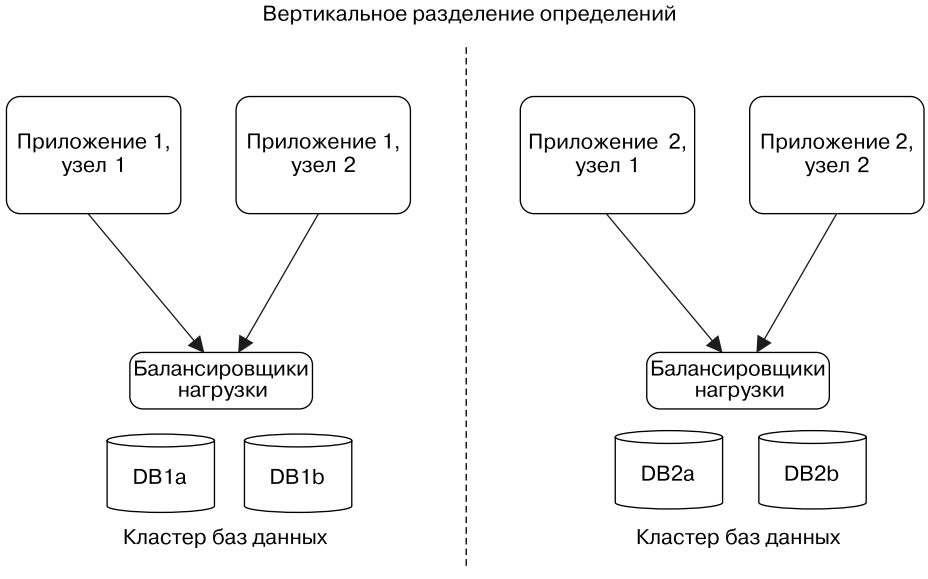


Рис. 6.1. Вертикальное разделение

На данном этапе нужно создать три определения. Одно будет определением общей базы данных, два — определениями отдельных сервисов, за исключением уровня базы данных (рис. 6.2).

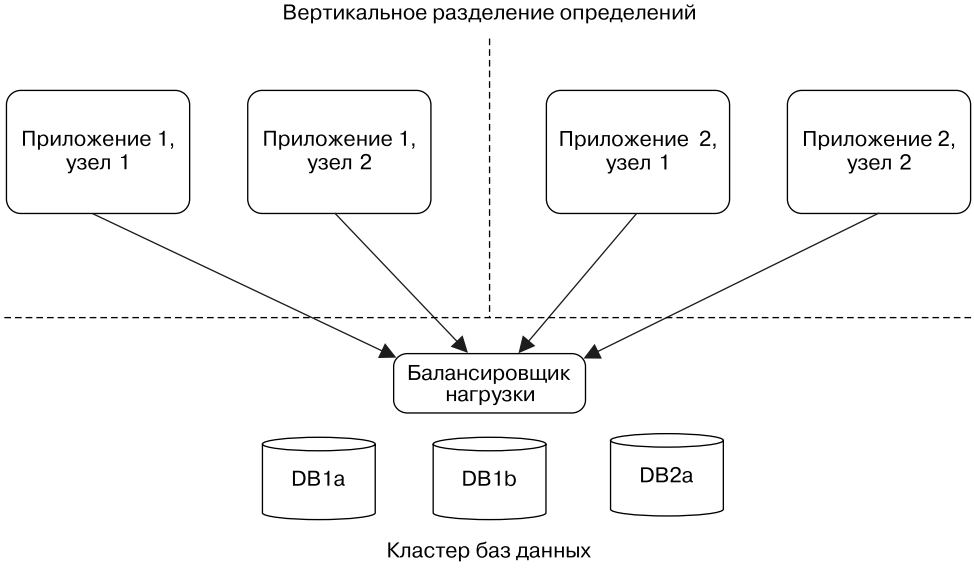


Рис. 6.2. Вертикально разделенные определения сервисов с общей базой данных

Теперь область отказов при изменениях файла определения еще больше уменьшается, потому что каждое определение меньше по размеру и точнее сфокусировано. Однако у вас по-прежнему есть два приложения, связанных с базой данных. Это означает: необходимо выполнить интегральное тестирование, чтобы убедиться, что изменения, внесенные на уровне базы данных, эффективно синхронизированы со стеками других приложений. Таким образом, тестирование по-прежнему требует сборки и развертывания всех приложений. Разделение приложений означает, что теперь у вас есть возможность создавать и тестировать приложения последовательно или параллельно в зависимости от ограничений инфраструктуры.

Разделенные уровни (горизонтальные определения)

Что касается разделенных определений приложений, то их файлы определений можно разбить по уровням, называемым *горизонтальным разделением*. Таким образом, стандартное веб-приложение может иметь стек веб-сервера, стек сервера приложений и стек базы данных. Основное преимущество такого разделения определений инфраструктуры заключается в том, что теперь мы еще больше сократили область отказов. Другими словами, если вам нужно изменить конфигурацию серверов базы данных, то не приходится беспокоиться о том, что, возможно, из-за этого будет разрушена сборка для веб-серверов.

После разделения уровней по вертикали и, возможно, по горизонтали вы столкнетесь с новыми интересными сложностями. В частности, теперь у вас есть пути коммуникации между стеками, которые требуют совместного доступа к данным. Виртуальные IP-адреса балансировщика нагрузки базы данных должны использоваться также серверами приложений, но у стеков есть собственные определения. Этой динамической инфраструктуре нужен каталог сервисов, чтобы любой ее компонент мог эффективно предоставлять доступ к своему состоянию любому другому компоненту с целью коммуникации и интеграции.

Приемочное тестирование и согласованность

Добавление автоматизации и использование инфраструктуры как кода, очевидно, дают много преимуществ. Еще один момент, о котором мы не упомянули, — это приемочное тестирование и согласованность. Имея образы инфраструктуры, можно работать с такими инструментами, как ServerSpec, в котором используется специальный язык описаний для тестов ваших образов инфраструктуры. Это распространяет принципы разработки через тестирование (test-driven development, TDD) и на инфраструктуру, открывая широкие возможности для дальнейшего сближения процессов разработки инфраструктуры и программного обеспечения.

С помощью такой инфраструктуры, как ServerSpec для автоматического тестирования, можно тщательно проработать согласованность и безопасность. Работая

с отделами обеспечения безопасности и совместимости, вы можете создать набор тестов, ориентированный на безопасность и согласованность БД. Для ServerSpec предусмотрен плагин Inspec, который довольно хорошо выполняет эту работу. Подробнее об этом читайте в блоге Chef (https://blog.chef.io/2015/11/04/the-road-to-inspec/).

Каталог сервисов

Поскольку динамические среды создаются, масштабируются и уничтожаются автоматически, должен существовать источник истины для текущего состояния, на который смогут ориентироваться все компоненты инфраструктуры. Обнаружение сервисов — это абстрактное действие, при котором устанавливается соответствие между конкретными обозначениями, номерами портов сервисов и балансировщиками нагрузки, с одной стороны, и семантическими именами — с другой. Например, суть репликаций MySQL состоит в том, что произвольное множество хостов MySQL реплицируют данные с основного хоста. Польза от создания каталога сервисов заключается в возможности обращаться к объектам не по IP-адресам и даже не по именам хостов, а по семантическим именам. Информация из каталога может быть доступной через HTTP или DNS.

Далее приведены некоторые из наиболее распространенных инструментов обнаружения сервисов, доступных на момент написания книги:

- Zookeeper;
- Consul.io;
- Etcd;
- Build your own!

Существует множество ситуаций и задач, с которыми может столкнуться DBR-инженер и в которых был бы полезен такой каталог сервисов. Вот некоторые из них (подробнее обсудим их в последующих главах, рассматривая конкретные архитектуры).

- *Отказоустойчивость базы данных*. Регистрируя в каталоге IP-адреса записи, можно создавать шаблоны для балансировщиков нагрузки. При переключении IP конфигурации балансировщика нагрузки перестраиваются и обновляются.
- *Сегментирование*. Обеспечивается совместный доступ к информации об открытых для записи сегментах для хостов приложений.
- *Узлы Cassandra Seed*. Предоставляется информация для узлов начальной загрузки, на которые нужно обращаться за сидами.

Каталог сервисов может быть очень простым, хранящим только данные о сервисе для интеграции сервисов, или же предоставлять множество дополнительных

функций, включая проверку работоспособности, чтобы гарантировать, что данные в каталоге соответствуют рабочим ресурсам. Во многих подобных каталогах также можно хранить пары «ключ — значение» (http://bit.ly/2zxRX4D).

Собираем все вместе

Итак, информации много, и в основном общей. Рассмотрим один день из жизни DBR-инженера, используя эти концепции для MySQL. Надеемся, это добавит немного важных деталей. Для простоты предположим, что система работает в среде Amazon EC2. Перед вами была поставлена задача создать новый кластер MySQL для основной пользовательской базы данных, которая должна быть изолированной и иметь большую емкость. Вам предоставили следующие инструменты:

- MySQL Community версии 5.6;
- MySQL MHA для управления репликацией и отработки отказов;
- Consul для базы данных состояния кластера.

Разумеется, файлы Terraform и сегменты Chef Cookbook для пользовательской базы данных MySQL регистрируются в GitHub. Ничто не следует изменять вручную. Вы поворчали по поводу того, что, вероятно, необходимо автоматизировать анализ емкости и развертывание новых сегментов, но вы работаете здесь всего третий месяц и у вас еще не было на это времени.

Проверяя журналы развертывания, вы увидели, когда в последний раз был развернут сегмент MySQL, и сравнили его с текущими версиями terraform- и chef-кода, чтобы убедиться, что с тех пор ничего не изменилось. Все сходится, так что все должно быть в порядке. Сначала вы запускаете Terraform с опцией `plan`, чтобы задать путь выполнения и убедиться, что ничего плохого не произойдет. Предполагая, что все прошло хорошо, вы продолжаете и запускаете команды Terraform, чтобы создать сегмент.

Terraform запускает провайдер Chef, который запрашивает у Consul самое последнее значение `shard_id` и увеличивает его на единицу. Используя это значение `shard_id`, он выполняет следующие операции.

1. Запускает три экземпляра EC2 в двух зонах доступности (AZ), задействуя соответствующие образы машин Amazon (AMI) для хостов MySQL Shard.
2. Настраивает MySQL на этих хостах и запускает сервис.
3. Регистрирует все узлы в Consul в пространстве имен `shard_id`:
   - первый из них будет зарегистрирован как главный, второй — как резервный на случай отказа;
   - начинает репликацию, используя эти данные.

4. Запускает два экземпляра EC2 в двух AZ, используя соответствующие AMI для хостов MySQL MHA Manager.
5. Регистрирует диспетчер Master High Availability (MHA) для Consul в пространстве имен `shard_id`.
6. Настраивает конфигурацию MHA, используя данные узла от Consul.
7. Запускает диспетчер репликации MHA.
8. Выполняет серию тестов восстановления после сбоя.

К текущему моменту у вас есть кластер MySQL, управляемый MHA и зарегистрированный в Consul. После этого автоматически будет выполнено следующее.

- Резервные копии автоматически запустят создание моментального снимка с мастера, так как сценарии используют Consul.
- Агенты мониторинга, находящиеся в AMI, автоматически начнут отправлять метрики и файлы журналов в стек оперативного контроля.

Наконец, когда вы будете удовлетворены сделанным, вы отметите сегмент в Consul как активный. На этом этапе прокси-серверы начинают добавлять данный сегмент в свои шаблоны и перезагружаться. Серверы приложений идентифицируют его как доступный и начинают отправлять в него данные в соответствии с указанными правилами.

Среды разработки

Локальное тестирование в среде разработки или в изолированной программной среде имеет решающее значение для рабочего процесса. Вы должны быть уверены в том, какими будут последствия изменений, прежде чем вносить их в VCS. Одной из целей всех действий, которые перечислены в этой главе, является воспроизводимость (repeatability). Это означает, что «песочница» должна как можно точнее соответствовать реальной инфраструктуре в областях программного обеспечения и конфигурации. Иными словами, одинаковыми должны быть управление конфигурацией, оркестрация, сама ОС и даже каталоги сервисов.

Обсуждая развертывание, мы упомянули Packer. Напомним: Packer позволяет создавать несколько образов из одной конфигурации. Сюда входят образы для виртуальных машин *на вашем рабочем месте*. Использование такого инструмента, как Vagrant, на рабочем месте позволяет загружать последние версии образов, создавать виртуальные машины и даже запускать стандартный набор тестов, чтобы убедиться, что все работает должным образом.

После того как вы внесли изменения, протестировали их, удалили все новые изменения из вашей VCS и протестировали снова, можете передать их обратно команде VCS для подготовки к интеграции и развертыванию.

Резюме

Умение использовать инфраструктуру как код, автоматизировать работу и выполнять контроль версий очень важно для любого инженера по надежности, и DBRE не исключение. С помощью инструментов и методов, рассмотренных в этой главе, вы можете начать устранять трудности, уменьшать количество ошибок и создавать самообслуживаемые развертывания в своей организации.

В главе 7 мы начнем углубленно изучать важнейшую функцию инфраструктуры — резервное копирование и восстановление. Одним из ключевых отличий на уровне базы данных является важность сохранения и доступности данных. Большинство других сред можно создавать как артефакты и развертывать быстро и легко, однако базы данных требуют больших рабочих наборов данных для безопасного подключения и обслуживания. Для этого предусмотрен широкий набор инструментов, о которых мы и поговорим в следующей главе.

7 Резервное копирование и восстановление

В главах 5 и 6 мы сосредоточились на проектировании инфраструктуры и управлении ею. Это означает, что вы хорошо представляете, как создавать и развертывать распределенные инфраструктуры, в которых работают базы данных, а также управлять ими. Мы рассмотрели методы быстрого добавления новых узлов для увеличения емкости или замены неисправного узла. Теперь пришло время обсудить самое главное — резервное копирование и восстановление данных.

Посмотрим правде в глаза: все считают резервное копирование и восстановление скучными и утомительными занятиями. Для большинства эти процедуры — воплощение рутины. Команде не нравится взаимодействовать с младшими инженерами и внешними подрядчиками и работать со сторонними инструментами. Прежде нам приходилось иметь дело с просто ужасным программным обеспечением для резервного копирования. Мы вам сочувствуем, честно.

Тем не менее это один из самых значимых процессов в вашей работе. Перемещение важных данных между узлами, центрами обработки данных и их перенос в долгосрочные архивы — это постоянное движение самого ценного актива вашего бизнеса — информации. Мы настоятельно рекомендуем вам не считать восстановление и резервирование операциями второго сорта, а относиться к ним как к VIP-операциям. Каждый должен не только понимать цели восстановления данных, но и быть хорошо знакомым с принципами этой работы и мониторингом процессов. В философии DevOps предполагается, что каждый должен иметь возможность писать код и внедрять его в реально работающей системе. Мы предлагаем каждому инженеру хотя бы один раз принять участие в процессах восстановления критически важных данных.

Мы создаем и храним копии данных — резервные копии и архивы — как средство удовлетворения конкретной потребности. Они нужны для восстановления. Иногда восстановление оказывается приятным и неспешным делом, например при создании среды для аудиторов или настройке альтернативной среды. Но чаще всего оно требуется для быстрой замены неисправных узлов или увеличения емкости существующих кластеров.

Сегодня в распределенных средах мы сталкиваемся с новыми проблемами в области резервного копирования и восстановления данных. Как и раньше, большинство

локальных наборов данных распределяются в разумных пределах, максимум на несколько терабайт. Различие состоит в том, что эти локальные наборы данных — лишь часть большого распределенного набора. Восстановление узла является относительно управляемым процессом, но сохранение состояния в кластере — более сложная задача.

Основные принципы

Начнем с обсуждения основных принципов резервного копирования и восстановления данных. Опытному специалисту по базам данных или системному инженеру некоторые из них могут показаться элементарными. Если это так, можете спокойно пролистать несколько страниц.

Физическое или логическое?

При физическом резервном копировании базы данных создаются резервные копии реальных файлов, в которых хранятся данные. Это означает, что поддерживаются свойственные базе данных форматы файлов, и обычно в базе есть набор метаданных, которые определяют, какие есть файлы и какие структуры БД в них находятся. Если, создавая резервные копии файлов, вы ожидаете, что другой экземпляр базы данных сможет их использовать, то вам потребуется сделать резервную копию и сохранить связанные с ней метаданные, на которые опирается эта база данных, чтобы резервная копия была переносимой.

При создании логической резервной копии данные экспортируются из базы в формат, который теоретически можно перенести в любую систему. Обычно метаданные тоже сохраняются, но, скорее всего, они будут актуальны для того момента, когда выполнялось резервное копирование. Примером является экспорт всех операторов вставки, необходимых для заполнения пустой базы данных при ее обновлении. Другой пример — строка в формате JSON. В итоге логическое резервное копирование, как правило, занимает очень много времени, потому что это не физическая операция копирования и записи, а построчное извлечение данных. Аналогично восстановление сопровождается всеми обычными издержками базы данных, такими как блокировки и создание журнальных записей повторов и отмены.

Отличный пример такого разделения — различие между репликацией на основе строк и репликацией на базе операторов. Во многих реляционных базах данных репликация на основе операторов означает, что после записи в систему контроля версий к ним добавляется журнал операторов языка манипулирования данными (DML, или вставка, обновление, замена, удаление). Эти операторы передаются в реплики, в которых воспроизводятся. Другой подход к репликации основан на строках или захвате данных изменений (Change Data Capture, CDC).

Автономное или оперативное?

Автономное (или холодное) резервное копирование — такое копирование, при котором экземпляр базы данных, использующий файлы, отключен. Благодаря этому можно быстро скопировать файлы, не беспокоясь о сохранении состояния на текущий момент, пока другие процессы читают и записывают данные. Это идеальное, но очень редко встречающееся состояние.

При оперативном (или горячем) резервном копировании вы все равно копируете все файлы, но есть дополнительная сложность, связанная с необходимостью получить согласованный моментальный снимок данных, который должен существовать то время, в течение которого выполняется резервное копирование. Кроме того, если текущий трафик обращается к базе данных во время резервного копирования, необходимо быть осторожными и следить за тем, чтобы не превысить пропускную способность подсистемы ввода-вывода на уровне хранилища. Даже ограничивая нагрузку, вы можете обнаружить, что механизмы, используемые для поддержания согласованности, вносят необоснованные задержки в работу приложения.

Полное, инкрементное и дифференциальное

Наличие полной резервной копии, независимо от того, каким методом она создана, означает, что локальный набор данных полностью зарезервирован. Для небольших наборов данных это довольно обычное дело. Для 10 Тбайт это может занять невероятное количество времени.

Дифференциальное резервное копирование позволяет создавать резервные копии только данных, измененных с момента последнего полного резервного копирования. Но на практике обычно резервируется больше данных, чем только измененные, потому что данные организованы в виде структур определенного размера — страниц. Размер страниц составляет, например, 16 или 64 Кбайт, и страница содержит много строк данных. При дифференциальном резервном копировании создается резервная копия всех страниц, на которых были изменены данные. Таким образом, при больших размерах страниц получаются резервные копии значительно большего размера, чем если бы там хранились только измененные данные.

Инкрементное резервное копирование аналогично дифференциальному, за исключением того, что в качестве момента времени, к которому относятся измененные данные, будет использоваться дата последней резервной копии, инкрементной или полной. Таким образом, при восстановлении из инкрементной резервной копии может потребоваться восстановить последнюю полную резервную копию, а также одну или несколько инкрементных резервных копий, чтобы добраться до текущего момента.

Зная это, обсудим несколько моментов, которые следует учитывать при выборе эффективной стратегии резервного копирования и восстановления данных.

Соображения по восстановлению данных

Впервые выбирая эффективную стратегию, следует снова рассмотреть свои целевые показатели качества обслуживания (SLO), которые обсуждались в главе 2. В частности, необходимо учесть показатели доступности и надежности. Какую бы стратегию вы в итоге ни выбрали, она все равно должна предусматривать возможность восстанавливать данные в рамках заранее определенных ограничений по времени безотказной работы. И вам придется выполнять резервное копирование быстро, чтобы гарантировать соответствие заданным характеристикам надежности.

Если проводить резервное копирование каждый день, а журналы транзакций между резервными копиями держать в хранилище на уровне узла, можно легко потерять эти транзакции до следующего резервного копирования.

Кроме того, необходимо учесть, как набор данных функционирует в рамках целостной экосистемы. Например, ваши заказы могут храниться в реляционной базе, где все зафиксировано в виде транзакций и, следовательно, легко восстанавливается по отношению к остальным данным, хранящимся в этой БД. Однако, после того как заказ сформирован, рабочий процесс может запускаться событием, сохраненным в системе очередей или хранилище типа «ключ — значение». Эти системы могут обеспечивать целостность данных лишь эпизодически или даже кратковременно (быть эфемерными), ссылаясь при необходимости на реляционную базу или используя ее для восстановления. Как учесть эти рабочие процессы при восстановлении?

Если вы имеете дело со средой, где ведется быстрая разработка, то может оказаться, что данные, сохраненные в резервной копии, были записаны и использованы одной версией приложения, а после восстановления выполняется другая. Как приложение будет взаимодействовать с устаревшими данными? Хорошо, если данные версионированы — тогда это можно учесть, но вы должны знать о такой возможности и быть готовыми к подобным случаям. Иначе приложение может логически повредить эти данные, что приведет к еще большим проблемам в будущем.

Все эти и многие другие нюансы, которые невозможно предсказать, необходимо учитывать при планировании восстановления данных. Как говорилось в главе 3, невозможно подготовиться к любой ситуации. Но очень важно постараться это сделать. Восстановление данных — одна из главнейших обязанностей инженера по обеспечению надежности БД. Таким образом, ваш план восстановления данных должен быть максимально широким и учитывать как можно больше потенциальных проблем.

Сценарии восстановления

Приняв все сказанное во внимание, обсудим типы происшествий и операций, которые могут потребовать восстановления данных, чтобы можно было спланировать удовлетворение всех потребностей. Сначала можно разделить все сценарии на плановые и незапланированные. Рассматривая восстановление данных лишь как

инструмент для разрешения чрезвычайных ситуаций, вы ограничите инструментарий своей команды только средствами неотложной помощи и имитации аварий. И наоборот, если включить восстановление данных в повседневную деятельность, то можно ожидать более высокой степени осведомленности и успешного разрешения чрезвычайных ситуаций. Кроме того, у нас появится больше данных, позволяющих определить, поддерживает ли стратегия восстановления наши SLO. При нескольких ежедневных прогонах сценария будет проще получить набор образцов, который включает в себя предельные значения и который можно довольно уверенно использовать для планирования.

Сценарии планового восстановления

В какие повседневные задачи можно внедрить процессы восстановления? Вот список, который чаще всего встречался нам на разных сайтах:

- создание новых узлов и кластеров в среде промышленной эксплуатации;
- создание различных сред;
- выполнение извлечения, преобразования и загрузки (Extract, Transform and Load, ETL) и стадий технологического процесса обработки данных для последовательно размещенных хранилищ;
- проведение эксплуатационных тестов.

При выполнении этих операций обязательно включите процесс восстановления в стек оперативного контроля. Учитывайте следующие показатели.

- *Время*. Сколько времени занимает выполнение каждого компонента и всего процесса? Распаковка? Копирование? Выполнение журнала? Тестирование?
- *Размер*. Сколько места занимает резервная копия, сжатая и несжатая?
- *Пропускная способность*. Какая нагрузка на оборудование создается?

Эта информация поможет вам избежать проблем с пропускной способностью, что позволит обеспечить стабильность процесса восстановления.

Новые узлы и кластеры в среде промышленной эксплуатации

Независимо от того, являются ваши базы данных частью неизменной инфраструктуры или нет, существуют возможности для регулярных перестроек, при которых по мере необходимости будут использоваться процедуры восстановления.

Базы данных редко включаются в автоматическое масштабирование систем из-за того количества времени, которое может потребоваться для начальной загрузки нового узла и его помещения в кластер. Тем не менее нет никаких причин, меша-

ющих команде создать расписание регулярного ввода новых узлов в кластер для тестирования этих процессов. Chaos Monkey (http://bit.ly/2zy1qsE) — инструмент, разработанный Netflix, который случайным образом отключает системы, позволяет сделать это таким образом, чтобы можно было протестировать весь процесс мониторинга, выдачи уведомлений, сортировки и восстановления. Если вы еще этого не сделали, то можете включить это в план контрольного списка процессов, которые ваш отдел эксплуатации должен выполнять через регулярные промежутки времени, чтобы все сотрудники ознакомились с процедурой. Эти действия позволяют протестировать не только полное и инкрементное восстановление, но и включение в поток реплицирования и процесс ввода узла в эксплуатацию.

Создание разных сред

Вы неизбежно будете создавать среды разработки, интеграционного и оперативного тестирования для демонстрационных и других целей. Некоторые из этих сред требуют полного восстановления данных, и в них необходимо реализовать восстановление узлов и полное восстановление кластера. Часть сред предъявляют другие требования, такие как поддержка частичного восстановления для тестирования функций и очистка данных в целях обеспечения конфиденциальности пользователя. Это позволяет тестировать восстановление данных на определенный момент времени, а также восстановление конкретных объектов. Все это сильно отличается от стандартного полного восстановления и бывает полезно для восстановления повреждений, вызванных действиями оператора и ошибками приложений. Создавая API, которые обеспечивают восстановление данных на уровне объекта и на определенный момент времени, вы можете облегчить автоматизацию и ознакомление работников с этими процессами.

ETL и конвейерные процессы для хранилищ данных, расположенных далее по конвейеру

Как и для задач построения среды, процессы и API восстановления из моментальных снимков или на уровне отдельных объектов могут быть успешно применены при передаче данных из рабочих БД в конвейеры для последующей аналитики и в потоковые хранилища.

Эксплуатационное тестирование

В процессе выполнения различных сценариев тестирования вам понадобятся копии данных. Одни тесты, например для емкости и нагрузки, требуют полного набора данных, что отлично подходит для полного восстановления. При функциональном тестировании могут потребоваться меньшие наборы данных, что позволит выполнить восстановление на определенный момент времени и на уровне объекта.

Само тестирование восстановления данных может стать непрерывной операцией. Кроме применения процессов восстановления данных в повседневных сценариях, можно настроить постоянное выполнение операций восстановления. Это позволит автоматизировать тестирование и валидацию, чтобы быстро выявлять любые проблемы, которые могут возникнуть при нарушении процесса резервного копирования. Когда заходит речь о реализации этого процесса, многие спрашивают о том, как проверить успешность восстановления.

При создании резервной копии можно получить много данных, которые затем использовать для тестирования, например:

- самый последний идентификатор в последовательности автоматического приращения;
- счетчик строк для объектов;
- контрольные суммы для подмножеств данных, которые только вставляются и, следовательно, могут рассматриваться как неизменяемые;
- контрольные суммы в файлах определения схемы.

Как и при любом тестировании, подход должен быть многоуровневым. Существует ряд тестов, которые либо будут выполнены успешно, либо быстро закончатся сбоем. Это должен быть первый уровень тестирования. Примеры — сравнение контрольных сумм по определениям метаданных/объектов, успешный запуск экземпляра базы данных и успешное подключение к потоку репликации. Операции, которые могут занять больше времени, такие как вычисление контрольных сумм и подсчет таблиц, должны выполняться позже в ходе проверки.

Незапланированные сценарии

С учетом всех повседневных плановых сценариев, которые можно использовать, процесс восстановления данных должен быть хорошо отлажен, документирован, отработан и в достаточной степени свободен от ошибок и проблем. Благодаря этому незапланированные сценарии редко оказываются настолько пугающими, насколько могли бы быть. Команда не должна видеть разницы между плановым и незапланированным восстановлением. Перечислим и подробно рассмотрим ситуации, которые могут потребовать от нас выполнения процессов восстановления:

- ошибка пользователя;
- ошибка приложения;
- наличие инфраструктурных сервисов;
- ошибки операционной системы и аппаратные ошибки;
- аппаратные сбои;
- сбои центра обработки данных.

Ошибка пользователя

В идеале ошибки пользователя должны возникать редко. Выстраивая для инженеров «перила» и «заграждения», вы можете предотвратить многие подобные ситуации. Тем не менее всегда остается вероятность того, что оператор случайно нанесет ущерб. Типичный пример — везде и всеми забываемое условие `WHERE` при выполнении `UPDATE` или `DELETE` в клиентском приложении базы данных. Или, например, выполнение сценария очистки данных не в тестовом окружении, а в рабочей «производственной» системе. Зачастую операция выполняется правильно, но в неподходящее время или не для тех хостов. Все это относится к ошибкам пользователя. Часто их выявляют и исправляют сразу же. Однако иногда последствия таких изменений остаются незамеченными в течение нескольких дней или недель.

Ошибки приложений

Ошибки приложений — самый страшный из обсуждаемых сценариев, поскольку они бывают очень коварными. Приложения постоянно изменяют способ взаимодействия с хранилищами данных. Многие приложения также управляют ссылочной целостностью и внешними указателями на такие ресурсы, как файлы или сторонние идентификаторы. Страшно представить, что будет, если просто внести изменение, которое портит данные, удаляет их или добавляет неверные данные способами, которые могут остаться незамеченными в течение довольно длительного времени.

Инфраструктурные сервисы

В главе 6 мы познакомились с магией сервисов управления инфраструктурой. К сожалению, эти системы могут оказаться столь же разрушительными, сколь и полезными, что может привести к широкомасштабным последствиям, связанным с редактированием файла, указанием на другую среду или неправильной настройкой конфигурации.

Ошибки ОС и аппаратные ошибки

Операционные системы и оборудование, с которым они взаимодействуют, — это тоже системы, созданные людьми, и, таким образом, в них возможны ошибки, которые могут иметь неожиданные последствия из-за недокументированных или плохо известных конфигураций. В контексте восстановления данных это особенно верно в отношении пути передачи данных из БД через кэши ОС в файловые системы, контроллеры и в итоге на диски. Повреждение или потеря данных случаются гораздо чаще, чем мы думаем. К сожалению, наше доверие к этим механизмам

и опора на них порождают ожидание целостности данных вместо скептического отношения к ней.

Незаметное повреждение данных

Именно такая ошибка ОС и оборудования произошла в Netflix в 2008 году. Для обнаружения и исправления ошибок на дисках использовался код исправления ошибок (ECC). ECC автоматически ликвидирует однобитовые ошибки и обнаруживает двухбитовые. Таким образом, ECC-код позволяет обнаружить ошибку, вдвое превышающую расстояние Хэмминга, которую он способен исправить. Следовательно, он может исправить 46 байт на жестком диске 512-байтового сектора и обнаружить до 92 байт ошибки. То, что невозможно исправить, передается контроллеру как неисправимое, и контроллер диска увеличивает значение счетчика «неисправимая ошибка» в S.M.A.R.T. Ошибки размером свыше 92 байт передаются прямо в контроллер как хорошие данные. Это касается и резервных копий. Страшно, правда?

Именно поэтому следует с большим скептицизмом относиться к облачным и так называемым серверным вычислениям. Если у вас нет доступа к деталям реализации, то вы не можете быть уверены в том, что целостность данных будет главным приоритетом. Слишком часто она игнорируется или даже настраивается так, чтобы скорость имела более высокий приоритет. Нет знаний — нет силы.

Инструменты вычисления контрольных сумм для файловых систем, такие как ZFS, будут проверять контрольную сумму каждого блока, гарантируя обнаружение неверных данных. Если же использовать RAID-массив, что подразумевает зеркалирование или проверку четности, то он даже исправит данные.

Аппаратные сбои

Аппаратные компоненты выходят из строя в принципе, а в распределенных системах это случается регулярно. Вы постоянно сталкиваетесь со сбоями дисков, памяти, процессоров, контроллеров и сетевых устройств. Следствиями этих аппаратных сбоев могут стать выход из строя узлов или задержки на узлах, из-за чего система становится непригодной для использования. Системы общего доступа, такие как сетевые устройства, способны влиять на целые кластеры, делая их недоступными или разбивая на более мелкие кластеры, которым неизвестно о том, что сеть оказалась разделена. Это может привести к быстрому и значительному расхождению данных, которые необходимо объединить и исправить.

Сбои центра обработки данных

Иногда неполадки оборудования на сетевом уровне приводят к сбоям в центре обработки данных. Бывает так, что перегрузка соединительных панелей хранения данных вызывает каскадные сбои, как в случае с веб-сервисами Amazon в 2012 году (http://bit.ly/2zxSpzR). Иногда ураганы, землетрясения и другие катаклизмы приво-

дят к тому, что выходят из строя целые центры обработки данных. Последующее восстановление проверит на прочность даже самые надежные стратегии восстановления.

Область действия сценария

Перечислив запланированные и незапланированные сценарии, которые могут вызвать необходимость восстановления, добавим к этим событиям еще одно измерение, чтобы наше представление стало объемным. Это будет полезно для выбора самого подходящего способа реагирования. Рассмотрим следующие варианты:

- сбой, локализованный в рамках одного узла;
- сбой всего кластера;
- сбой, затронувший весь центр обработки данных или несколько кластеров.

В случае *локального*, или *одноузлового*, сбоя восстановление ограничивается одним хостом. Возможно, вы добавите в кластер новый узел для увеличения емкости или замены неисправного узла. Или же в системе реализовано непрерывное обновление и тогда восстановление будет выполнено узел за узлом. В любом случае это *локальное восстановление*.

В *масштабе кластера* необходимость восстановления является глобальной для всех членов этого кластера. Возможно, произошло деструктивное изменение или удаление данных, которое каскадно распространилось на все узлы. Или же вам понадобилось ввести новый кластер для тестирования емкости.

Если случился сбой *в масштабах центра обработки данных* или *нескольких кластеров*, значит, необходимо восстановить все данные в месте их физического размещения или по всей области сбоя. Это может быть связано с отказом общего хранилища данных или со сбоем, вызвавшим катастрофический отказ центра обработки данных. Такое восстановление может потребоваться также при плановом развертывании нового дополнительного сайта.

Кроме области действия, существует область набора данных. Здесь можно перечислить три возможных варианта:

- один объект;
- несколько объектов;
- метаданные базы данных.

В *масштабах одного объекта* только этот конкретный объект требует восстановления данных — некоторых или всех. Обсуждавшийся ранее случай, в результате которого при выполнении операции `DELETE` удалялось больше данных, чем

планировалось, относится к сбою в рамках одного объекта. При сбое *нескольких объектов* оказываются затронуты несколько или, возможно, все объекты в конкретной базе данных. Такое может произойти при повреждении приложения, неудачном обновлении или миграции сегмента. Наконец, бывают сбои в масштабах *метаданных базы*, когда с данными, хранящимися в базе, все в порядке, но теряются метаданные, делающие ее пригодной для использования, такие как пользовательские данные, полномочия безопасности или соответствия файлам ОС.

Последствия сценария

Важно не только определить сценарий, требующий восстановления, и выделить область сбоя, но и определить возможные последствия, поскольку при выборе подхода к восстановлению они будут значительными. Если потеря данных не влияет на SLO, можете выбрать методичный и медленный подход, позволяющий свести к минимуму расширение последствий. К более глобальным изменениям, которые приводят к нарушению SLO, следует подходить осторожно, выбирая быстрое восстановление обслуживания и только потом переходя к долгосрочной очистке. Можно разделить все подходы на следующие три категории.

- Воздействие на SLO, отказ приложений, затронуто большинство пользователей.
- Угроза SLO, некоторые пользователи пострадали.
- Затронуты функции, не угрожающие SLO.

Учитывая сценарий восстановления, масштабы и последствия сбоя, мы можем выбрать сочетание из 72 различных сценариев. Это много! На самом деле слишком много, чтобы уделить каждому из них необходимое внимание. К счастью, во многих сценариях можно использовать один и тот же подход к восстановлению. Тем не менее даже при таком совпадении невозможно полностью спланировать любую ситуацию. Таким образом, нужно предусмотреть многоуровневый подход к восстановлению данных, чтобы иметь как можно больше инструментов. В следующем разделе мы применим информацию, полученную только что, чтобы определить стратегию восстановления.

Содержание стратегии восстановления

Почему мы говорим «стратегия восстановления», а не «стратегия резервного копирования»? На это есть причина. Восстановление данных — та самая причина, по которой мы делаем резервные копии. Резервные копии — это просто средство для достижения цели, а она зависит от истинного требования: восстановления с заданными параметрами. Ответом на простой вопрос: «Создана ли резервная копия

базы данных?» — всегда должно быть «Да, несколькими способами в зависимости от сценария восстановления». Простое «да» звучит наивно и вызывает ложное чувство безопасности, что безответственно и опасно.

Эффективная стратегия восстановления базы данных не только предусматривает несколько сценариев с наиболее эффективными подходами, но и включает в себя обнаружение потери или повреждения данных, а также тестирование и валидацию восстановления.

Структурный блок № 1: обнаружение

Раннее обнаружение потенциальной потери или повреждения данных имеет решающее значение. Обсуждая ошибки пользователей и приложений в подразделе «Незапланированные сценарии» предыдущего раздела, мы отметили, что эти проблемы часто актуальны несколько дней, недель или даже дольше, прежде чем их обнаруживают. Это означает, что к тому времени, когда будет замечена потребность в резервных копиях, они могут оказаться устаревшими. Таким образом, обнаружение должно быть очень значимым для всех инженеров. Кроме раннего обнаружения потери или повреждения данных, крайне важно обеспечить максимально широкое временно́е окно, в течение которого возможно восстановить данные в том случае, если система раннего обнаружения не сработает. Рассмотрим описанные ранее сценарии сбоев и определим некоторые реальные подходы к их обнаружению и расширению окна восстановления.

Ошибка пользователя

Один из самых важных факторов, позволяющих сократить время на выявление потери данных, — это исключение изменений в производственной среде, вносимых вручную или по ситуации. Создавая обертки для сценариев или даже абстракции уровня API, инженеры должны предпринимать действенные меры, позволяющие гарантировать, что все изменения максимально безопасны, протестированы, зарегистрированы и будут переданы в соответствующие отделы компании.

Эффективная обертка или API обеспечивают:

- выполнение кода в различных средах с использованием параметризации;
- пробный запуск, при котором можно оценить и провести валидацию результатов выполнения;
- набор тестов для выполнения кода;
- валидацию после выполнения, чтобы убедиться, что изменения оправдали ожидания;

- мягкое удаление результатов или просто возврат к предыдущему состоянию посредством того же API;
- протоколирование по ID всех изменений в данных с возможностью идентификации и восстановления.

Исключая из этих процессов компоненты, подразумевающие операции, выполняемые по ситуации или вручную, можно повысить вероятность того, что все изменения будут отслеживаться инженерами по устранению неполадок. Все изменения будут записываться, что обеспечит отслеживаемость событий, так что их нельзя будет просто потерять в повседневной рутине. И наконец, выполняя постепенные изменения и удаления и создавая простые процедуры возврата в предыдущее состояние для любых данных, вы даете больше времени на выявление и исправление проблем, вызванных изменением данных. Это, конечно, не гарантия. В конце концов, процессы, выполняемые вручную, могут быть чрезвычайно хорошо журналированы, а люди способны забыть настроить журналирование в автоматизированных процессах или же пропустить их.

Ошибки приложений

Ключевым условием раннего обнаружения ошибок приложений является валидация данных. Когда разработчики вводят новые объекты и атрибуты, инженеры по надежности БД должны тесно сотрудничать с ними, чтобы выбрать способ валидации данных, которая может быть выполнена на выходе, вне самого приложения.

Как и при любом тестировании, сначала необходимо сосредоточиться на быстрых тестах, которые образуют быстрые циклы обратной связи для критически важных компонентов данных, таких как внешние указатели на файлы, сопоставление отношений для обеспечения ссылочной целостности и информации для идентификации пользователей (Personal Identification Information, PII). По мере увеличения количества данных и развития приложений валидация становится все дороже и ценнее. Формирование корпоративной культуры, при которой инженеры несут ответственность за качество и целостность данных, а не за механизмы их хранения, приносит дивиденды не только в смысле гибкости использования различных БД, но и в том, что люди чувствуют себя более уверенно, когда экспериментируют и быстро переходят к новым функциям приложений. Валидация играет роль защитного ограждения, помогая каждому чувствовать себя смелее и увереннее.

Инфраструктурные сервисы

Любые катастрофические воздействия на инфраструктуру, в результате которых потребуется восстановление, нужно быстро обнаруживать путем мониторинга стека оперативного контроля. При этом существуют изменения, которые происходят

незаметно и потенциально могут привести к потере данных, их повреждению или потере доступности. Использование эталонных образов и регулярное сравнение с ними компонентов инфраструктуры может помочь быстро выявить отклонения от тестовых образов. Аналогично использование инфраструктуры с поддержкой версий позволит выявить отклонения и предупредить соответствующих специалистов или помочь автоматизировать восстановление.

Ошибки ОС и аппаратные ошибки

Как и в случае с инфраструктурными сервисами, большинство этих проблем следует быстро выявлять в ходе мониторинга журналов и показателей. Существуют граничные случаи, которые не относятся к стандартным. Чтобы их идентифицировать и проводить мониторинг для их раннего обнаружения, нужно обладать некоторым опытом, к тому же потребуется все хорошо обдумать. Типичный пример — контрольные суммы на дисковых блоках. Не все файловые системы их вычисляют, и отделам, работающим с критически важными данными, стоит найти время и изучить подходящие файловые системы, способные идентифицировать скрытые повреждения посредством вычисления контрольных сумм.

Сбои оборудования и ЦОД

Как и в случае с инфраструктурными сервисами, эти сбои должны легко идентифицироваться с помощью мониторинга, рассмотренного в главе 4. Разве не здорово, что мы уже сделали это?

Структурный блок № 2: многоуровневое хранилище

Эффективная стратегия восстановления основана на размещении данных по нескольким уровням хранения. Различные варианты восстановления могут обслуживаться разными областями хранения, что обеспечивает не только нужную скорость, но и нужный уровень затрат и надежность для любого количества сценариев.

Онлайновое высокоскоростное хранилище

Это область памяти, в которой будет работать большинство хранилищ данных в производственной среде. Она характеризуется высокой пропускной способностью, низкими задержками и, следовательно, высокими затратами. Если время восстановления имеет решающее значение, размещение последних версий копий хранилища данных и связанных с ним инкрементных резервных копий на этом уровне имеет первостепенное значение. Вообще говоря, здесь будут находиться

лишь несколько копий последних версий данных, что позволяет быстро восстанавливать данные для наиболее распространенных и эффективных сценариев. Типичными вариантами использования будут полные копии базы данных для ввода в промышленную эксплуатацию новых узлов сервиса после сбоев или в ответ на резкий рост трафика, которые требуют дополнительной емкости.

Онлайновое низкоскоростное хранилище

Эта область хранения часто применяется для данных, нечувствительных к задержке. Такие хранилища представлены дисками большего размера с низкой пропускной способностью и высокими задержками, зато и более дешевыми. Поэтому такие области хранения часто намного больше, и, следовательно, на этом уровне можно хранить больше копий данных, относящихся к более позднему времени. Эти старые резервные копии будут довольно редко использоваться сценариями восстановления небольшой важности или с длительным временем выполнения. Типичными вариантами применения таких хранилищ являются поиск и исправление тех ошибок приложений или пользователей, которые были упущены при раннем обнаружении.

Автономное хранилище

Примерами такого типа хранилищ являются хранилища данных на магнитной ленте или даже что-то вроде Amazon Glacier. Они расположены удаленно, и часто их нужно переносить на каком-то носителе, чтобы доставить туда, где они будут доступны для восстановления. Такое хранилище может поддерживать непрерывность ведения бизнеса и требования аудита, но не использоваться в повседневных сценариях восстановления данных. Тем не менее благодаря размеру и стоимости здесь доступны огромные объемы, что позволяет хранить все данные за все время существования предприятия или по крайней мере в течение полного срока, предписанного законодательством.

Хранилище объектов

Хранилище объектов — это такое хранилище, которое управляет данными, представленными в виде объектов, а не файлов или блоков. Хранилище объектов обеспечивает функции, недоступные в традиционных архитектурах хранения, например предоставляет API, доступный для приложений, управление версиями объектов, высокую степень доступности посредством репликации и распространения. Хранилище объектов обеспечивает масштабируемую и самовосстанавливающуюся доступность большого количества объектов с учетом всех их версий и истории изменений. Оно может быть идеальным вариантом, позволяющим легко восстанавливать определенные объекты, которые не являются структурированными и для

обеспечения согласованности которых не требуется взаимосвязь с другими данными. Это привлекательная возможность для восстановления ошибок приложений и пользователей. Классический пример недорогого, масштабируемого и надежного хранилища на уровне объектов — Amazon S3.

Каждый из этих уровней играет важную роль в комплексной стратегии восстановления данных для нескольких возможных сценариев. Поскольку у нас нет возможности предусмотреть все допустимые сценарии, необходима именно такая степень разнообразия. Далее мы обсудим инструменты, которые используют эти уровни хранения для обеспечения возможности восстановления данных.

Структурный блок № 3: разнообразие инструментария

Итак, пора выбрать необходимые процессы восстановления, перебрав разные сценарии и оценив варианты. Ранее в главе вы узнали, что существует ряд доступных инструментов. Рассмотрим их подробнее.

Репликация — это не резервное копирование!

Возможно, вы заметили, что мы нигде не обсуждаем репликацию как способ эффективного резервного копирования данных для их восстановления. Репликация — слепой процесс, она может каскадно распространять ошибки пользователей, ошибки приложений и повреждения. Репликацию следует рассматривать как инструмент для перемещения и синхронизации данных, но не для создания полезных артефактов восстановления. Если кто-то вам скажет, что использует репликацию для резервного копирования, посмотрите на него неодобрительно и идите своей дорогой. RAID также не является резервной копией — это средство обеспечения избыточности.

Полное физическое резервное копирование

Мы знаем, что понадобится выполнить полное восстановление данных на всех уровнях области действия: уровнях узла, кластера и центра обработки данных. Быстрое, переносимое, полное восстановление — это невероятно мощный механизм, обязательный в динамических средах. Полное резервное копирование позволяет быстро создавать узлы для увеличения емкости или развертывания замещающих узлов при сбоях. Полное резервное копирование можно выполнить с помощью полных копий данных по сети или с использованием томов, которые можно легко подключить к соответствующим хостам/экземплярам и отключить от них. Для этого необходимы полные резервные копии.

Для полного резервного копирования реляционной базы данных необходимо иметь возможность либо заблокировать ее для получения согласованного

снимка, из которого можно создавать копию, либо отключить на время копирования. В асинхронно реплицируемой среде нельзя полностью доверять репликам при синхронизации с первичным устройством записи, поэтому следует при создании полных резервных копий копировать данные с основного устройства, если это вообще возможно. Создав моментальный снимок базы данных, файловой системы или инфраструктуры, можно скопировать его в промежуточное хранилище.

Полные резервные копии добавляемого хранилища данных записи, такого как Cassandra, включают в себя моментальный снимок с явным обращением к именам файлов в файловой системе (жесткие ссылки — *hard links*)[1]. Поскольку данные в этих распределенных хранилищах имеются не на всех узлах, резервное копирование считается *в конечном счете согласованным*. Для восстановления потребуется вернуть узел в кластер, после чего регулярные операции согласованности в итоге приведут его в актуальное состояние.

Полное резервное копирование в высокоскоростном онлайновом хранилище применяется для немедленной замены онлайн-кластера. Эти резервные копии, как правило, являются распакованными, так как распаковка занимает много времени. Полные резервные копии в онлайн-хранилище с низкой скоростью используются для создания различных сред, таких как среды для тестирования, аналитики и экспертизы данных. В качестве эффективного инструмента, позволяющего увеличить время полного резервного копирования в ограниченных пулах хранения, применяется сжатие.

Инкрементное физическое резервное копирование

Как говорилось ранее, инкрементное резервное копирование позволяет сократить разрыв между последней полной резервной копией и реальным состоянием через некоторое время. Инкрементные физические резервные копии обычно создаются из блоков данных, в которых содержится измененный фрагмент. Поскольку полное резервное копирование может быть затратным как с точки зрения влияния на скорость резервного копирования, так и с точки зрения хранения, инкрементное резервное копирование позволяет быстро привести в соответствие полную резервную копию, которая может быть устаревшей для использования в кластере.

---

[1] Имеется в виду характерная для файловых систем Unix концепция: имя файла в каталоге ссылается на описывающие файл служебные структуры. Соответственно, имя принято называть hard link — «жесткая ссылка». Жесткие ссылки действуют только в пределах одной файловой системы (одного логического раздела), поэтому зависят от монтирования файловых систем. Альтернатива — мягкие, или символические, ссылки (symbolic link). — *Примеч. науч. ред.*

Полное и инкрементное логическое резервное копирование

Полное логическое резервное копирование обеспечивает переносимость и более простое извлечение подмножеств данных. Этот вид копирования не применяется для быстрого восстановления узлов, но является идеальным инструментом для экспертизы, перемещения данных между хранилищами и восстановления определенных подмножеств данных из больших наборов.

Хранилища объектов

Хранилища объектов, такие как логические резервные копии, позволяют легко восстанавливать определенные объекты. В сущности, хранилища объектов оптимизированы именно для этого конкретного случая использования, и их удобно задействовать в API для программного восстановления объектов по мере необходимости.

Структурный блок № 4: тестирование

Удивительно, как часто тестирование такого важного инфраструктурного процесса, как восстановление данных, отходит на второй план. Тестирование — это важная процедура, которая гарантирует, что резервные копии пригодны для восстановления. Тестирование часто настраивается как случайный процесс, который выполняется периодически, например ежемесячно или ежеквартально. Это лучше, чем ничего, но между тестами допускается слишком долгий период, в течение которого резервные копии могут перестать работать.

Существует два эффективных способа внедрить тестирование в текущие процессы. Первый подразумевает внедрение восстановления в повседневные процессы. Таким образом, восстановление проверяется постоянно, что позволяет быстро выявлять ошибки и сбои. Кроме того, при регулярном восстановлении формируются данные о том, сколько времени занимает восстановление, что необходимо для калибровки процессов восстановления в соответствии с соглашениями об уровне качества обслуживания (SLA). Далее перечислены примеры постоянной интеграции восстановления в повседневные процессы:

- создание интеграционных сред;
- построение сред тестирования;
- регулярная замена узлов в кластерах производственной среды.

Если в среде недостаточно возможностей для восстановления хранилищ данных, можно запланировать непрерывный процесс тестирования, при котором восстановление последней резервной копии будет постоянным процессом, подразумевающим

последующую проверку успешности восстановления. Независимо от наличия автоматизации даже внешние уровни резервного копирования требуют периодического тестирования.

С помощью этих структурных блоков можно настроить глубокую защиту для различных сценариев восстановления данных. Составляя план сценариев и применения инструментов для их восстановления, можно начать оценивать потребности с точки зрения разработки и ресурсов.

Определение стратегии восстановления

Как говорилось ранее в этой главе, существует несколько сценариев отказов, к которым нужно быть готовыми. Для этого нужен широкий набор инструментов и план применения каждого из них.

Онлайновое быстрое хранилище с полными и инкрементными резервными копиями

Эта часть стратегии предусматривает самое главное в ежедневном восстановлении данных. Именно она используется, когда нужно построить новый узел для быстрого внедрения в среде промышленной эксплуатации или тестирования.

Примеры применения

Следующие сценарии являются основными вариантами использования этой части стратегии:

- замена неисправных узлов;
- ввод новых узлов в промышленную эксплуатацию;
- создание тестовых сред для интеграции функций;
- создание сред для операционного тестирования.

Полное резервное копирование, как правило, выполняется один раз в день — делать это чаще невозможно из-за задержки, возникающей в процессе резервного копирования. Если делать так в течение недели, то можно будет быстро получить доступ к любым недавним изменениям, обычно этого более чем достаточно. Это означает семь полных копий базы данных, несжатых, плюс данные, необходимые для отслеживания всех изменений при создании инкрементных резервных копий. Бывает, что для этого нет достаточных мощностей или денег, поэтому в качестве элемента настройки можно изменить *срок хранения копий* и *частоту копирования*.

Обнаружение

Система мониторинга информирует о сбое узла или компонента, требующего восстановления на новых узлах. Обзоры и прогнозы планирования емкости позволяют узнать, когда нужно добавить новые узлы, чтобы увеличить емкость.

Многоуровневое хранилище

Вам требуется высокоскоростное онлайн-хранилище, поскольку сбои в производственной среде следует устранять быстро. Тестировать тоже нужно максимально быстро, чтобы поддерживать высокую скорость разработки.

Инструментарий

Полные и инкрементные физические резервные копии обеспечивают самое быстрое восстановление и лучше всего подходят в данном случае. Поскольку время восстановления — критический параметр, эти резервные копии хранятся в несжатом виде.

Тестирование

Интеграционное тестирование выполняется часто, поэтому такие сценарии восстановления тоже будут регулярными. В виртуальных средах ежедневное повторное введение одного узла в кластер подразумевает столь же частое восстановление. Наконец, непрерывный процесс восстановления введен из-за своей большой важности.

Онлайновое медленное хранилище с полными и инкрементными резервными копиями

Это более медленное хранилище с более дешевым пространством большего объема.

Примеры применения

Основными вариантами использования этой стратегии являются следующие сценарии:

- выявление ошибок приложений;
- обнаружение ошибок пользователей;
- устранение повреждений;
- создание сред для тестирования операций.

При появлении новых функций, неудачных изменений или миграций, которые могут повредить данные, необходимо иметь доступ к последним и извлекать большие их объемы для восстановления. Вот тут-то и вступает в игру данный вариант. Это, пожалуй, самый неприятный этап восстановления, потому что необходимо учесть слишком много перестановок, способных причинить ущерб. В ходе восстановления часто приходится писать код, что само по себе при отсутствии эффективного тестирования может вызвать множество ошибок и дефектов.

Простой способ получить полные резервные копии в этой части стратегии — копирование полных резервных копий из высокоскоростного в низкоскоростное хранилище с использованием механизма сжатия. Из-за сжатия и меньшей стоимости хранения возможно хранить данные месяц или даже дольше в зависимости от бюджета и потребностей. В высокодинамичных средах вероятность отсутствия проблем с повреждением и целостностью данных гораздо выше, а следовательно, необходимо рассчитывать на более длительное время.

Обнаружение

Валидация данных — главное условие для определения необходимости восстановления из данного пула. При сбое валидации инженеры могут использовать резервные копии, чтобы определить, что и когда произошло, и приступить к извлечению «чистых» данных для повторного применения в производственной среде.

Многоуровневое хранилище

Необходимо онлайновое низкоскоростное хранилище, поскольку эта часть стратегии требует значительного времени. Нужно дешевое и большое хранилище.

Инструментарий

Здесь лучше всего подходят полные и инкрементные физические резервные копии. Они также сжимаются в зависимости от времени, требующегося на восстановление. Здесь можно использовать не только физические, но и логические резервные копии, такие как журналы репликации, чтобы обеспечить бóльшую гибкость при восстановлении.

Тестирование

Поскольку такое восстановление происходит не очень часто, непрерывные автоматизированные процессы восстановления имеют решающее значение для того, чтобы все резервные копии были полезными и находились в хорошем состоянии.

Периодически проводимые «учения» для отработки определенных сценариев, таких как восстановление таблицы или диапазона данных, помогут командам освоиться с процессами и инструментарием.

Автономное хранилище

Это определенно самый медленный уровень хранилища, из которого можно извлекать данные.

Примеры применения

Основными случаями использования этой части стратегии являются следующие сценарии:

- аудит и проверка на согласованность;
- непрерывное ведение бизнеса.

Таким образом, эта часть решения действительно ориентирована на редкие, но крайне важные потребности. Аудит и проверка на согласованность часто требуют данных за последние семь лет и более. Но эти операции нечувствительны ко времени выполнения, их подготовка и реализация могут оказаться продолжительными. Для непрерывного ведения бизнеса требуется размещать копии данных за пределами физических носителей действующих производственных систем, чтобы обеспечить возможность восстановления при стихийных бедствиях. Эта операция требует много времени, но восстановление можно выполнять поэтапно, что дает гибкость.

Перенос полных резервных копий из низкоскоростного хранилища в автономное с помощью механизма сжатия — простой способ получить полные резервные копии в этой части стратегии. Хранение данных в течение семи лет и более не только возможно, но и необходимо.

Обнаружение

Обнаружение не является существенной частью данного компонента стратегии.

Многоуровневое хранилище

Здесь понадобится дешевое хранилище больших размеров, потому что реализация этой части стратегии требует длительного времени. Подойдут магнитная лента или решения наподобие Amazon Glacier.

Инструментарий

Здесь лучше всего применять полные резервные копии. Поскольку время восстановления довольно велико, они сжимаются.

Тестирование

Стратегии тестирования в этом случае аналогичны тем, что используются на уровне онлайнового медленного хранилища.

Хранилище объектов

Пример хранилища объектов — Amazon S3. Оно характеризуется программным, а не физическим доступом.

Примеры применения

Основными вариантами использования этой части стратегии являются следующие сценарии:

- выявление ошибок приложений;
- обнаружение ошибок пользователей;
- устранение повреждений.

Для интеграции в приложения и инструменты администрирования с целью эффективного восстановления после пользовательских ошибок и ошибок приложений инженерам-программистам предоставляются API проверки, размещения и поиска хранилищ объектов. При использовании управления версиями это становится довольно тривиальным процессом для восстановления после удалений, непредвиденных изменений и других потенциальных ситуаций, заставляющих администраторов зря терять время, если у них нет этих инструментов.

Обнаружение

Валидация данных и пользовательских запросов имеет решающее значение для определения необходимости восстановления из этого источника. При сбое валидации инженеры могут определить диапазоны дат возникновения события и выполнить программное восстановление после инцидента.

Тестирование

Поскольку восстановление на уровне объектов становится частью приложения, для обеспечения работоспособности последнего должно быть более чем достаточно стандартного интеграционного тестирования.

Используя эти четыре подхода к восстановлению данных, мы можем обеспечить достаточную комплексную стратегию восстановления данных при большинстве сценариев, даже при тех, которые мы не ожидаем или не планируем. Необходима точная настройка этой схемы, в зависимости от ожиданий уровня восстановления, бюджета и ресурсов. Но в целом мы заложили основу для эффективного плана, который включает в себя обнаружение, сбор показателей, отслеживание и непрерывное тестирование.

Резюме

Заканчивая читать эту главу, вы уже должны хорошо понимать, какие для вашей среды существуют риски, а также что может потребовать восстановления данных. Имя этим рискам легион, и они непредсказуемы. Одним из самых важных моментов является то, что невозможно заранее все спланировать и необходимо разработать комплексную стратегию, которая позволит справиться с любой ситуацией. Некоторые из них подразумевают сотрудничество с разработчиками программного обеспечения, чтобы можно было включить восстановление данных в само приложение. В других случаях вам придется самостоятельно создать довольно надежное программное обеспечение для восстановления. И всегда для реализации восстановления вы должны будете опираться на предыдущие главы этой книги, посвященные управлению уровнем качества обслуживания, рисками, инфраструктурой, а также построению инфраструктуры.

В главе 8 мы обсудим управление релизами. Надеемся, что в оставшейся части книги восстановление данных останется в центре вашего внимания. Каждый шаг вперед в развитии приложений и инфраструктуры несет в себе риски для данных и сервисов с отслеживанием состояния. Основное правило для DBR-инженера гласит: обеспечивайте возможность восстановления данных.

8 Управление релизами

По мере внедрения автоматизации, облегчающей бремя управления инфраструктурой, инженер по обеспечению надежности БД сможет уделять больше времени своим самым важным обязанностям. Одной из них является работа с инженерами-программистами над созданием, тестированием и развертыванием функций приложений. Прежде администратор БД был чем-то вроде вахтера при вводе продукта в эксплуатацию. Он следил за каждой миграцией базы данных, определением объекта в базе, кодом, обращающимся к базе данных, чтобы убедиться, что все было сделано правильно. Если администратор оставался доволен увиденным, он планировал соответствующее изменение, которое выполнялось вручную, и запускал его в производственной среде.

Вы, вероятно, уже догадались, что это не самый стабильный процесс для тех сред, в которых происходит значительное количество развертываний и изменений структур базы данных. На самом деле, если вам уже приходилось выполнять подобную работу, вы и сами хорошо знаете, как быстро администратор БД (DBA) может превратиться из вахтера в помеху, что приведет к перегрузке на стороне DBA и срыву разработки программного обеспечения.

Цель этой главы — показать, как DBR-инженер может эффективно использовать свое время, навыки и опыт для поддержки разработки программного обеспечения, в которой применяется непрерывная интеграция (Continuous Integration, CI) и даже непрерывное развертывание (CD), и не стать слабым звеном.

Обучение и сотрудничество

В первую очередь DBR-инженер должен информировать разработчиков о хранилищах данных, с которыми они имеют дело. Если разработчики смогут лучше выбирать структуры данных, соответствующий SQL и общие стратегии взаимодействия, то будет меньше потребностей в прямом вмешательстве DBRE. Взяв на себя роль проводника знаний по базам данных для разработчиков, вы можете

существенно влиять на процессы непрерывного обучения коллег. Это улучшает взаимоотношения и коммуникацию, способствует сохранению доверия и в целом имеет решающее значение для успешной технической организации процесса.

Внесем ясность: мы выступаем не за то, чтобы DBR-инженер не вмешивался в действия команды разработчиков программного обеспечения. Напротив, предлагаем, чтобы он регулярно взаимодействовал с другими специалистами и прилагал значительные усилия для создания компетентной команды, которая имеет доступ к ресурсам и способна по большей части автономно принимать решения, связанные с базами данных.

Помните: все, что вы делаете, должно быть конкретным, измеряемым и действенным. Выберите ключевые параметры, определяющие успешность команды в этом смысле, и то, как вы будете реализовывать стратегии и изменения, посмотрите, как они помогают команде. Вот выбранные ключевые показатели, которые необходимо учитывать в этом процессе:

- количество ситуаций, когда потребовалось взаимодействие с DBRE;
- соотношение успешных и неудачных развертываний БД;
- скорость введения функционала — то, как быстро разработчики способны дать законченное готовое решение;
- время простоя, вызванное изменениями в БД.

Гибкая методология и культура DevOps требуют перекрестных взаимодействий между специалистами из разных областей, с разным уровнем квалификации и, конечно же, профессионального контекста для тесного сотрудничества. Обучение и сотрудничество являются важной частью этого процесса и открывают большие возможности, поскольку DBR-инженер выходит из устаревшего режима DBA и становится неотъемлемой частью технической организации.

Станьте источником знаний

Вскоре вы непременно обнаружите, что погрузились в блоги, твиты и социальные сети людей и организаций, которые, по вашему мнению, играют исключительно важную роль в мире данных и БД. Там вы найдете статьи, сессии вопросов и ответов, подкасты и проекты, которые как-то относятся к тому, чем занимаетесь вы и ваши коллеги, и имеют значение в этой области. Отслеживайте эти публикации и делитесь ими. Оформите регулярную рассылку новостей, создайте форум или даже канал в чате, чтобы размещать соответствующую информацию и выносить ее на обсуждение. Покажите команде инженеров, что вы и другие DBRE вкладываете силы в их успех и дальнейшее развитие.

Развивайте общение

Следующий шаг — выстраивание активного диалога и взаимодействия с разработчиками программного обеспечения. Именно с этого момента вы и команда разработчиков углубляетесь в соответствующий контент, которым делитесь, чтобы генерировать идеи, учиться применять информацию и даже улучшать ее, выявляя пробелы и объединяясь для дальнейшего изучения и экспериментов. Это можно сделать несколькими способами, выбор которых в значительной степени зависит от культуры обучения и совместной работы, принятой в вашей организации. Вот несколько примеров:

- еженедельные технические переговоры;
- неофициальные встречи;
- онлайн-сессии типа «вопрос — ответ»;
- чат-канал, ориентированный на обмен знаниями.

Вы можете также предложить коллегам задавать вам вопросы и в нерабочее время общаться на конкретные темы и вместе изучать их.

Знания из конкретной области

Рассмотренные компоненты позволяют сформировать основы работы и дают знания, относящиеся к соответствующим хранилищам данных и архитектурам, используемым в вашей организации, однако по-прежнему остается необходимость в передаче знаний, связанных с конкретной предметной областью вашей организации.

Архитектура

Мы не сторонники статичной документации, не связанной с процессами, обеспечивающими построение и внедрение наших архитектур. Благодаря системам управления конфигурацией и оркестрации вы получите много бесплатной документации, которая остается актуальной долгое время. Добавив к ней инструменты, которые позволят легко находить, заимствовать, комментировать и добавлять к публикациям аннотации, вы скомпонуете динамичную, всегда актуальную документацию для различных отделов компании.

К этому добавляется возможность понимать текущее состояние и прошедшие события. Все хранилища данных, конфигурации и топологии создаются по конкретной причине. Все это помогает инженерам выяснить, какая это архитектура, почему используется именно она, где найти документацию о том, как с ней взаимодействовать, и, наконец, какие были достигнуты соглашения и компромиссы, чтобы получилось то, с чем мы сейчас имеем дело.

Ваша работа как DBR-инженера состоит в том, чтобы предоставить доступ к информации о текущем состоянии и истории происходивших изменений инженерам, которые ежедневно принимают решения, разрабатывая функции без вашего участия. Создав базу знаний проектных документов, вы построите структуру, необходимую для понимания прошлого и настоящего вашей архитектуры. Эти документы могут относиться к целым проектам, требующим новых архитектурных компонентов, или к более мелким поступательным изменениям либо подпроектам. Например, вам определенно пригодится проектный документ, который отражал бы переход от логической репликации к построчной, но этот процесс не обязательно будет соответствовать тем же требованиям, что и первая инсталляция Kafka, поддерживающая создание распределенного файла журнала для событийно-ориентированных архитектур.

Создание и распространение шаблонов для этих документов — дело всей команды. Однако очень важно проследить, чтобы в них вошли такие информационные разделы.

- *Краткое содержание*. Для тех, кому нужна только основная информация.
- *Цели и антицели*. Что ожидается от этого проекта? Что выходит за его рамки?
- *Базовые знания*. Контекст, который может понадобиться читателю.
- *Структура*. Может включать как общие сведения, так и довольно подробное описание. Необходимо привести диаграммы, примеры конфигураций или алгоритмы.
- *Ограничения*. О чем необходимо помнить и какие проблемы решить, например: соответствие требованиям PCI, специфические потребности IaaS или потребности в персонале.
- *Альтернативы*. Вы рассмотрели другие варианты? Какую методологию использовали и почему отказались от других вариантов?
- *Детали запуска*. Как это развертывалось? Какие возникли проблемы и как с ними справились? Здесь также описываются сценарии и процессы, а также приводятся комментарии.

Как видите, эти документы могут быть достаточно объемными. Для некоторых проектов это нормально. Распределенные системы и многоуровневые сервисы довольно сложны и подразумевают много дополнительных сведений, которые следует изучить. Помните, что в данном случае главная цель — предоставить инженерам эти сведения, чтобы вам не требовалось тратить больше времени, чем необходимо.

Модель данных

Не менее важной является информация о типе хранимых данных. Информируя разработчиков программного обеспечения о том, какие данные уже хранятся и где их найти, вы можете значительно уменьшить избыточность и сократить время на

выяснение этого в процессе разработки. Кроме того, это позволит вам поделиться информацией о представлении одних и тех же данных в различных парадигмах — реляционных, парах «ключ — значение» и документоориентированных. Кроме того, это позволяет давать рекомендации в тех случаях, когда хранилища данных не подходят для определенных видов данных.

Рекомендуемые методы и стандарты

Предоставление инженерам стандартов для той работы, которую они регулярно выполняют, — еще один эффективный метод оптимизировать свою деятельность и сделать ее более полезной. Это можно делать постепенно, помогая инженерам и принимая решения. Вот несколько примеров того, что можно добавить:

- стандарты типов данных;
- индексирование;
- атрибуты метаданных;
- хранилища данных для использования;
- отслеживаемые показатели;
- паттерны проектирования;
- миграция и паттерны изменения БД.

Публикация этих параметров во время работы с инженерами позволяет создать базу знаний самообслуживания (https://martinfowler.com/articles/evodb.html), доступную в любое время, вместо того чтобы заставлять команду разработчиков регулярно обращаться к вам для решения этих вопросов.

Инструменты

Дать инженерам-программистам эффективные инструменты для разработки — ваша главная задача. Возможно, вы помогаете им с инструментами сравнения и сценариями, средствами оценки согласованности данных, шаблонами или даже конфигураторами для новых хранилищ данных. В конце концов, то, что вы делаете, позволяет ускорить ход разработки, одновременно высвобождая ваше время для более важных дел.

Вот несколько отличных примеров таких инструментов:

- Etsy's Schemanator;
- Percona Toolkit, особенно для изменения схемы по сети;
- пакеты для настройки и оптимизации SQL;

- SeveralNines Cluster Configurator;
- опробованные и включенные в базу паттерны и примеры планов модификации;
- опробованные и включенные в базу примеры сценариев и паттернов миграции;
- наборы тестов для простого тестирования, визуализации и анализа.

Отнеситесь к команде разработчиков программного обеспечения как к своим клиентам и практикуйте расчетливый подход к развитию продукта. Предоставьте им надежный инструментарий, позволяющий выполнять их работу, и регулярно проводите собеседования, отслеживайте и изучайте их успехи, неудачи, болевые точки и пожелания. Это поможет понять, какие инструменты принесут им наибольшую пользу.

Сотрудничество

Если вы регулярно проводите обучение, создаете инструменты и расширяете возможности инженеров, то между вами, естественно, установятся хорошие отношения. Это очень важно, потому что ведет к развитию сотрудничества. Любой инженер-программист должен иметь возможность связаться с командой DBRE, чтобы запросить информацию или совместно поработать над задачей. Это очень важно для обеих команд, так как инженеры-программисты узнают больше о том, как работает команда DBRE и чем занимаются входящие в нее специалисты, а команда DBRE подробнее знакомится с процессом разработки программного обеспечения.

DBRE могут этому способствовать, активно поддерживая контакты с программистами. Бывают случаи, когда ход проекта сильно зависит от разработки и рефакторинга БД. Именно в таких случаях DBRE должны направить свои усилия на то, чтобы гарантировать успех и эффективность этой работы. Попросите себе помощника или станьте частью команды разработчиков. Аналогично отслеживание миграций, сохраняемых в главной ветви системы контроля версий, поможет команде DBRE определить, где следует выполнять проверки.

Безусловно, уверенность в том, что DBRE не закрылись в своей скорлупе и не отдалились от написания кода, поможет гарантировать, что это сотрудничество действительно есть. Этому способствует взаимное участие DBR-инженеров и программистов в проектах и задачах друг друга.

Ранее мы уже обсудили способы помочь программистам быть максимально самодостаточными в процессе разработки. По мере увеличения команды разработчиков вам понадобится эффективно проводить обучение, использовать стандарты и инструментарий, чтобы гарантировать, что разработчики принимают

правильные решения, не нуждаясь в вашем непосредственном вмешательстве. Одновременно сообщайте инженерам о тех случаях, когда нужно рассмотреть предстоящие изменения и принять решения, а вы можете вовремя подключиться и помочь.

Далее мы обсудим, как DBRE может эффективно поддерживать различные этапы процесса доставки. Хотя непрерывная доставка (Continuous Delivery, CD) никоим образом не является новой концепцией, организации до сих пор бьются над интеграцией в этот процесс и баз данных. В следующих разделах поговорим о том, как успешно внедрить уровни базы данных в полный цикл доставки.

Интеграция

Частая интеграция изменений в базе данных означает меньшие, лучше управляемые наборы изменений и более быстрые циклы обратной связи, так как можно быстрее выявлять критические изменения. Многие организации стремятся к непрерывной интеграции (CI), позволяющей автоматически вносить все зарегистрированные изменения. Основная ценность CI заключается в возможности автоматизированных тестов, которые позволяют убедиться в том, что база данных соответствует всем ожиданиям приложения. Эти тесты запускаются каждый раз, когда изменения кода фиксируются в системе контроля версий.

В течение всего жизненного цикла разработки ПО любое изменение в коде или компонентах базы данных должно вызывать создание новой сборки с последующими интеграцией и тестированием. Вы и команда разработчиков программного обеспечения отвечаете за установку рабочих определений базы данных. При интеграции происходит постоянное подтверждение того, что БД поддерживает рабочее состояние, пока разработчики программного обеспечения выполняют рефакторинг модели данных, вводят новые наборы данных, а также находят новые интересные варианты запросов к базе.

Выполнение CI на уровне базы данных оказывается очень сложным. В дополнение к функциональным аспектам всех приложений, использующих объекты базы данных, есть свойственные промышленной эксплуатации требования к доступности, согласованности, времени отклика и безопасности. Изменения объектов могут влиять на хранимый код (в том числе функции, триггеры и представления) и даже на запросы из других частей приложений. Кроме того, расширенные функции, такие как выдача событий, могут создавать дополнительные уязвимости в базах данных. Даже после тестирования функциональности остается множество граничных случаев, потенциально затрагивающих целостность данных. Иногда целостность можно обеспечить ограничениями, однако и эти правила необходимо протестировать. Еще более подозрительны случаи, когда нет никаких ограничений.

Предпосылки

Чтобы установить CI на уровне базы данных, необходимо выполнить пять требований. Рассмотрим их.

Система контроля версий

Как и в случае с кодом и конфигурациями инфраструктуры, все миграции БД должны быть зарегистрированы в той же системе управления версиями (VCS), что и остальная часть приложения. Крайне важно собирать приложение на основе последних конфигураций, чтобы понять, какими новыми интересными способами недавние изменения конфигурации БД способны повредить или разрушить сборку приложения.

При таком риске избыточности в коде должно быть проверено все, включая следующее:

- миграции объектов БД;
- триггеры;
- процедуры и функции;
- представления;
- конфигурации;
- образцы наборов данных для функциональности;
- сценарии очистки данных.

Таким образом, помимо CI, мы получаем еще много полезного:

- все взаимосвязанные элементы расположены в одном месте, и их там легко найти;
- поддерживаются все автоматизированные сборки, необходимые для автоматического развертывания (см. главу 6);
- сохраняются вся история и все версии базы данных, что помогает при восстановлении, проведении экспертизы и устранении неполадок;
- версии приложения и базы данных с гарантией синхронизированы, по крайней мере в идеальном случае.

Если вы будете выполнять интеграцию, проверяя вносимые в систему контроля версий изменения в коде и инфраструктуре в известном рабочем состоянии, то инженеры-программисты смогут использовать последние версии БД в своих средах разработки.

Автоматизация сборки базы данных

Если вы используете методы управления и автоматизации конфигурации, описанные в главе 6, то можете автоматически создавать базы данных для интеграции. Сюда относятся применение новейших сценариев языка описания структур данных (Data Definition Language, DDL) и загрузка репрезентативных наборов данных для тестирования. Это может оказаться более сложным, чем ожидалось, потому что данные из производственной среды часто необходимо очищать или фильтровать, чтобы избежать проблем с раскрытием информации о клиентах.

Тестовые данные

Пустые базы данных почти всегда работают очень хорошо. Небольшие наборы данных тоже часто работают хорошо. Вам понадобятся три разных набора данных. Во-первых, все метаданные, необходимые для таблиц поиска. Сюда входят идентификаторы для типов клиентов, идентификаторы местоположения, рабочий процесс и внутренние таблицы. Эти наборы данных, как правило, невелики и имеют решающее значение для правильной работы приложения.

Затем вам понадобится рабочий набор функциональных данных, например, о клиентах и заказах. Как правило, этого достаточно, чтобы успешно выполнить быстрые тесты на ранних этапах интеграции, прежде чем начать тратить время на более интенсивное тестирование.

Наконец, вам понадобятся большие наборы данных, чтобы представить, как все выглядит при реальной нагрузке. Обычно такие наборы создают из реальных рабочих наборов данных, очищая их, чтобы гарантировать, что не будут раскрыты данные клиентов, не будут случайно отправлены электронные письма тысячам пользователей или использованы другие интересные и замечательные возможности для взаимодействия с клиентами и юридическими лицами.

Метаданные и тестовые наборы данных должны быть версионированы, зафиксированы и использованы как часть сборок. Большие наборы данных часто поступают из производственной среды, и сценарии, необходимые для восстановления и очистки данных, должны быть версионированы и зафиксированы в системе контроля версий, чтобы была обеспечена синхронизация между приложением и уровнем базы данных.

Миграции и упаковка базы данных

Все сказанное основано на предположении, что изменения базы данных задействуются как миграции (поступательные изменения кода). Каждый набор изменений, такой как таблица изменений, добавление метаданных или нового семейства столбцов, будет зарегистрирован и получит порядковый номер. Поскольку все изменения

применяются последовательно, у каждой версии базы данных в любое время есть номер, полученный на основе последней миграции.

Традиционно администраторы БД либо получали список изменений от разработчиков, либо составляли *список различий схемы* между средой разработки и промышленной средой, чтобы получить информацию, необходимую для внедрения нужных изменений в релизе. Преимущество этого подхода заключается в том, что специалисты по базам данных могут очень тщательно управлять крупными, потенциально очень важными изменениями. Это позволяет свести к минимуму возможные простои и последствия при дорогостоящих миграциях.

Однако у такого традиционного подхода есть отрицательная сторона: иногда бывает сложно увидеть, какие изменения каким функциям соответствуют. Если потребуется что-то вернуть назад, то может быть сложно идентифицировать, какая версия из постепенно изменявшейся базы данных относится к конкретной функции. Точно так же если изменение базы данных завершится неудачно, то все функции, ожидающие этих изменений, будут приостановлены, что повлияет на время внедрения изменений в производственной среде.

При инкрементном подходе наблюдается все, что и должно наблюдаться при гибком подходе: быстрый выход на рынок, небольшие поступательные изменения, прозрачный контроль и быстрая обратная связь. Но это означает, что разработчики должны быть лучше осведомлены о том, как создавать безопасные миграции и когда следует звать на помощь команду DBRE. Кроме того, существует риск конфликта между миграциями. Если два разработчика изменят одни и те же объекты, то их миграции будут выполняться последовательно, что может привести к двум изменениям вместо одного. Если в объекте содержится много данных, может значительно увеличиться время миграции. В этих случаях необходимо учитывать компромиссы, то есть разработчики должны знать, что иногда они могут наступать друг другу на пятки.

Сервер CI и среда тестирования

Предполагается, что при интеграции вашего приложения все указанное в заголовке уже используется. Хорошая CI-система обеспечивает всю необходимую функциональность для интеграции. Среды тестирования предоставляют как тесты системного уровня, так и тесты компонентов кода.

На системном уровне доступны такие среды тестирования, как Pester для Windows и Robot для Linux. Кроме того, можно использовать Jepsen (http://jepsen.io/) — платформу для тестирования распределенных систем, специально созданную для проверки согласованности и безопасности данных в распределенном хранилище.

При таких условиях можно начать использовать платформу CI для миграции БД в вашей компании. Как следует из названия, непрерывная интеграция означает, что

каждый раз, когда происходит изменение базы данных, интеграция выполняется автоматически. Чтобы это произошло и чтобы команда разработчиков была уверена, что изменения не повлияют отрицательно на функциональность приложения и ожидаемый уровень качества обслуживания, ключевым инструментом становится тестирование.

Тестирование

Итак, все наши разработчики фиксируют все изменения, вносимые ими в базу данных, в VCS. Сервер CI позволяет выполнять автоматические сборки базы данных, синхронизированные с релизами приложений, и у вас есть среда тестирования. Что дальше? Необходимо убедиться, что интеграция работает, и определить, какие последствия она будет иметь на очередном этапе — при развертывании.

К сожалению, мы вынуждены сказать: это сложно! Как известно, изменения в базе данных влияют на огромное количество кода и функциональности. Тем не менее существуют способы создания приложений, позволяющие упростить дело.

Приемы разработки с тестированием

Есть множество способов упростить тестирование при построении процессов разработки. Приведем два примера.

Абстракция и инкапсуляция

Существует множество способов абстрагировать доступ к базе данных для разработчиков. Зачем это нужно? Наличие кода для централизованного доступа к базе предоставляет стандартный понятный способ реализации новых объектов и обращения к ним. Это также означает, что вам не придется искать код по всей кодовой базе при необходимости внести изменения в БД. Это значительно упрощает тестирование и интеграцию. Есть несколько средств для реализации такой абстракции:

- объекты доступа к данным (Data Access Objects, DAO);
- API и веб-сервисы;
- хранимые процедуры;
- специальные фреймворки.

С их помощью при интеграции можно сконцентрироваться на тестировании примитивов, связанных с доступом к данным и обновлением данных, чтобы определить, повлияли ли изменения на возможность их использования. Как и при любом тестировании, вам прежде всего потребуются быстрые и эффективные тесты, а централизованный код доступа к данным значительно облегчает эту задачу.

Эффективность

Часто встречаются разработчики, которые используют запрос `"select *"` или извлекают для работы всю строку объекта. Они так поступают с прицелом на будущее или чтобы гарантировать, что им всегда будут доступны все необходимые данные. Возможно, они хотят быть уверенными, что если к объекту добавится еще один атрибут, то он автоматически будет извлекаться. Это не только опасно, но и, как все, что делается на будущее, расточительно и подвергает риску приложения во время изменений. Запрос `"select *"` позволит извлечь все столбцы, и если код не готов к их обработке, то он будет поврежден. Кроме того, все извлеченные данные должны быть переданы по сети, что требует большей пропускной способности, и если извлекаются несколько строк, то может получиться слишком много TCP-пакетов. Избирательность при извлечении данных из базы имеет решающее значение. Всегда можно изменить код доступа к объекту, когда для этого наступит время, и в тот момент, как это произойдет, вы будете к этому готовы.

Тестирование после фиксации в VCS

Целью тестирования после фиксации в VCS является проверка того, что изменения успешно применяются и приложение не было повреждено. Кроме того, на этом уровне можно анализировать последствия и выполнять валидацию на основе правил для обеспечения безопасности и соответствия. После того как код будет зафиксирован в VCS, сервер сборки приложений должен сразу же создать хранилище данных интеграции, применить изменения и выполнить серию тестов, способных обеспечить довольно быстрый цикл обратной связи с разработчиками. Это означает быструю сборку БД с использованием проверенного минимального набора данных, содержащего все необходимые метаданные, учетные записи пользователей и тестовые данные, нужные для выполнения соответствующих функций на всех объектах доступа к данным. Это позволяет разработчикам быстро убедиться, что они не повредили сборку.

На начальных этапах существования организации многое из этого реально делать вручную. По мере введения правил можно применять инструменты и средства автоматизации, чтобы сделать эти процессы более быстрыми и безопасными.

Перед сборкой

Перед тем как изменения вступят в силу, можно выполнить валидацию на соответствие установленным правилам для анализа последствий и соответствия:

- валидацию правильности формирования SQL-запросов;
- валидацию количества строк, которые могут быть изменены;
- валидацию создания индекса для новых столбцов;

- валидацию того, что к новым столбцам в таблицах с существующими данными не применяются значения по умолчанию;
- валидацию влияния на хранимый код и ссылочные ограничения.

Кроме того, можно создать отчет об обновлении важных объектов и атрибутов базы данных и отчет о том, были ли нарушены правила, необходимые для соблюдения соответствия.

Во время сборки

При реализации сборки снова выполняется валидация SQL-запросов, в этот раз не на базе анализа правил, а на основании фактического применения изменений.

После сборки

После того как изменения внесены в сборку, можно запустить наборы функциональных тестов. Можно также формировать отчеты о результатах анализа последствий и любых нарушениях правил, которые произошли в результате внесения изменений.

Тестирование на полном наборе данных

Предполагается, что после того, как приложение начнет работать с полным набором реальных рабочих данных, появится риск, что сервис перестанет соответствовать ожиданиям уровня качества обслуживания. Это означает, что набор тестов должен запускаться для наборов реальных рабочих данных при соответствующих нагрузках. Для этого требуются дополнительная подготовка и ресурсы, поэтому выполнение этого набора тестов можно запланировать асинхронным относительно стандартных интеграционных тестов. В зависимости от частоты интеграции и сохранения кода в хранилище наиболее подходящим для этих тестов может оказаться еженедельное или даже ежедневное выполнение.

Этапы этого обширного тестирования могут быть различными, но обычно выполняются в такой последовательности.

1. Предоставление экземпляра хранилища данных и приложения.
2. Развертывание кода.
3. Восстановление полного набора данных.
4. Анонимизация данных.
5. Сбор показателей.
6. Применение изменений к хранилищу данных.

7. Выполнение функциональных быстрых тестов.
8. Выполнение нагрузочных тестов с наращиванием количества параллельно выполняющихся задач.
9. Удаление экземпляров.
10. Анализ результатов тестов.

Далее перечислено кое-что из того, что можно узнать в результате этих тестов:

- изменения задержки в тестах, по сравнению с предыдущими запусками, на меньших наборах данных;
- изменения пути доступа к базе данных в оптимизаторах, которые могут повлиять на задержку или использование ресурсов;
- метрики базы данных, указывающие на потенциальное воздействие на производительность или функциональность (блокировка, ошибки, ожидание доступа к ресурсам);
- изменения использования ресурсов по сравнению с предыдущими запусками тестов.

Некоторые этапы анализа, такие как регистрация запросов в централизованном хранилище данных и сравнение изменений плана, можно автоматизировать. Часть из них, например анализ показателей, потребуют от оператора эффективного пересмотра, чтобы определить, насколько приемлемы те или иные изменения.

При появлении любых тревожных звоночков, как автоматизированных, так и неавтоматизированных, команда DBRE может сузить круг изменений, требующих анализа, просматривая только те, что были применены с момента последнего запуска теста. Это не позволяет немедленно выделить конкретное изменение, зафиксированное в VCS, но дает возможность намного быстрее идентифицировать его.

Кроме быстрого и медленного анализа приложений, существуют дополнительные тесты, которые должны периодически выполняться для быстро развивающихся хранилищ данных. Они гарантируют, что развивающаяся база данных останется «законопослушным членом» экосистемы. Это нисходящее и эксплуатационное тестирование.

Нисходящее тестирование

Нисходящее тестирование используют для того, чтобы убедиться, что никакие конвейеры данных и потребители хранилища данных не будут подвергаться неблагоприятному воздействию каких-либо изменений, применяемых в процессе миграции. Как и тестирование полного набора данных, нисходящее тестирование

лучше всего выполнять асинхронно независимо от процесса фиксации изменений в VCS. Далее представлены несколько примеров нисходящих тестов:

- валидация рабочих процессов событий, вызванных данными в БД;
- валидация извлечения, преобразования и загрузки данных в аналитические хранилища;
- валидация пакетных и запланированных заданий, которые напрямую взаимодействуют с базой данных;
- валидация того, что время выполнения заданий не увеличилось значительно, что могло бы повлиять на своевременную доставку для случаев доставки с заданным сроком или для последующих процессов.

Подобно тестированию на полном наборе данных, эти тесты часто являются гораздо более обширными и требуют больших наборов данных. Если выполнять их асинхронно, но регулярно, будет легче определить потенциальные изменения, которые повлияли на последующие процессы, отмеченные при тестировании. Если тесты не пройдены, то запуск в эксплуатацию может быть приостановлен, и в случае нарушения правил могут быть автоматически сгенерированы и поставлены в очередь задания для DBRE.

Операционные тесты

По мере увеличения наборов данных и развития схем операционные процессы могут занимать больше времени и, возможно, давать сбои. Тестирование этих процессов включает в себя:

- тестирование процессов резервного копирования и восстановления;
- тестирование отказоустойчивости и кластерных процессов;
- тестирование конфигурации инфраструктуры и оркестрации;
- тестирование безопасности;
- тестирование производительности.

Эти тесты необходимо выполнять регулярно при создании автоматических сборок на основе наборов реальных данных, причем все ожидаемые и зафиксированные в VCS изменения должны быть применены до запуска тестов. Наличие неудачных тестов сообщит серверу сборки о том, что существует проблема, которая должна быть исследована и устранена, прежде чем изменения будут переданы в производственную среду. Изменения базы данных реже затрагивают эти процессы, однако они могут серьезно повлиять на уровень качества обслуживания и, следовательно, требуют осмотрительности.

Сочетая непрерывные легкие сборки и стратегически запланированные более интенсивные тесты, можно удостовериться в надежности разработки при создании

и эксплуатации программного обеспечения и управлении им, в итоге изменения базы данных могут быть безопасно введены в промышленную эксплуатацию без непосредственного вмешательства команды DBRE.

Эти интеграционные процессы — отличный пример того, какую большую пользу могут принести DBR-инженеры благодаря совместным действиям, обмену знаниями и автоматизации, если позволят инженерам-программистам не ограничивать свои возможности. В следующем разделе мы обсудим самую острую проблему, о которой обычно не говорят, — развертывание. Признание того, что изменение базы данных безопасно, — это только первый шаг. Не менее важно безопасно вносить эти изменения в производственной среде.

Развертывание

В предыдущем разделе, посвященном интеграции, мы затронули концепцию миграции БД, а также некоторые достоинства и недостатки этого процесса. Поскольку мы уже обсудили, насколько значительными они могут быть, имеет смысл разделить миграцию данных на этапы таким образом, чтобы разработчики могли легко и постепенно вносить изменения в среду безопасным способом. Или по крайней мере настолько безопасным, насколько это возможно.

В идеальном мире наши цели должны были бы заключаться в том, чтобы дать возможность разработчикам распознавать, когда изменения в базе данных требуют анализа и управления со стороны DBRE, чтобы эффективно внедрять их в производственной среде. Кроме того, мы сможем предоставить и разработчикам инструменты для безопасного и надежного внедрения в производственной среде большинства их изменений. Наконец, дадим им возможность вносить изменения в саму производственную среду в любое время, а не только в периоды технического обслуживания. О том, как приблизиться к такому идеальному миру, мы и поговорим в этом разделе.

Миграции и управление версиями

Как обсуждалось в подразделе «Предпосылки» раздела «Интеграция» ранее в главе, каждому набору изменений, применяемому к базе данных, должна быть присвоена числовая версия. Обычно это делается приращением целых чисел, которые сохраняются в базе данных после использования набора изменений. Таким образом, система развертывания может обратиться к базе и узнать текущую версию. Благодаря этому можно легко выполнять изменения при подготовке к записи кода. Если развертывание кода сертифицировано для версии 456 базы данных, а ее текущая версия — 455, то группа развертывания знает, что до записи кода необходимо применить набор изменений для версии 456.

Итак, предположим, что программист сохранил в кодовой базе набор изменений с версией 456 и интеграция была успешно выполнена без каких-либо серьезных изменений. Что дальше?

Анализ последствий

Мы уже анализировали последствия в предыдущем разделе, посвященном тестированию после записи в VCS. Некоторые последствия, такие как признание недействительным кода, хранимого в базе данных, или нарушение мер безопасности, являются теми сигналами, которые нельзя пропустить. Разработчик должен вернуться и модифицировать свои изменения так, чтобы эти воздействия были устранены или хотя бы смягчены.

В этом разделе мы обсудим влияние *процесса* миграции базы данных на рабочие серверы БД. Изменения в базе данных могут по-разному влиять на сервис в прозводственной среде.

Блокировка объектов

Изменения могут привести к тому, что таблица или даже несколько таблиц окажутся недоступными для записи, чтения или того и другого вместе. В этих случаях время, в течение которого объекты недоступны, должно быть оценено и определено как приемлемое или неприемлемое. Приемлемая блокировка действительно является частью целевого уровня качества обслуживания и потребностей бизнеса и, следовательно, субъективна. Предыдущие изменения этих объектов могут быть записаны с соответствующими показателями примерно за то время, которое потребовалось для внесения изменений. Это позволит использовать некоторые объективные данные для определения времени воздействия, несмотря на то что время, необходимое для внесения изменений в объекты, будет увеличиваться по мере расширения размера набора данных и увеличения активности.

Если этот вариант не подходит, то DBRE должен совместно с группой развертывания составить план сокращения времени до приемлемого или перенаправления трафика до тех пор, пока изменение не будет успешно завершено.

Насыщение ресурсов

При изменении также может потребоваться значительное количество операций ввода/вывода, что увеличит задержку для остальных транзакций, использующих хранилище данных. Это способно привести к нарушению целевого уровня качества обслуживания и в итоге к возвращению процессов в состояние, при котором приложение становится непригодным для применения и наступает насыщение остальных ресурсов. Это легко приводит к каскадному сбою.

Проблемы целостности данных

Частью этих изменений зачастую являются переходные периоды, в течение которых ограничения могут быть смягчены или отложены. Аналогично блокировка и признание данных недостоверными могут привести к тому, что данные не будут сохраняться так, как ожидают разработчики.

Задержки репликации

Изменения в базе данных могут вызвать также увеличение активности и задержку репликации. Это может повлиять на полезность реплик и даже подвергнуть риску процедуры восстановления после отказа. Мы, как инженеры по обеспечению надежности БД, должны помогать разработчикам активно выявлять такие последствия и избегать их.

Паттерны миграции

Проанализировав последствия, разработчик должен иметь возможность принять решение о выборе способа развертывания миграции. Во многих случаях при миграции нет причин выполнять множество последовательных изменений и затем проводить масштабную работу по их проверке. Создание новых объектов, вставка данных и другие модификации могут быть легко применены и переданы в промышленную эксплуатацию.

Однако после того, как данные будут внесены в систему, изменены или удалены существующие данные либо изменены или удалены объекты с содержащимися в них данными, возникают возможности для миграции, которая, как уже говорилось, повлияет на целевой уровень качества обслуживания. Именно в это время разработчик должен обратиться к DBR-инженеру. К счастью, набор изменений, которые можно спланировать, ограничен. Разрабатывая с программистом план и выполняя эти миграции, вы можете создать репозиторий шаблонов для внесения изменений в базу. В какой-то момент, если они происходят довольно часто и безболезненно, можете их автоматизировать.

Например, можете настроить шлюзы развертывания при интеграции и тестировании, которые путем анализа на основе правил по результатам тестирования определяют, безопасны ли миграции для развертывания. Вот несколько примеров таких операций с ограничениями:

- обновление и удаление строк без фильтрации с помощью `WHERE`;
- количество задействованных строк больше *N*;
- изменение таблиц с определенным размером набора данных;

- изменение таблиц, хранящихся в метаданных, как слишком нагруженных, чтобы вносить изменения на ходу;
- создание новых столбцов со значениями по умолчанию;
- использование определенных типов данных, таких как файлы BLOB (Binary Large Object — большой двоичный объект), в инструкциях `create/alter`;
- применение внешних ключей без индексов;
- операции в особо важных таблицах.

Чем больше ограничений и мер безопасности вы установите для всех с целью обеспечения безопасности в производственной среде, тем увереннее будут себя чувствовать все отделы. Это повысит скорость разработки. Теперь предположим, что изменение, которое наш бесстрашный программист зафиксировал в VCS под версией 456, было помечено как небезопасное, так как в нем содержится операция `alter`, которая, как считается, способна привести к слишком серьезным последствиям. Для данной операции программист может использовать паттерн миграции, если он был применен и задокументирован. В противном случае разработчик должен создать такой паттерн совместно с отделом DBRE.

Паттерн «Операции блокировки»

Добавление столбца — популярная операция в большинстве систем управления базами данных. В некоторых СУБД такие операции могут выполняться быстро и просто, без блокировки таблицы. Другие СУБД потребуют создать таблицу заново. При добавлении столбца для него можно указать значение по умолчанию. Это определенно серьезно повлияет на работу, поскольку данное значение должно быть введено во все существующие строки таблицы, прежде чем изменение будет завершено, а блокировка снята.

Чтобы избежать некоторых операций блокировки, можно воспользоваться кодом, например:

- добавить пустой столбец;
- выполнить регрессионное тестирование;
- задействовать условный код в операторе `select` во время доступа, чтобы определить, нужно ли обновлять строку вместо того, чтобы выполнять обновление в виде пакетного оператора;
- настроить наблюдатель, который сообщает, когда все поля будут заполнены и условный код можно удалить.

Для некоторых операций блокировка объекта неизбежна. В этих случаях необходимо предоставить разработчикам паттерн, реализуемый автоматически или вруч-

ную. Это может быть инструмент для выполнения изменений по сети с помощью триггеров и путем переименования таблиц. Или же это может быть скользящая миграция с использованием прокси и восстановлений после отказов, при которой изменения применяются по очереди ко всем неработающим узлам.

Может оказаться заманчивым создать два процесса: один легкий, а другой — с большим количеством шагов. Тогда можно будет разворачивать сложный паттерн только для серьезных изменений. Однако это может привести к тому, что вы будете слишком сильно полагаться на один процесс, в результате чего другой будет задействоваться недостаточно часто и, возможно, окажется нарушен. Лучше всего придерживаться того процесса, который наиболее эффективно работает для всех операций блокировки.

Паттерн «Операции с высокой нагрузкой на ресурсы»

В этом случае можно использовать несколько паттернов в зависимости от выполняемых операций.

Регулирование путем выполнения обновлений в пакетном режиме для изменения данных — это простое решение, подходящее для выполнения пакетных операций. В больших средах часто имеет смысл запустить код для отложенного («ленивого») обновления при входе пользователя в систему или, например, запросе строки.

Для удаления данных можно рекомендовать программистам задействовать в коде *мягкое удаление*. В этом случае строка помечается как удаляемая, что означает: она будет отфильтрована из запросов приложения, а при необходимости удалена окончательно. Затем можно регулировать операции удаления, выполняя их асинхронно. Как и в случае пакетного обновления, для больших наборов данных это может оказаться невозможным. Если операции удаления регулярно выполняются для определенных диапазонов данных, таких как даты или группы идентификаторов, можно использовать функции разделения для удаления целых разделов. При удалении таблицы или раздела не создаются отмененные операции ввода-вывода, что может снизить потребление ресурсов.

Если вы обнаружите, что DDL-операции, такие как изменение таблиц, создают слишком много операций ввода-вывода, влияющих на время отклика, это следует считать сигналом тревоги, показывающим, что емкости может быть недостаточно. В идеале следует работать с операциями так, чтобы увеличить емкость хранилищ данных. Но если это невозможно или невозможно прямо сейчас, то такие DDL-операции нужно считать операциями блокировки и применять к ним соответствующие паттерны.

Паттерн «Скользящая миграция»

Как говорилось в предыдущих разделах, зачастую имеет смысл позволять разработчикам постепенно вносить изменения в каждый узел кластера по очереди. Это часто называют *скользящим обновлением*, поскольку изменения применяются к узлам по очереди. Каждый раз это делается немного по-разному в зависимости от того, возможна запись в кластер через данный узел или только один узел может функционировать как узел для записи.

Если возможна запись в любую точку кластера, как в случае Galera, то можно отключить один узел, удалив его из каталога сервиса или конфигурации прокси. После того как трафик уйдет с этого узла, можно применить изменение. Затем узел снова вводят в соответствующие конфигурацию или каталог, чтобы вернуть его в эксплуатацию.

В кластере, где есть ведущий узел записи и все операции записи направляются в один узел, необходимо отключить по очереди каждый ведомый узел в цепочке репликации, как и в случае кластера с возможностью записи в любой точке. Однако после того, как это будет сделано для всех узлов, кроме ведущего, нужно переключиться на узел с уже использованным набором изменений.

Очевидно, что оба эти варианта требуют значительной оркестрации. Понимание того, какие из операций затратные и могут потребовать скользящего обновления, — важный фактор, определяющий выбор хранилища данных. По этой же причине часто изучают решения для БД, которые позволяют развивать схему с минимальными последствиями.

Тестирование миграции

Это может показаться очевидным, однако необходимо напомнить, что если изменены детали реализации набора изменений, то пересмотренная миграция должна быть сохранена в VCS и полностью интегрирована перед развертыванием в постинтеграционные среды, включая производственную среду.

Тестирование отката к предыдущему состоянию

Кроме тестирования миграций и их последствий, DBR-инженеры и их группы поддержки должны учитывать сбои в случае миграции или развертывания и возврат частичных или полных наборов изменений. Сценарии изменения базы данных должны быть зарегистрированы одновременно с миграцией. Для некоторых миграций могут использоваться автоматически сгенерированные настройки по умолчанию, например при создании таблиц, но следует вести учет поступающих

данных. Поэтому мы не рекомендуем выполнять откат, просто удаляя объект. Переименование таблиц позволяет им оставаться доступными в случае, если туда были записаны данные, которые должны быть восстановлены.

Паттерны миграции также упрощают процесс определения откатов. Отсутствие эффективного сценария отката может стать определяющим фактором в процессе интеграции и развертывания. Чтобы проверить, работают ли сценарии, можно использовать следующий паттерн развертывания и тестирования:

- применить изменения;
- провести быстрые интеграционные тесты;
- откатить изменения;
- выполнить быстрые интеграционные тесты;
- применить изменения;
- выполнить быстрые интеграционные тесты;
- провести более длительное и периодическое тестирование.

Подобно тестированию восстановления, тестирование откатов имеет решающее значение, и необходимо включить его во все процессы сборки и развертывания.

Вручную или автоматически?

Еще одно преимущество использования паттернов миграции заключается в том, что в них можно предусмотреть автоматическое утверждение и развертывание вместо ожидания проверки и выполнения развертывания командой DBRE. Это также означает, что некоторые паттерны позволяют автоматически уведомлять инженеров по обеспечению надежности БД о том, что необходимо сделать, помимо автоматизированного процесса.

Нет смысла торопиться с автоматизацией, особенно когда речь идет о важных данных. Рекомендуемые сообществом методы предполагают, что все, что приходится делать часто, должно быть автоматизировано настолько, насколько возможно, но энтузиазм снижается под влиянием прежних неудачных попыток автоматизации. Если вы создали среду, в которой есть проверенные и надежные запасные варианты, предусмотрены быстрые и отработанные процессы восстановления и с которой работают опытные разработчики, то можете начать использовать паттерны миграции и автоматизировать применение этих изменений. Но только переход к стандартизированным моделям, развертываниям и откатам «в один клик», а также защитные ограждения и флажки обеспечивают существенный прогресс в достижении наших целей.

Резюме

В этой главе мы рассмотрели способы, с помощью которых команда DBRE может серьезно облегчить работу программистов на этапах разработки, интеграции, тестирования и развертывания ПО. Мы не устаем подчеркивать, насколько сильно это зависит от сотрудничества и тесных взаимоотношений между командами DBRE, эксплуатации и разработки. Вы, как инженер по обеспечению надежности БД, в этом уравнении должны играть одновременно роль учителя, дипломата, переговорщика и студента. Чем больше сил вы будете вкладывать в обучение коллег и налаживание отношений с ними, тем больше дивидендов получите, когда начнете применять на практике знания, полученные в этой главе.

От управления релизами естественно перейти к безопасности. Данные — одна из главных целей хакерских атак во всей инфраструктуре. Каждое изменение и добавление функционала потенциально создают уязвимости, которые должны быть запланированы и устранены. В главе 9 мы обсудим, что может быть сделано для планирования и процессов в сфере безопасности.

9 Безопасность

Функция безопасности всегда была важной частью работы администратора базы данных. Как и в случае с восстановлением данных, безопасность самого важного актива организации имеет первостепенное значение. Инциденты и попытки нарушения безопасности происходят все чаще и чаще. Достаточно лишь почитать новости, чтобы узнать о громких случаях взлома сотен тысяч (даже миллионов) профилей пользователей, кредитных карт и электронных адресов.

В изолированном мире администратор базы данных (DBA) сосредоточился бы только на своих средствах управления безопасностью БД, усиливая изоляцию и считая, что есть и другие люди, отвечающие за безопасность. Однако, будучи управляющим данными организации, инженер по надежности баз данных (DBRE) должен использовать в работе более комплексный подход.

В предыдущих главах мы уже говорили о процессах непрерывного развертывания (CD), облачных средах и инфраструктуре как коде. В каждой из этих областей у потенциальных воров и вандалов, желающих получить ваши данные, есть свои направления атаки. В этой главе мы обсудим парадигму безопасности баз данных для DBRE, которой должны соответствовать современные организации и инфраструктуры. Затем мы перейдем к практике, рассмотрим потенциальные направления атак, методологию и стратегию смягчения последствий, а также разберемся, какую комплексную модель могут разработать DBR-инженеры.

Цель безопасности

Разумеется, безопасность — это ключевая цель. Она важна не меньше восстановления данных, о котором говорилось в главе 7. Зачастую, если данные были украдены, то это ничем не лучше, чем если бы они были повреждены. Но, подобно тому как восстановление — задача с большим количеством вариантов использования, многие из которых выходят за рамки просто аварийного восстановления данных, так и безопасность имеет множество функций.

Защита данных от кражи

Это классический вариант использования. Как правило, если есть возможность хранения данных, значит, найдется кто-то, кто захочет получить к ним доступ, не предусмотренный вариантами их обычного использования. Частные лица как внутри, так и вне организации, могут пожелать получить доступ к базам данных, чтобы перепродать данные клиентов, узнать секреты фирмы или просто причинить ущерб, используя полученные данные. Вот направления атаки, применяемые в этом случае:

- данные в сетевых базах данных;
- данные, перемещаемые между хранилищами;
- данные в резервных копиях и архивах;
- данные, поступающие из хранилищ в приложения и на клиентские устройства;
- данные в памяти на серверах приложений;
- данные, передаваемые пользователям из приложений через Интернет.

Как мы только что отметили, не все воры приходят извне. Внутренние пользователи, которые уже, возможно, имеют представление о системах и аутентифицированном доступе, даже более опасны, чем многие воображаемые призраки из Интернета. Независимо от местоположения, DBRE работает с InfoSec, Ops и разработчиками ПО (SWE), чтобы убедиться, что данные можно прочитать, продублировать или переместить из соответствующего места.

Защита от целенаправленного повреждения

Иногда намерение злоумышленника состоит только в том, чтобы нанести вред организации. Повреждение данных или манипулирование ими, закрытие баз данных или нагрузка всех ИТ-ресурсов так, чтобы они стали недоступными, — все это может привести к повреждению БД и данных организации. Это может принимать форму атак типа «отказ в обслуживании» (Denial of Service, DoS), эксплуатации ошибок, при которых базы данных отключаются, и видов доступа, позволяющего манипулировать данными или хранилищем. Хорошая новость о таких атаках заключается в том, что данные после них часто можно восстановить из резервных копий. Однако резервные копии можно очень легко повредить.

Защита от случайного повреждения

Хотя мы часто считаем, что безопасность — это функция, которая защищает нас от опасных людей, она также важна для обеспечения того, чтобы кто-нибудь случайно не попал в неверную среду, схему, объект или строку и не причинил непреднамеренный ущерб. Забор не только ограничивает проход посторонних: люди понимают,

что они заблудились и попали не туда, куда хотели. С точки зрения защиты данных от кражи есть не только внутренние субъекты, которые более опасны, чем внешние, но и случайные вандалы и диверсанты, которые часто приходят с инструментами и учетными данными, способными быстро привести к катастрофе.

Защита данных от раскрытия

Даже без умышленного или неумышленного воздействия со стороны людей все равно остаются риски доступа к данным. В сложных, распределенных и разделенных системах довольно легко совершить ошибку или неправильно разместить множество учетных данных, в результате чего конфиденциальные данные окажутся доступными в виде открытого текста в журналах, в браузере клиента или в электронной почте. Либо даже может появиться возможность входа в систему для неавторизованных пользователей под учетной записью другого человека. Такое раскрытие данных может быстро — и совершенно резонно — подорвать доверие к способности организации защищать и обрабатывать данные.

Стандарты соответствия и аудита

Организациям необходимо тщательно соответствовать многочисленным стандартам и выполнять законы, которые помогают защитить клиентов и отдельных лиц. Задача отдела безопасности — информировать организации об этих стандартах и обеспечивать соответствие им. Это неблагодарная работа, которая часто удручает тех, что стремится сосредоточиться на новых функциях и масштабировании. Тем не менее она важна, если вы не хотите, чтобы организацию закрыли или оштрафовали на крупную сумму.

Безопасность базы данных как функция

На протяжении всей книги мы твердо придерживаемся понятия межфункциональных взаимодействий и подходов к надежности базы данных. Сегодня DBRE-специалист в значительной степени становится связующим звеном, экспертом в предметной области и учителем для остальных сотрудников организации. Поскольку команды разработчиков расширяются в геометрической прогрессии, это единственный способ масштабирования. Эксперты по информационной безопасности часто являются одной из самых недоукомплектованных должностей в организации, что еще больше усложняет задачу для команд DBRE и информационной безопасности (IS) по эффективной защите данных компании в условиях постоянного развития и изменений.

Именно это мы обсуждали в главе 8, где сосредоточились на обеспечении безопасного, эффективного и быстрого развития с помощью методов самообслуживания,

обучения и избирательного подхода. Разработчики также находятся на переднем крае обеспечения безопасности, и те подходы, которые мы обсуждаем здесь, будут реализованы аналогичным образом. Безопасность должна быть в первую очередь интегрирована в процессы разработки приложений и инфраструктур, а не добавлена в виде примечания или флажка на листе соответствия во время релиза.

Как это сделать? С помощью тех же инструментов (http://bit.ly/2zw896p), которые мы обсуждали на протяжении всей книги. Давайте рассмотрим их еще раз.

Обучение и сотрудничество

Мы подробно рассмотрели это в главе 8, поэтому здесь кратко перечислим три подхода:

- поощрение общения;
- создание предметно-ориентированных баз знаний;
- сотрудничество путем совместной работы и сбора отзывов.

Это делается для того, чтобы научить разработчиков более эффективно и безопасно создавать средства защиты от атак на хранилища данных организации. Таким образом повышается производительность и эффективность приложения, сокращается время простоя и недостаточного обслуживания из-за неудачных реализаций и некачественного проектирования, а также увеличивается скорость работы разработчиков. Аналогичные коллективные усилия должны быть постоянно направлены на обучение обеспечению безопасности БД. Сюда входит следующее:

- безопасный доступ к конфигурации и элементам управления базы данных;
- эффективное использование функций безопасности, таких как шифрование, настраиваемый контроль доступа и управление данными;
- понимание, какие данные предоставляются базой и могут быть перенесены в инструментальные средства, журналы и телеметрию, чтобы помочь выявить злонамеренную или вредоносную активность;
- изучение особых уязвимостей базы данных, с которыми необходимо работать на других уровнях, включая обновления по мере выпуска новых *CVE*.

CVE

Аббревиатура CVE расшифровывается как Common Vulnerabilities and Exposures — общие уязвимости и воздействия (https://cve.mitre.org/). Вы также можете создавать собственные рассылки для отслеживания интересующих тем, таких как SQL-инъекции. Это отличный ресурс, позволяющий всегда быть в курсе недавно обнаруженных или обновленных уязвимостей.

Благодаря постоянному обучению и сотрудничеству специалистов безопасность баз данных становится регулярно обсуждаемой темой, которая исследуется, анализируется и изучается в организации.

Самообслуживание

Грамотная организация процесса обеспечения безопасности, способного масштабироваться в зависимости от размера и скорости работы программистов, требует подходов к обеспечению безопасности БД, основанных на самообслуживании. Вы никогда не сможете самостоятельно просмотреть каждую функцию, каждый новый сервис и каждое новое хранилище данных. Вместо этого вы по мере увеличения отставания будете постоянно блокировать запросы. Партнерство с InfoSec для создания многократно используемых утвержденных паттернов безопасности, которые инженеры могут проверять и применять по своему усмотрению, обеспечивает масштабируемый процесс безопасности.

Как уже обсуждалось в главе 5, инфраструктура как код позволяет создавать утвержденные развертывания всех хранилищ данных, которые могут быть созданы и запущены в работу. Это означает, что большую часть времени вы будете создавать эти эталонные экземпляры («золотые стандарты»), исследовать уязвимости, а также пересматривать и обновлять руководства для своей платформы с целью смягчения этих уязвимостей, в том числе делать следующее:

- утверждать номера сборок программного обеспечения;
- удалять стандартные учетные записи и пароли, которые поставляются с хранилищами данных;
- блокировать ненужные порты;
- составлять эффективно ограниченные списки доступа для сокращения точек входа в хранилища данных;
- удалять функции и конфигурации, позволяющие использовать эксплойты через файловую систему или сеть;
- создавать и настраивать ключи для связи по протоколу SSL;
- запускать сценарии проверки и применения политик паролей;
- настраивать аудит и пересылать журналы, позволяющие гарантировать, что весь доступ может быть просмотрен и защищен от взлома.

Проверив все это и сделав доступным для тех, кто занимается развертыванием новых хранилищ данных, вы можете рассматривать эти инфраструктуры как предварительно подтвержденные, поскольку используются эталонные экземпляры. Имея дело с этими уже знакомыми инсталляциями, командам DBRE и InfoSec уже не придется тратить время на их анализ и составление отчетов об уязвимостях.

В то же время можно зарегистрировать и сделать доступными библиотеки кодов, соответствующие ведению журнала, аутентификации, хешированию паролей и шифрованию. Сюда также входит клиентское программное обеспечение.

Настройка клиентских приложений, использующих базы данных

Подобно самообслуживанию для инфраструктуры, предоставление самообслуживания для клиентских приложений и библиотек станет эффективным методом смягчения последствий. Приложения, предоставляемые поставщиками, часто сразу имеют обходные пути, о которых вы, возможно, не подозреваете, тем самым невольно создавая потенциальные возможности для уязвимостей. Эти клиентские приложения также часто используют старые протоколы для обеспечения обратной совместимости. Написав свое собственное клиентское приложение, вы уменьшите риск неизвестных факторов и получите контроль над основной частью уровня базы данных: уровнем доступа.

Интеграция и тестирование

Интеграция и тестирование предоставляют отличные возможности для раннего выявления уязвимостей. Эти действия можно выполнять часто, а не в конце процесса разработки, когда стоимость исправления экспоненциально возрастает. Однако есть риск, что злоумышленник, которому принадлежит сервер тестирования и интеграции, легко обойдет все тесты и внедрит вредоносный код.

Во время интеграции можно применять стандартные тесты, утвержденные системой безопасности, чтобы автоматически проверить, не были ли внедрены уязвимости. Сюда входят следующие тесты (но не только они):

- на уязвимость SQL-инъекций в функциях доступа к базе данных;
- тестирование уровня аутентификации на наличие распространенных дефектов, включая коммуникацию открытым текстом, хранение учетных данных в виде простого текста или подключение с расширенными правами администратора;
- тестирование нового сохраняемого кода на наличие эксплойтов, таких как переполнение буфера.

Кроме тестов, выполняемых сразу после записи в VCS, можно проводить более интенсивные тесты — асинхронно, по регулярному расписанию. Сюда входят тесты на проникновение на уровне приложений, а также тщательное тестирование через сеть на наличие уязвимостей, которые могут быть использованы с аутентификацией и без нее в попытке получить доступ к базе данных или операционной системе.

Оперативный контроль

Сбор и интеграция всех входных и выходных данных подсистемы безопасности, результатов ее работы с помощью стандартных средств телеметрии и журналирования имеют решающее значение. Эти данные поступают из разных мест стека, включая уровни приложений, базы данных и ОС. Это подробно описывалось в главе 4.

Измерения на уровне приложений

Отслеживание всех неудачных и успешных выполнений операторов SQL, переданных в базу данных, имеет решающее значение для выявления атак SQL-инъекций. Здесь основным показателем могут быть синтаксические ошибки в SQL. Они предупреждают вас о том, что кто-то или что-то пытается передать незапланированный SQL-код из приложения в базу данных. Синтаксические ошибки в работающем, протестированном приложении должны быть очень редкими. Точно так же шаблоны SQL-инъекций, если они изучены, способны указать на строки, по которым часто бывает заметно, что атака идет полным ходом. Сюда входят операторы `UNION` и `LOAD_FILE`. Подробнее мы обсудим это в следующем разделе.

Данные контроля также должны собираться для идентификационной информации (Personally Identifiable Information, PII) или критически важных данных. Использование метаданных для пометки конечных точек API как PII или критически важных данных позволяет собирать подробные сведения о доступе, изменении или удалении данных, таких как пароли, электронные письма, номера кредитных карт или blob-документы. Несмотря на то что контроль также будет выполняться на уровне базы данных, контроль на уровне приложений позволит соответствующим сотрудникам проверить, случайно или преднамеренно используется код приложения.

Измерения на уровне базы данных

На уровне базы данных любая активность должна журналироваться и помещаться в стек оперативного контроля для анализа. Ниже приведены виды активности, которые стоит учитывать.

- *Изменения конфигурации*. Это может происходить в файле или в памяти. Изменения в файле конфигурации могут полностью открыть систему для любого использования.
- *Изменения пользователей базы данных*. Изменение привилегий или паролей, а также создание новых пользователей должно проверяться на предмет того, были ли соответствующие миграции внесены в систему контроля версий

и интегрированы. Если нет, то эти изменения могли быть внесены намеренно, чтобы создать дыры в безопасности.

- *Выбор, вставка, обновление и удаление всех данных.* Контроль на уровне базы данных хорошо дополняет контроль на уровне приложений, так что иногда второй может не потребоваться. Чрезмерные запросы или изменения, запросы от неожиданных пользователей и неожиданно большие наборы результатов могут указывать на наличие проблем.
- *Создание новых объектов базы данных, особенно хранимого кода.* Новые или измененные функции, процедуры, триггеры, представления и пользовательские функции (User-Defined Functions, UDF) должны соответствовать миграциям базы данных, поскольку могут быть признаками эксплойтов.
- *Удачный и неудачный вход в систему.* Для любой базы данных должны быть характерны определенные модели трафика. Пользователи приложения будут приходить с конкретных групп хостов, и, вообще говоря, никто не должен входить в базу данных напрямую. В некоторых средах можно пойти дальше и пометить хранилище данных как подозрительное, если в него был осуществлен вход не с сервера приложений, прокси или другого доверенного клиента.
- *Патчи и бинарные изменения.* Горячие исправления могут исходить от пользователей, получивших доступ к ОС. Это может случиться в результате переполнения сетевого буфера или других эксплойтов. Такие изменения могут приводить к созданию лазеек и потенциально вредоносного кода.

Средства управления операционной системой

Как и в случае с базой данных, операционные системы тоже должны тщательно контролироваться, а события в них — журналироваться. Контролю подлежат следующие моменты.

- *Изменения конфигурации.* То же, что и в случае с изменениями базы данных.
- *Новое программное обеспечение, сценарии и файлы.* Появление нового или измененного программного обеспечения, сценариев или файлов за пределами временных каталогов, каталогов журналов или других ожидаемых мест размещения новых файлов почти всегда оказывается плохим признаком. Регулярное сравнение с эталоном помогает выявить вредоносную активность.
- *Удачные и неудачные входы в систему.* То же, что и при входе в базу данных.
- *Патчи и бинарные изменения.* То же, что с изменениями базы данных.

Всесторонний сбор данных в сочетании с использованием эффективных инструментов для сравнения и обнаружения аномалий имеет решающее значение для выявления вредоносных действий. Не существует стратегии безопасности, которая

была бы всеобъемлющей и достаточно актуальной, чтобы не пропускать никого и ничего, поэтому необходим эффективный контроль, и команда DBRE работает рука об руку с Ops, InfoSec и программистами, чтобы обеспечить этот контроль соответствующим инструментарием.

Уязвимости и эксплойты

Мы уже говорили, не вдаваясь в подробности, об основных рабочих обязанностях DBRE-специалиста, выполняя которые он может настроить масштабируемую функцию безопасности. В предыдущем разделе мы в очень общих чертах обсудили различные потенциальные угрозы. В этом разделе обсудим потенциальные уязвимости, которые необходимо учитывать и планировать, поскольку DBR-инженер занимается обучением и повышением квалификации в организации, созданием конфигураций самообслуживания и настройкой мониторинга, паттернов и инструкций для реагирования на угрозы безопасности.

При моделировании угроз важно классифицировать их и расставлять приоритеты. Балансировка и приоритизация с учетом всех остальных приоритетов имеют решающее значение. Для этого уже предусмотрены структурированные подходы. Например, Microsoft поставляет продукты STRIDE для классификации уязвимостей и угроз и DREAD — для приоритизации.

STRIDE

STRIDE — это классификационная схема для описания известных угроз в соответствии с используемыми типами эксплойтов (или мотивами злоумышленника). Аббревиатура STRIDE (http://bit.ly/2zxfqCJ) образована первыми буквами следующих категорий.

- *Spoofing identity — подмена идентификатора*. Подмена идентификатора позволяет пользователю указать другой идентификатор, чтобы обойти контроль доступа. Поскольку в большинстве многопользовательских приложений только один пользователь попадает в базу данных, это довольно рискованно.
- *Tampering with data — фальсификация данных*. Пользователи могут изменять данные с помощью POST-директив приложений, а также действий, предпринятых с поддельными или ложными идентификаторами. Валидация данных и использование API даже для действий администратора имеет решающее значение для защиты от этой угрозы.
- *Repudiation — оспариваемость*. Без надлежащего уровня контроля клиенты и внутренние пользователи могут оспаривать предпринятые ими действия.

Это может привести к финансовым потерям в спорах, неудаче при проверках и невозможности обнаружить действия злоумышленников.

- *Information disclosure — утечка информации*. Сведения о клиентах и частные данные могут оказаться доступны широкой общественности, конкурентам и злоумышленникам. Такая утечка информации также может быть случайной.
- *Denial of service — отказ в обслуживании*. Приложения и отдельные компоненты инфраструктуры также могут получить отказ в обслуживании. Это может происходить из-за затратных операций или просто вследствие огромного количества действий из разных точек мира.
- *Elevation of privilege — повышение привилегий*. Пользователи могут менять свою роль на роль с более высоким уровнем привилегий. На самом высоком уровне пользователи приложений могут получить доступ к серверу с полномочиями суперпользователя.

DREAD

Классификация DREAD позволяет анализировать риски и определять приоритеты на основе риска, присвоенного каждой из рассмотренных угроз. Представленный здесь алгоритм DREAD (http://bit.ly/2zjZfrL) используется для вычисления значения риска, которое является средним по всем пяти категориям.

- *Степень уязвимости объекта*. Какой ущерб будет причинен в случае существования угрозы?
  - 0 — никакой.
  - 5 — будут скомпрометированы или затронуты данные отдельных пользователей.
  - 10 — полное уничтожение системы или данных.
- *Воспроизводимость*. Насколько легко воспроизвести угрозу?
  - 0 — очень сложно или невозможно, даже для администраторов приложения.
  - 5 — требуется один или два шага, может потребоваться авторизация пользователя.
  - 10 — достаточно браузера и адресной строки, без аутентификации.
- *Возможность эксплойтов*. Что нужно для того, чтобы случилась эта угроза?
  - 0 — расширенные знания в области программирования и сетей со специальными расширенными инструментами атаки.
  - 5 — в Интернете распространяется вредоносное ПО или эксплойт легко выполняется с использованием доступных инструментов атаки.
  - 10 — только браузер.

- *Затронутые пользователи*. Сколько пользователей будет затронуто?
  - 0 — нисколько.
  - 5 — некоторые, но не все пользователи.
  - 10 — все пользователи.
- *Легкость обнаружения*. Насколько легко обнаружить угрозу?
  - 0 — очень трудно или невозможно; требует доступа к исходному коду или административного доступа.
  - 5 — можно выявить, догадавшись или отслеживая следы по сети.
  - 9 — подробная информация о таких сбоях уже есть в свободном доступе и ее легко найти с помощью поисковой системы.
  - 10 — информация отображается в адресной строке браузера или в форме.

Сейчас, когда мы разбираемся с каждым из возможных направлений атак, наличие подобной категоризации может помочь определить, на что направить силы и ресурсы.

Основные меры предосторожности

В этом разделе мы обсудим несколько возможных мер предосторожности с примерами из разных хранилищ данных. Сюда входят общие методы смягчения ущерба, которые будут подробнее рассмотрены в разделе, посвященном стратегии. К методам смягчения ущерба, которые являются достаточно общими и могут применяться к нескольким категориям угроз, относятся следующие.

- *Настройка конфигурации*. Удалите из базы данных все ненужные функции и параметры конфигурации. Многие системы управления базами данных имеют множество функций, большинство приложений не будут использовать даже малую их часть. Отключение лишних функций поможет уменьшить число направлений атаки.
- *Установка исправлений (патчей)*. Постоянно проверяя базы данных на наличие уязвимостей, регулярно устанавливайте патчи безопасности для их устранения. Устанавливая все последние версии патчей, вы снизите риск эксплойта.
- *Удаление ненужных пользователей*. Стандартные пользователи и пароли хорошо известны и создают значительную угрозу безопасности.
- *Доступ к сети и хосту*. Используйте брандмауэры и группы безопасности, чтобы свести к минимуму группы хостов, имеющие доступ к базам данных, и порты, через которые этот доступ осуществляется. Аналогично первостепенное значение имеет применение ограничений с помощью ролей и привилегий, чтобы свести к минимуму вероятность доступа к системам для произвольного пользователя.

Опасность значений по умолчанию

Когда мы писали эту книгу, существовал крайне показательный — и легко предотвратимый — эксплойт безопасности для баз данных MongoDB и ElasticSearch, прослушивающий общедоступные IP-адреса. В 2015 году Шодан (Shodan) написал статью (https://blog.shodan.io/its-the-data-stupid/) о том, что более 30 000 экземпляров MongoDB оказались общедоступными, поскольку их IP-адрес по умолчанию был 0.0.0.0, а аутентификация не была включена. Это более 595,2 Тбайт данных, попавших в открытый доступ только потому, что никто не обращал внимания на настройки по умолчанию в ранних версиях сервера.

Мы обсудим следующие категории:

- отказ в обслуживании;
- SQL-инъекции;
- сетевые протоколы и протоколы аутентификации.

Отказ в обслуживании

Атаки типа «отказ в обслуживании» (Denial of Service, DoS) — это семейство атак, предназначенных для того, чтобы сделать сервис или приложение недоступными вследствие большого количества запросов. При такой атаке происходит насыщение и перегрузка ресурсов, из-за чего становится невозможно выполнять запросы от реальных пользователей. Это часто выглядит как исчерпание пропускной способности сети из-за того, что некая распределенная сеть клиентов переполнила сеть своими запросами. К другой категории относятся атаки, которые заключаются в исчерпании ресурсов конкретного сервера или кластера, например базы данных. Из-за того что все ресурсы процессора, памяти или диска заняты, критически важный сервер может перестать отвечать, гарантированно отключая все зависящие от него сервисы.

Такие атаки, как правило, не являются разрушительными, поскольку при них не происходит повреждение или кража данных. Они предназначены для того, чтобы отключить сервис то ли из-за общих неудобств, в качестве борьбы с конкурентом, то ли как прикрытие для отвлечения команд InfoDec и Ops на время других атак.

Лавинные атаки в больших сетях — привычное дело, поэтому на них фокусируются большинство методов защиты. Это привело к тому, что злоумышленники пошли выше и обратились к сервисным компонентам, которые более уязвимы, имеют меньше доступных ресурсов и играют роль центров притяжения. К сожалению, базы данных являются идеальной мишенью. Так появились атаки типа DB-DoS —

отказ в обслуживании со стороны базы данных. При минимальных усилиях нагрузка на логику базы данных может расти экспоненциально, перегружая ресурсы способами, которые не сильно отличаются от обычного повышения трафика.

Далее приведены возможные последствия атаки DB-DoS, которые мы хотели бы смягчить:

- перегрузка пользовательских соединений вплоть до исчерпания доступных серверов приложений;
- нарушение работы оптимизатора вследствие огромного разброса запросов, требующих анализа, хеширования и проверки во время оптимизации запросов;
- автоматическое масштабирование ресурсов до тех пор, пока сервис не закроется вследствие исчерпания бюджета;
- удаление из кэшей актуальных данных, что приводит к интенсивному обмену данными с диском;
- усиленное использование памяти, потенциально способное вызвать подкачку;
- рост размера таблиц и журналов, которые могут занять все дисковое пространство;
- чрезмерное замедление репликации вследствие большого количества записей;
- недостаток ресурсов ОС, включая файловые дескрипторы, процессы или разделяемую память.

Такие атаки могут быть легко спровоцированы с помощью разных тактик. Самая простая из них — использовать функциональность самого приложения. Вот несколько примеров:

- создание большого количества заказов в корзине;
- поиск без ввода критериев или со слишком широкими критериями поиска;
- медленные вызовы API могут сообщить злоумышленнику, что данный запрос не может быть оптимизирован или проиндексирован и, следовательно, может использоваться для повторяющихся вызовов;
- добавление UNION во входные данные в формах, что может привести к огромному количеству объединений и сканирований (эта методика также используется для SQL-инъекций);
- сортировка большого количества результатов запроса;
- создание граничных случаев, таких как огромное количество постов на форуме или огромное количество друзей в социальной сети.

Подобно злоупотреблению функциональностью приложения, опытный злоумышленник, который умеет идентифицировать используемую базу данных, часто может найти способы прекратить ее работу. Например, блокировка пользователей путем неправильного входа в систему, выполнение административных команд с помощью SQL-инъекции, которая может очистить кэш, или отправка искаженного XML-файла могут привести к переполнению в синтаксическом анализаторе.

Смягчение рисков

В дополнение к стандартным методам смягчения рисков, которые мы уже обсуждали, есть эффективные подходы к смягчению рисков при атаках типа DB-DoS — они очень похожи на методы, что используются в условиях интенсивного трафика и при проблемах роста. Они служат подспорьем, которое позволяет пережить внезапные скачки нагрузки ресурсов, создаваемые как ожидаемыми «правильными», так и неизвестными источниками трафика. Обратите внимание, что сюда не входит автоматическое масштабирование емкости хранилищ. В зависимости от возможностей оборудования, программного обеспечения или бюджета всегда есть какая-то верхняя граница установленной емкости, и любая атака DB-DoS, вероятно, может подловить вас на этом.

Управление ресурсами и сброс нагрузки

Со временем инженеры начнут понимать, какой может быть общая рабочая нагрузка в их приложениях. Тогда можно будет рассчитывать на то, что они будут в состоянии сформировать набор инструментов, позволяющий эффективно справляться со скачками нагрузки. Ваша обязанность, как DBRE-специалиста, заключается в том, чтобы обучить разработчиков, помочь им проанализировать эти рабочие нагрузки и расставить приоритеты в работе для эффективного снижения рисков. В число возможных средств могут входить следующие.

- *Ограничения на стороне клиента*. Вместо того чтобы позволить роботу постоянно обращаться к одной и той же конечной точке, установите ограничитель на промежуток времени, который должен пройти до повторной отправки запроса. Такая мера позволит замедлить или даже полностью остановить скачки нагрузки. Это можно сделать добавлением простейших счетчиков, с помощью экспоненциального снижения возможностей, введения коэффициентов для повторных вызовов, а также коэффициентов с учетом данных, возвращаемых из приложения, в зависимости от превышения квот на уровень качества обслуживания.
- *Качество обслуживания*. Вы также можете классифицировать трафик, поступающий в приложение, по степени его важности. Помечая высокозатратные запросы, например поисковые, как менее важные, вы сможете установить в при-

ложении квоты на уровень качества обслуживания, что затруднит выполнение атак DB-DoS.

- *Ухудшение результатов.* Для высокозатратных запросов и удаленных вызовов процедур (Remote Procedure Calls, RPC) можно разработать два пути выполнения. В случае обычной нагрузки вполне подойдет полный вариант выполнения. Однако, возможно, вы захотите уменьшить количество просмотренных строк или сегментов, запрашиваемых во время больших нагрузок — например, таких, которые могут быть вызваны атакой типа DB-Dos.
- *Аннулирование запросов и другие жесткие подходы.* Если нет возможности использовать более комплексные подходы на базе кода, то вам, возможно, придется грубо вмешаться в работу приложения. Достаточно эффективны аннулирование долго выполняющихся запросов или размещение профилей рабочих характеристик на уровне базы данных для сокращения количества ресурсов, которые может потребовать запрос. Взамен, возможно, придется пожертвовать контролем.

Постоянная оптимизация доступа к базам данных и их рабочих нагрузок

Если вам когда-либо требовалось оправдание, чтобы направить все силы DBRE и программистов на очистку базы данных от самых затратных запросов на уровнях базы данных, то возможные атаки DB-DoS — именно то, что вам нужно. Поскольку эти запросы часто не относятся к обычным типам рабочей нагрузки, их можно игнорировать при составлении методологии настройки, которая основана на совокупном потреблении ресурсов на уровне базы данных. Если эти выбросы случаются нечасто, то их легко игнорировать. Но тот, кто ищет, — тот их найдет и придумает, как использовать. Это означает, что в идеале в процессе с высокой производительностью следует искать самые затратные запросы, независимо от частоты выполнения, и поместить их в очередь на настройку.

Мониторинг и журналирование

Несмотря на все эти усилия, настойчивый злоумышленник все равно может повредить вашу базу данных. Эффективный мониторинг выполнений вызовов по их конечным точкам позволит выявлять заметные всплески и сообщать о них команде, занимающейся расстановкой приоритетов, чтобы при необходимости эти вызовы ограничили или даже отменили. Аналогично при наличии запросов или действий, не имеющих ограничений, мониторинг количества элементов — возвращаемых, в памяти, в постоянных или временных таблицах или аналогичных структурах — помогает выявлять потенциальные проблемы.

Важно помнить, что атаки могут принимать различные формы, их цель не всегда воровство данных. При планировании функции обеспечения безопасности об атаках DB-DoS можно легко забыть. Далее мы рассмотрим еще одну угрозу безопасности.

SQL-инъекция

SQL-инъекции — это тип эксплойтов, при которых ко входным данным приложения добавляется код управления базой данных, обычно на SQL. Это делается для того, чтобы обойти защиту и выполнить в базе код, который не имеет ничего общего с ожидаемыми входными данными приложения. SQL-инъекции обычно приводят к ошибкам приложения и вызывают переполнение буфера. Переполнение буфера может привести к отключению базы данных, запуску DB-DoS-атаки или предоставлению пользователю более высокого уровня привилегий доступа к базе данных, в том числе даже на уровне ОС.

Еще одно направление атаки с помощью SQL-инъекции — запись кода в саму базу данных. Например, код в виде хранимых процедур часто выполняет некоторые команды с повышенными привилегиями. Такую SQL-инъекцию может выполнить внутренний пользователь или еще кто-то, кому удалось получить доступ к базе данных, угадав учетные данные или прослушав сеть.

SQL-инъекции также можно использовать с целью доступа к данным через операторы `UNION` для получения наборов данных из других таблиц с таким же количеством столбцов, как и в таблицах, запрашиваемых в исходной форме. Например, если в форме поиска запрашивается таблица с пятью столбцами, то путем присоединения можно добавить наборы результатов в любую другую таблицу с пятью столбцами. Такие данные можно легко украсть, и никакие эксплойты для этого не понадобятся.

Снижение рисков

Основная мера по снижению рисков SQL-инъекций на уровне приложений — обучение команды разработчиков ПО. При написании кода инженерам-программистам следует избегать динамических запросов и постараться исключить возможность ввода вредоносного SQL-кода во входные данные.

Подготовленные операторы

Прежде всего вам необходимо гарантировать, что инженеры будут использовать какой-либо из вариантов подготовленных операторов. Такие операторы еще называют *параметризованными*. В подготовленном операторе структура запроса

определена заранее. Затем данные, вводимые в форму, привязываются к переменной, которая используется для выполнения запроса. Противоположностью этому подходу является динамическое определение и сборка SQL-кода во время выполнения приложения. Использовать подготовленные операторы безопаснее, поскольку в этом случае злоумышленник не сможет изменить логику запроса. При попытке SQL-инъекции SQL-код будет рассматриваться как простая строка, предназначенная для сравнения, сортировки или фильтрации, а не как отдельный оператор SQL, подлежащий выполнению (пример 9.1).

Пример 9.1. Пример подготовленного оператора на Java

```
String hostname = request.getParameter("hostName");

String query = "SELECT ip, os FROM servers WHERE host_name = ? ";

PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, hostName);
ResultSet results = pstmt.executeQuery( );
```

Контроль ввода

Бывают случаи, когда подготовленные операторы не защищают. Динамические имена таблиц или входные данные, определяющие порядок при сортировке, не могут быть подготовлены заранее. Необходима валидация входных данных, от которых требуется защита. В таком случае приложение проверяет определенный список допустимых имен таблиц, а также наличие ключевых слов `Desc` и `Asc`, чтобы убедиться, что выполнение запроса безопасно. Валидация также позволяет гарантировать, что почтовые индексы состоят из пяти целых чисел, строки не имеют пробелов, а длина вводимых данных соответствует заданной или находится в пределах определенного диапазона.

Уменьшение ущерба

Существуют и другие меры снижения риска, которые можно предпринять за пределами приложения и которые позволяют уменьшить возможность внедрения SQL-кода в тех случаях, когда никакие меры предосторожности не позволяют защитить входные данные приложения. Но вы не можете гарантировать, что SQL-инъекция никогда не произойдет, поэтому лучше установить глубокую защиту. Она включает в себя исправление двоичных файлов базы данных для уменьшения количества уязвимостей. Кроме того, крайне важно изолировать код (и не предоставлять права доступа) от тех пользователей базы данных, которым этот код не нужен. Создав для каждого приложения собственного пользователя базы данных, вы уменьшите ущерб, который может причинить взломанный аккаунт.

Мониторинг

Как и в случае почти любой другой неисправности или нарушения нормального функционирования, телеметрия и мониторинг данных имеют решающее значение для смягчения последствий. Важно также обеспечить запись и анализ дампов и трассировки стека, отслеживание в журнале запросов команд, включающих в себя объединения (unions), точки с запятой и другие строки, которые могут указывать на SQL-инъекцию.

SQL-инъекцию очень легко предотвратить, и все же это один из самых популярных способов взлома баз данных. В больших компаниях по разработке программного обеспечения очень важны последовательное, постоянное обучение и сотрудничество, а также совместное использование библиотек, одобренных службой информационной безопасности и командой DBRE. Все это позволит не только обеспечить безопасность, но и повысить скорость работы.

Сетевые протоколы и протоколы аутентификации

Есть несколько способов атаковать сервер базы данных с помощью различных протоколов связи, доступных злоумышленнику. Если в сетевом протоколе есть ошибка, то злоумышленник может воспользоваться ею и получить прямой доступ к серверу. Именно так случилось с ошибкой *hello* (CAN-2002-1123), которая появилась в коде настройки сеанса на TCP-порте 1433. Встречаются также эксплойты, которые выполняются после аутентификации по сети, получения доступа к ОС, базе данных или расширения привилегий. Аналогичным образом протоколы базы данных часто изобилуют уязвимостями. Некоторые серверы, в сущности, допускают установку незашифрованных соединений, что позволяет похищать и использовать логины и пароли. В других случаях ошибки могут позволить пользователю отправлять данные аутентификации без передачи актуальных сведений об учетной записи.

Для уменьшения количества направлений атак в инфраструктуре базы данных стоит использовать описанные в начале этого раздела методы смягчения ущерба. Помимо этого, важно хорошо разбираться в протоколах аутентификации и функциях БД организации по разработке ПО — это позволит обеспечить максимальную безопасность конфигураций.

Теперь, когда мы кратко описали потенциальные направления атак и соответствующих стратегий смягчения ущерба, обсудим защиту данных при вторжении, расширении привилегий или даже полном доступе к базам данных и серверам. Естественным вариантом спасения данных в этом случае является шифрование.

Шифрование данных

Рано или поздно случится так, что какой-нибудь злоумышленник или обычный пользователь, неосознанно нарушивший правила, сможет получить доступ к данным, которого у него не должно быть. Даже если вы уже заблокировали все известные вам пути доступа к сети и ОС, свели к минимуму привилегии для всех пользователей, исправили все известные ошибки и закрыли все известные бреши в приложениях, которые можно было бы использовать, вам все равно нужно быть готовыми к неизбежному.

Шифрование — это процесс преобразования данных с использованием некоего набора ключей или секретных данных. Теоретически только те, у кого есть эти ключи, могут преобразовать зашифрованные данные обратно в пригодный для использования формат. Шифрование часто становится последним способом защиты данных, который работает даже в том случае, если данные украдены и находятся в руках злоумышленников.

В этом разделе мы обсудим шифрование на трех разных уровнях:

- при передаче данных;
- когда данные хранятся в клиенте базы данных;
- когда данные хранятся в файловой системе.

Каждое из этих состояний является потенциальным направлением атаки. Однако использование разных методов шифрования приводит к дополнительным издержкам и расходам, что в большинстве организаций требует дополнительного обсуждения и достижения компромиссов.

На каждом из этих уровней необходимо учитывать еще одно измерение — тип данных. Данные могут быть конфиденциальными или нет; по степени конфиденциальности данных они могут быть разбиты на категории (мы рассмотрим их в последующих подразделах). Если вы храните какие-либо данные такого рода и управляете ими, то важно, чтобы ваша команда DBRE очень тесно взаимодействовала с отделами информационной безопасности и разработки. Необходимо гарантировать, что все заинтересованные стороны понимают принятые обязательства и соблюдают установленные правила. Данные поступают в различных формах, и здесь мы рассмотрим некоторые из наиболее распространенных форм данных, которыми можно управлять.

Финансовые данные

К ним относятся номера банковских счетов и привязанные к ним данные, которые могут использоваться для аутентификации или идентификации. Сюда также входят истории транзакций и сведения о финансовом состоянии физического лица

или организации, такие как кредитный рейтинг, финансовые отчеты, балансы и т. д. В одних только Соединенных Штатах финансовые данные регулируются многочисленными законами, органами и стандартами, включая требования PCI DSS для данных кредитных карт, GLBA, SOX/J-SOX, NCUA, законы о конфиденциальности данных и их хранении, а также закон о борьбе с терроризмом. Подобные законы приняты и в других странах.

Личные данные о здоровье

В эту категорию входит информация о пациентах и их здоровье. Она включает в себя личные данные: номера карт социального страхования, имена и контактную информацию, а также сведения о состоянии здоровья пациентов, ходе их лечения и назначенных процедурах, информацию о страховании. В США защита данных о здоровье регулируется главным образом Законом об ответственности и переносе данных о страховании здоровья граждан от 1996 года (Health Insurance Portability and Accountability Act, HIPAA).

Данные частных лиц

Такие данные часто называют информацией, идентифицирующей личность (Personally Identifiable Information, PII). Сюда входят номера карт социального страхования, адреса, номера телефонов и адреса электронной почты. Эту информацию злоумышленники могут использовать для кражи личных данных, шантажа и пр. Основой для принятых США стандартов, действующих в настоящее время в отношении этих данных, послужил закон «О неприкосновенности частной жизни» 1974 года.

Военные и правительственные данные

Любые данные, имеющие отношение к правительству или армии, считаются сверхконфиденциальными, тем более информация о военных операциях и военнослужащих. В любой организации, обслуживающей и хранящей такие данные, предусмотрены очень строгие процедуры для их защиты.

Конфиденциальные данные и коммерческие тайны

К ним относятся любые данные, которые должны держаться в секрете для сохранения конкурентоспособности бизнеса. Это данные, представляющие собой интеллектуальную собственность (Intellectual Property, IP), коммерческую

тайну, финансовые показатели и отчеты о результатах деятельности и прибыльности компании. В эту категорию также входит информация о клиентах и продажах.

Понимание природы данных, расположенных в конкретных хранилищах, имеет решающее значение — оно позволяет грамотно выбрать вид шифрования и защиты. Это также важно для организации совместной работы и проведения обучения. В противном случае в быстро меняющихся условиях разработчик может непреднамеренно поместить конфиденциальные данные в не предназначенные для этого незащищенные хранилища.

Существуют также базовые стандарты, на которые нужно ориентироваться абсолютно всегда.

- Интерфейсы веб-администрирования, напрямую обращающиеся к базам данных, должны всегда использовать SSL или безопасный прокси-сервис. Мы в обоих случаях используем SSL. Преемником SSL 3.0 стал протокол Transport Layer Security (TLS), и обычно оба протокола называют SSL. TLS 1.0 имеет уязвимости, поэтому его следует считать не соответствующим требованиям (http://bit.ly/2zxqT5t).
- Для подключения к серверам стоит использовать протокол SSH2 или протокол удаленного доступа (Remote Desk Protocol, RDP).
- Для входа администратора в базу данных следует использовать отдельную администраторскую сеть и задействовать TLS версии 1.1 или 1.2, если БД это позволяет.
- Все протоколы SSL должны быть защищены достаточно стойким шифрованием. Защищенность зашифрованного сеанса определяется шифром, согласованным между сервером и клиентом.

Сначала мы рассмотрим шифрование данных при передаче.

Передача данных

Данные приходится передавать по сетям. С точки зрения DBRE-специалиста, желающего защитить данные, это неизбежный и печальный факт. Подобно тому как деньги и ценности приходится перевозить в бронированных грузовиках, так и данные должны транспортироваться должным образом. Это очень уязвимый процесс, который также называется *data in flight* (буквально: *«данные в полете»*).

Прежде чем углубляться в подробности, стоит убедиться, что DBRE-специалист разбирается в различных компонентах и оптимальных методах шифронабора.

Средства шифрования изнутри

Каждый сервер базы данных связывается с клиентами, используя определенный *шифронабор* (cipher suite). Важно понимать последствия конкретной реализации средств шифрования (криптозащиты) для БД, чтобы обеспечить необходимую защиту данных. Устанавливать требования и стандарты должен был бы отдел информационной безопасности, но если эта работа досталась вам, важно понимать следствия из конкретных особенностей реализаций баз данных. Вот пример стека средств шифрования:

```
ECDHE-ECDSA-AES128-GCM-SHA256
```

Первая часть, `ECDHE`, — алгоритм обмена ключами. В данном примере используется обмен ключами по *эллиптической кривой* с применением эфемерных ключей (EDHE). Другими вариантами значений могут быть `RSA`, `DH` и `DHE`. Обмен эфемерными ключами происходит по протоколу Диффи — Хеллмана (DHE) (https://www-ee.stanford.edu/~hellman/publications/24.pdf). Эфемерные (одноразовые) ключи обеспечивают совершенную прямую секретность (Perfect Forward Secrecy, PFS), то есть, если долгосрочный ключ подписи сервера будет скомпрометирован, это не нарушит конфиденциальность прошедшего сеанса. Если сервер использует временный ключ, то подписывает его своим долгосрочным ключом (долгосрочный ключ — это обычный ключ, доступный в сертификате сервера)[1]. Алгоритм DHE считается более надежным, чем EDHE, и лучше выбирать его.

Следующая часть, `ECDDSA`, характеризует алгоритм подписи, который используется для подписи параметров обмена ключами. Здесь предпочтительнее использовать RSA, а не DSA или DSS, которые могут быть очень слабыми в зависимости от источника подписи.

Далее, часть `AES128` относится к шифру из набора. В данном случае это расширенный стандарт шифрования (Advanced Encryption Standard, AES) со 128-битным ключом. Затем следует способ обработки шифра — счетчик с аутентификацией Галуа (Galois/Counter Mode, GCM), который обеспечивает аутентифицированное шифрование. GCM поддерживается только AES, Camellia и Aria, поэтому эти шифры идеально подходят. Для AES Национальный институт стандартов и технологий США (National Institute of Standards and Technology, NIST) выбрал в качестве стандарта три компонента, каждый размером блока 128 бит, но с тремя разными длинами ключа: 128, 192 и 256 бит.

Наконец, `SHA-256` представляет собой реализацию хеш-функции кода аутентификации сообщений с ключом (Message Authentication Code, MAC). SHA-256 — это функция хешированного MAC (HMAC), используемая органами сертификации

[1] Подробнее читайте в статье: Transport Layer Protection Cheat Sheet. http://bit.ly/owasp-cheat-sheet.

для подписи сертификатов и списков отозванных сертификатов (Certificates Revocation List, CRL). Этот алгоритм применяется для создания главного секретного ключа. Получатели сообщений используют его для проверки правильности содержимого. После того как одна сторона отправила свое зашифрованное сообщение, а затем получила и проверила зашифрованное сообщение от другой стороны, она может начать отправлять и получать данные приложения через соединение. Предпочтительной реализацией этого алгоритма является SHA2. В него входит четыре вида хеш-функций: SHA224, SHA256, SHA384 и SHA512.

При выборе реализации SSL для базы данных важно учитывать список шифров: он показывает последовательность сканирования шифров, пока не будет найден соответствующий шифр, доступный как клиенту, так и серверу.

При оценке необходимости в шифровании для защиты связи нужно учитывать не только типы данных (перечислены выше), но и способы их переноса, в частности следующие:

- коммуникации внутри сети;
- коммуникации вне сети.

Каждый из этих видов передачи данных требует особого рассмотрения и согласованного набора допущений. На базе этих допущений формируется перечень требований, для каждого из которых потребуется соответствующая реализация.

Коммуникации внутри сети

В защищенной подсети, как правило, предполагается, что коммуникации защищены на сетевом уровне и, таким образом, большинство правил не требуют дополнительной защиты каналов связи (таких как само сетевое соединение). Это означает, что в защищенной сети серверы приложений, запрашивающие или передающие данные, а также осуществляющие репликацию между серверами и другие межсетевые коммуникации, не должны устанавливать зашифрованные соединения. Это хорошо, поскольку именно этим обычно занята база данных, а шифрование довольно сильно загружает процессор. При этом, если в базе данных хранятся конфиденциальные данные, они все равно должны быть каким-то образом зашифрованы. В идеале это сделать на уровне приложения, когда данные помещаются в саму базу данных. Мы обсудим это подробнее далее, когда будем рассматривать *неактивные данные* (в состоянии покоя, *data at rest*).

Если конфиденциальные данные, хранящиеся в базе, по различным причинам не могут быть защищены, стоит рассмотреть вопрос о необходимости использования связи с криптографическим закрытием информации для всего, что имеет отношение к базе данных.

Коммуникация между сетями

В тех случаях, когда устанавливается соединение между двумя сетями, которыми владеет и управляет ваша организация, или между вашей сетью и Интернетом, при транспортировке любых данных, независимо от их конфиденциальности, необходимо использовать виртуальную частную сеть (Virtual Private Network, VPN) с применением протокола IPSec или SSL.

Аналогичным образом для большинства коммуникаций интернет-клиенты должны обмениваться данными с балансировщиками нагрузки по протоколу SSL/TLS. Это увеличивает нагрузку на процессор для клиентов и балансировщиков нагрузки, но вряд ли вы согласитесь на то, чтобы данные между клиентами и вашим сервером передавались в открытую.

Теперь, понимая необходимость использования SSL, рассмотрим варианты архитектуры, которые поддерживают SSL.

Установление защищенных соединений для передачи данных

Современные системы управления базами данных обычно в той или иной степени поддерживают SSL. Но среди них есть исключения, такие как Redis, так что при выборе хранилища данных, соответствующего вашим запросам, важно убедиться в этой поддержке. Незашифрованные конфиденциальные данные не должны кэшироваться!

Следует отметить, что на практике затраты на SSL, как правило, очень невелики. Большая часть вычислительных ресурсов используется при инициализации соединения и редко превышает 2 % загрузки процессора, а задержка редко возрастает более чем на 5 миллисекунд (http://bit.ly/2zykkiT). Для некоторых шифров, таких как AES, предусмотрены встроенные инструкции для большинства современных процессоров, что существенно повышает их скорость по сравнению с программно реализованными шифрами.

Существует многоуровневый набор подходов, которые можно применять для защиты соединений.

Простейшее шифрование соединения. На самом базовом уровне сначала необходимо настроить сервер базы данных, чтобы он запрашивал защищенный обмен данными для всех соединений. После этой настройки создается сертификат центра сертификации (Certificate Authority, CA). Он используется для подписи сертификата открытого и закрытого ключа сервера. Этот же сертификат используется на стороне клиента для создания клиентских сертификатов открытых и закрытых

ключей. Если клиенты сами хранят свои ключи и серверы настроены надлежащим образом, то все соединения считаются зашифрованными.

Как вы понимаете, ключи нельзя хранить там, где их можно легко взломать и использовать. Это означает, что динамическая конфигурация может применяться для непосредственной загрузки ключей в память приложения, но не для их хранения непосредственно в файловой системе клиента. Для этого есть лучшие способы.

Надежное хранение секретной информации. Защита соединения по протоколу SSL является важным первым шагом, однако по-прежнему остаются уязвимости. В конце концов, если кому-то удастся получить доступ к клиентским хостам, то соединения, использующие эти ключи, можно будет задействовать для запроса данных. Использование защищенного сервиса инфраструктуры управления ключами, такого как хранилище Hashicorp, сервиса управления ключами Amazon (KMS) или любого другого решения позволяет отделить хранилище ключей и управление ими от тех, кто получает доступ к данным.

Кроме того, другие фрагменты информации, применяемые для доступа к базе данных, такие как имя пользователя, пароль, IP-адреса и порты, могут храниться удаленно в некоторых из этих сервисов. Это гарантирует, что учетные данные не будут сохраняться в файловой системе, где злоумышленник сможет их получить и использовать вне контекста приложения.

Динамическое создание пользователей баз данных. Построив на предыдущих двух этапах надежный сервис обеспечения безопасности, такой как Vault, естественно было бы использовать его же и для динамического создания эфемерных учетных записей пользователей в хранилище данных. Это позволяет хосту приложения регистрировать и запрашивать учетную запись пользователя, которая будет создана в этот момент. Использование таких ролей, как «только для чтения», облегчает автоматическое присвоение различных привилегий, и соответствующим пользователям может быть ограничено время существования, благодаря чему гарантируется, что любой доступ, который потенциально может быть перехвачен, будет предоставлен в течение ограниченного времени. Кроме того, теперь появляется возможность привязать пользователей к конкретным хостам приложений. Это позволяет проводить аудит запросов и доступа, что было бы сложнее сделать в среде, где предусмотрены одни и те же имена пользователей для всего пула серверов.

При использовании сочетания технологий SSL и VPN должна быть возможность зашифровать любой канал связи, в зависимости от потребностей хранимых данных. Кроме того, можно воспользоваться преимуществами сервиса управления секретными данными, чтобы уменьшить число возможных атак на файлы конфигурации и ключей, находящихся в файловых системах, для тех, у кого есть права чтения и появляется возможность злоупотребления в операционной системе. Такова концепция защиты данных при передаче. Теперь перейдем к данным, находящимся в покое. Начнем с данных, хранящихся в самой базе.

Данные, хранящиеся в базе

Такие данные также называют *данными в обработке (data in use)*. Данные, хранящиеся в базе данных, должны быть доступны для приложений, аналитиков и пользовательских процессов. Это означает, что любой метод шифрования должен разрешать доступ к данным для тех, кто прошел аутентификацию и имеет соответствующие привилегии, и в то же время воспрепятствовать тем, кто желает получить доступ к данным со злонамеренной целью. Если пользователю удается пройти аутентификацию в базе данных, то он обычно может читать любые данные, в отношении которых у него есть соответствующие права на чтение.

Всех потенциальных злоумышленников можно разделить на следующие категории:

- внешний злоумышленник;
- внутренний злоумышленник;
- администратор.

Внешние злоумышленники получают доступ к базе данных или ее серверу для извлечения ценной информации. Внутренние злоумышленники входят в группы доверенных лиц с правами доступа к базе данных или ОС и пытаются получить информацию, выходящую за пределы предоставленных им привилегий. Администраторы — это люди, у которых есть права администратора на уровне базы данных и/или ОС и которые используют эти права для получения ценной информации.

Большая часть типов данных, перечисленных в начале этого раздела, требует еще более усиленного шифрования, чтобы гарантировать, что только соответствующие пользователи могут их прочитать. Рассмотрим доступные варианты шифрования, их особенности и потенциальные недостатки. Мы также хотели бы напомнить вам, что стандарты и рекомендованные методы шифрования здесь актуальны так же, как и при описанном ранее SSL-шифровании.

Безопасность на уровне приложений

При таком подходе выбор таблиц и столбцов, требующих шифрования, определяется при моделировании угроз. Используя библиотеки шифрования, приложение шифрует данные перед их отправкой в хранилище. Затем данные передаются так же, как и любые другие строковые или двоичные данные. Данные извлекаются аналогичным образом, и приложение знает, как их расшифровать, прежде чем передать для использования. Это шифрование и дешифрование может выполняться с помощью таких библиотек, как Bouncy Castle и OpenSSL.

Использование библиотек на уровне приложений обеспечивает переносимость базы данных. Даже если серверная часть изменится, вы все равно сможете вы-

полнять шифрование и дешифрование как раньше. Это также позволяет контролировать библиотеки шифрования. Отдел безопасности может поместить эти общие библиотеки в систему контроля версий, откуда она будет доступна всем желающим, и больше не будет необходимости проверять этот код на соответствие, так как он одобрен и используется во всей организации. Наконец, данный метод допускает выборочное шифрование, при котором остальные столбцы и таблицы остаются незашифрованными, что упрощает создание отчетов, индексацию и обработку запросов.

Основной недостаток такого подхода состоит в том, что он не является универсальным и не применяется ко всем данным в базе. Таким образом, при моделировании и вводе в приложение новых данных разработчики должны рассмотреть вопрос, нуждается ли этот новый фрагмент данных в шифровании, а затем фактически реализовать это шифрование. Такой подход также требует, чтобы все остальные клиенты в конвейере данных, которые нуждаются в чтении этих данных, использовали библиотеки дешифрования.

Шифрование на уровне приложений обеспечивает наибольшую гибкость за счет скорости разработки.

Шифрование базы данных с помощью плагина

Плагин шифрования использует пакет шифрования, установленный в самой базе данных. Этот метод не зависит от приложения и требует меньше дополнительного кода. В состав разных плагинов часто могут входить выборочное шифрование на уровне столбцов, функции контроля доступа и аудит доступа.

Выполнять полное шифрование базы данных одним ключом, как правило, не рекомендуется, поскольку если пользователю удастся найти уязвимость, то он получит больший, чем необходимо, доступ для чтения данных в любом месте базы, где используется этот ключ. Например, внутренний пользователь с доступом к ключу шифрования может получить данные пользователя с расширенными правами и затем получить доступ к данным вне своей группы безопасности. Шифрование таблиц, принадлежащих разным группам безопасности, с использованием разных ключей гарантирует, что пользователи смогут дешифровать только те объекты, которые относятся к их группе безопасности. Это означает, что любые плагины, которые вы собираетесь использовать, должны иметь функции выборочного шифрования и контроля доступа, иначе эти плагины неэффективны.

В отличие от шифрования на уровне приложений при выборе такого варианта возможны проблемы переносимости между базами данных. Таким образом, если вы работаете в стартапе или в среде с часто меняющимися требованиями, это решение может оказаться слишком негибким.

Прозрачное шифрование базы данных

Некоторые устройства безопасности шифруют и дешифруют все коммуникации, проходящие через базу данных. Это относительно простой подход, который может облегчить шифрование, но он увеличивает издержки за счет шифрования всех данных. Однако универсальность сводит эти издержки к минимуму.

Влияние на скорость обработки запросов

Несмотря на то что шифрование данных является относительно тривиальным действием, запрос таких данных может обрабатываться медленно, а шифрование повлияет на схему и структуру запроса. Шифрование данных на уровне столбцов или таблиц не всегда поддерживает запросы диапазона или поиск строк. Поэтому в запросах необходимо учитывать, каким образом данные будут фильтроваться и сортироваться.

В большинстве функций шифрования не сохраняется последовательность данных, поэтому с такими функциями нельзя использовать *индекс со структурой B-дерева*, который широко применяется для индексации диапазонов зашифрованных данных. Незашифрованные поля часто можно использовать для эффективной фильтрации. Например, можно задавать фильтры диапазона дат, чтобы уменьшить набор данных, который необходимо просмотреть в процессе поиска зашифрованных значений. Чтобы обеспечить более эффективный запрос зашифрованных данных, можно сохранить в схеме код аутентификации сообщений с хеш-ключом (Keyed-Hash Message Authentication Code, HMAC) для зашифрованного поля и предоставить ключ хеш-функции. Последующие запросы защищенных полей, которые содержат HMAC запрашиваемых данных, не будут содержать в запросе значения в виде открытого текста. Выбрав такой подход, можно выполнять запрос к зашифрованным данным в БД, не раскрывая текстовые значения в запросе. Он также защищает от манипулирования данными в базе со стороны пользователей, у которых нет информации, необходимой для создания HMAC (http://bit.ly/2zzUoUe).

Благодаря индексам в этих хешированных полях можно получить почти такую же производительность, что и без шифрования, однако раскрывается информация о частоте и количестве индексированных значений. Аналогичным образом злоумышленник может получить информацию о зашифрованном значении базы данных по его положению в индексе или даже искать другие вхождения хеша. При долгосрочном доступе можно получить информацию о данных, наблюдая и анализируя изменения во времени. Например, после вставки данных осведомленный пользователь может найти нужные значения по событиям и положению в индексе.

Таким образом, в зависимости от ценности данных, можно вводить дополнительные методы маскировки, чтобы уменьшить вероятность того, что сторонний наблюдатель сможет определить взаимосвязи или значения на основе хеша, его связи с другими хешами и значениями последующих вставок. Сюда может входить

добавление фиктивных данных при каждой вставке или пакетная вставка, чтобы не допустить последовательные наблюдения за атомарными вставками.

Это всего лишь поверхностный обзор разных аспектов схемы для обеспечения производительности и безопасности, однако мы посчитали важным затронуть эти проблемы и привести несколько примеров подходов к смягчению последствий, чтобы вы не думали, что шифрование базы данных на этапе планирования — это легко и просто.

Данные, которые хранятся в базе, в итоге все равно находятся в файловой системе. В ней же хранятся журналы, дампы данных и резервные копии, которые тоже необходимо учитывать при защите данных. Итак, рассмотрим шифрование данных, пребывающих в состоянии покоя, на уровне файловой системы.

Данные в файловой системе

Благодаря шифрованию данных, находящихся в процессе передачи, и неактивных данных мы обеспечили значительный уровень защиты. Тем не менее существует возможность доступа к данным непосредственно на диске, магнитной ленте или другом носителе. Как и с остальными методами смягчения ущерба, здесь есть несколько подходов. Выбирая метод решения, важно учитывать объем хранимых данных, возможности процессора, влияние задержки на операции чтения и записи, а также частоту доступа к данным.

Злоумышленники могут выбрать прямую или косвенную атаку на данные в файловой системе. При прямых атаках на хранилища данных злоумышленник обращается к файлам БД напрямую, вне программного обеспечения базы. Это может быть копирование файлов данных с сервера по сети путем физического удаления устройств хранения или получения данных из инфраструктуры резервного копирования. При косвенных атаках злоумышленник может получить информацию о схеме, данные журнала и метаданные из файлов, используемых базой.

Кроме стандартных сетевых стратегий и стратегий контроля доступа, необходимо добавить шифрование данных файловой системы, чтобы гарантировать, что злоумышленник, которому удастся обойти эти меры безопасности, не сможет воспользоваться нашими данными. При выборе способа шифрования хранилища нужно учитывать несколько уровней: данные, хранящиеся в файловой системе, данные, помещаемые в файловую систему, и данные об устройстве.

Шифрование данных перед помещением в файловую систему

Когда данные помещаются в файловую систему, они могут быть зашифрованы автоматически. Это делается по тем же соображениям, что были описаны в предыдущем подразделе. Обычно это относится к данным, загружаемым в файловую

систему, таким как резервная копия или файл данных, подлежащий импорту. При шифровании на этом уровне всегда можно узнать статус шифрования критически важных файлов.

Кроме того, эти данные можно разбить на фрагменты для распределения между несколькими устройствами хранения. Это отличный вариант для резервного копирования конфиденциальных сведений и больших дампов данных, поскольку он не позволяет человеку, имеющему доступ к одному из устройств хранения, получить полный набор данных. Такое разделение также допускает распараллеливание операций чтения и записи, что обеспечивает более быстрое восстановление. Как правило, выбор такого шлюза хранения определяется компромиссом между временем разработки и временем обслуживания.

Шифрование файловой системы

Большинство хранилищ данных создают собственные файлы для метаданных, журналов и хранения данных. Поэтому система должна быть готова к шифрованию на уровне файловой системы. Это можно сделать поверх существующей файловой системы, в виде файловой системы с многоуровневым шифрованием, непосредственно в файловой системе с помощью встроенных механизмов шифрования или ниже файловой системы, на блочном уровне.

Размещение зашифрованной файловой системы поверх существующей позволяет использовать любую файловую систему в качестве базовой. Это довольно гибкий вариант, поскольку его можно использовать для шифрования отдельных каталогов, а не всего тома. В качестве примеров на основе Linux можно привести eCryptfs и EncFs. Для этого необходимо, чтобы ключи предоставлялись вручную или через интерфейс управления ключами (Key Management Interface, KMI). Во многих файловых системах, таких как ZFS и BTRFS, также предусмотрены дополнительные опции шифрования, хотя важно проверить, видны ли в них незашифрованные метаданные.

Системы шифрования на уровне блоков работают под файловой системой, зашифровывая дисковые блоки по отдельности. В Linux для этого есть, в частности, Loop-AES, dm-crypt и Vera. Все эти инструменты работают ниже уровня файловой системы, используя драйверы устройств в пространстве ядра. Эти инструменты уместны, если нужно, чтобы все данные, записанные в томе, были зашифрованы, независимо от того, в каком каталоге они хранятся.

Все эти решения влияют на производительность, и при их выборе необходимо учитывать требования безопасности. Имеет смысл размещать журналы, метаданные и другие подобные файлы в зашифрованных файловых системах. Но как быть с файлами данных, к которым осуществляется доступ из самой базы? Многие считают, что шифрование основных данных на уровне приложений или на уровне

столбцов обеспечивает необходимое шифрование в базе данных. Это позволяет не шифровать сами файлы БД, что повышает производительность. Объединяя такое решение с шифрованием файловой системы для хранения журналов, файлов метаданных и других файлов системного уровня, можно создать эффективное многоуровневое решение, которое не влияет на производительность.

Шифрование на уровне устройств

Вы также можете использовать носитель со встроенной расшифровкой. При выборе такого решения повышается стоимость хранения, кроме того, оно имеет сомнительную ценность, поскольку характеризуется множеством известных уязвимостей. Однако шифрование на этом уровне, безусловно, углубляет защиту данных.

Как вы могли понять из этого обсуждения, шифрование данных заслуженно занимает видное место в процессе проектирования, реализации и аудита. В этом разделе мы обсудили защиту путем шифрования передаваемых данных, данных, хранящихся в базе, и данных, находящихся в состоянии покоя в файловых системах. Шифрование данных — это последний бастион безопасности, обеспечивающий защиту при нарушении контроля доступа, усиления защиты кода и регулярных исправлений. Признать важность этого шифрования означает признать тот факт, что каждый уровень безопасности является уязвимым и для обеспечения разумного уровня безопасности необходимо обеспечить защиту данных на всех уровнях. Устанавливая шифрование, независимо от уровня, вам всякий раз необходимо задавать себе следующие контрольные вопросы.

- Все ли данные классифицированы в соответствии с их важностью?
- Существует ли стандарт шифронаборов и проверяется ли он на соответствие?
- Регулярно ли мы следим за сообщениями о новых уязвимостях и эксплойтах и учитываем их?
- Сообщают ли всем новым сотрудникам в командах SWE, SRE, Ops и DBRE о централизованных библиотеках и стандартах шифрования?
- Эффективно ли налажено управление ключами, в том числе обмен, удаление и тестирование?
- Проводится ли регулярное автоматическое тестирование на взлом ключевых компонентов, включая журналы, резервные копии, критически важные таблицы и соединения с базой данных?

Как и при любом автоматическом и ручном тестировании, здесь важно понимать, что невозможно протестировать все. Именно поэтому, сосредоточив внимание на компонентах с наибольшим уровнем риска и наиболее уязвимых для эксплойтов, вы правильно распределите усилия, расставите приоритеты и обеспечите тесную обратную связь при непрерывном процессе тестирования и совершенствования приложения.

Резюме

Теперь вы не только глубже изучили особенности разных уровней защиты базы данных, но и узнали, как более эффективно обеспечить безопасность в вашей организации. Как и в отношении всех других компонентов, описанных в этой книге, DBRE-специалист не может нести единоличную ответственность за эту функцию. Используя устаревшие подходы, администратор баз данных просто не сможет эффективно работать в высокоскоростных динамических средах, которые требуют менталитета DBRE. Вместо этого необходимо активно сотрудничать со всеми командами специалистов, упомянутыми в этой главе, чтобы делиться с ними своими знаниями, необходимыми для построения платформ самообслуживания, общедоступных библиотек и групповых процессов.

В этой, как и в предыдущих восьми главах, мы постарались создать прочную основу не только для деятельности отдела эксплуатации, но и для эффективного сотрудничества и поддержки со стороны остальных технических отделов, которые полагаются на вас как на DBRE-специалиста. Теперь мы направим силы на то, чтобы помочь вам разобраться в великом множестве вариантов сохранения базы данных и в том, как в них реализованы ключевые технологии для обеспечения гибкого, масштабируемого и производительного хранения и извлечения данных.

10 Хранение, индексирование и репликация данных

Обсуждение различных видов операций в большей части этой книги было подготовкой к тому, чтобы углубленно изучить хранилища данных. Самое главное в любом хранилище данных — это то, что там... внимание... хранятся данные! В этой главе мы объясним, каким образом отдельный узел структурирует свое хранилище данных, как большие наборы данных разбиваются на разделы и как узлы реплицируют данные между собой. Это будет та еще глава!

Темы, рассмотренные в этой книге, касаются главным образом вопросов надежности и эксплуатации, поэтому здесь мы исследуем схемы хранения и доступа, чтобы упростить вам выбор инфраструктуры. Мы изучим характеристики производительности и убедимся, что у вас, как у инженера по обеспечению надежности баз данных (DBRE), есть информация, которая позволит помочь команде разработчиков выбрать подходящие хранилища данных для своих сервисов. Для более глубокого и подробного изучения этой темы мы настоятельно рекомендуем вам прочитать книгу Мартина Клеппмана[1] (Martin Kleppmann).

Хранение структуры данных

Данные в базах данных традиционно хранятся в виде сочетаний таблиц и индексов. Таблица — это основной вариант хранения, а индекс — оптимизированное подмножество данных, упорядоченное для сокращения времени доступа. Сейчас, когда хранилища данных получили широкое распространение, эта ситуация значительно изменилась. Для того чтобы иметь возможность настраивать и оптимизировать подсистемы хранения баз данных, крайне важно понимать, как происходит запись и чтение данных из хранилища.

Чтобы понять, как хранятся данные в базе, вам необходимо изучить не только то, как хранятся данные в сыром виде, но и то, как они извлекаются из базы. В случае

---

[1] *Клеппман М.* Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб.: Питер, 2018.

больших наборов данных для доступа к определенным подмножествам данных со сколь-нибудь приемлемым уровнем задержки часто требуются специализированные структуры хранения, называемые индексами, которые ускоряют поиск и извлечение этих данных. Таким образом, при выборе хранилища необходимо учитывать требования к хранилищу и операциям ввода/вывода (I/O) для помещения данных на диск и в индексы, а также требования к операциям ввода/вывода для получения этих данных.

Хранение данных в виде таблиц

Большая часть информации в этом подразделе относится к традиционным реляционным системам. Мы начнем с них, а затем обсудим некоторые из самых распространенных альтернативных вариантов хранения. В реляционных базах данные хранятся в контейнерах, называемых *блоками*, или *страницами*, которые соответствуют определенному количеству байтов на диске. В терминологии одних баз данных эти контейнеры называются блоками, в других — страницами. В данной книге мы будем называть их блоками. Блоки — наиболее мелкая детализация при хранении данных в базе. Так, в базах Oracle Database данные хранятся в виде блоков. Блок или страница имеют фиксированный размер, подобно блокам на дисках. Блок — это минимальное количество байтов, которое можно прочитать или записать при получении доступа к данным в базе. Другими словами, если длина строки таблицы равна 1 Кбайт, а размер блока — 16 Кбайт, то при выполнении операции чтения все равно будет прочитано 16 Кбайт. Если размер блока базы данных меньше, чем размер блока файловой системы, то при чтении нескольких страниц, если они не согласованы с блоками файловой системы, вы все равно будете напрасно тратить время на дополнительные операции ввода-вывода. Это можно визуально представить в виде рисунка (рис. 10.1).

В блоке также хранятся некоторые метаданные, обычно в форме заголовка и трейлера, или концовки. Эти метаданные включают в себя дисковую адресную информацию, информацию об объекте, которому принадлежит блок, а также информацию о строках и действиях, которые выполнялись в этом блоке. В Oracle, начиная с версии 11g, Release 2, объем служебных данных блока составляет от 84 до 107 байт. В MySQL InnoDB, начиная с версии 5.7, заголовок и трейлер занимают 46 байт. Кроме того, у каждой строки данных есть собственные метаданные, в том числе информация о столбцах, ссылки на другие блоки, по которым разбросана информация этой записи, а также уникальный идентификатор строки[1].

Блоки данных часто организованы в виде контейнера большего размера, называемого экстентом. Из соображений эффективности именно экстент часто является единицей выделения, когда в табличном пространстве необходимо ввести новые

[1] *Cole J.* The physical structure of records in InnoDB. http://bit.ly/2zykQ0j.

блоки. Табличное пространство, как правило, является самой большой структурой данных, сопоставленной с одним или несколькими физическими файлами, которые могут размещаться на диске по мере необходимости. В системах, привязанных непосредственно к физическим дискам, файлы табличного пространства могут размещаться на разных дисках, чтобы уменьшить конфликты ввода-вывода. Но при подходах, которые мы рассматриваем в этой книге, такая организация ввода-вывода не всегда имеет место. Большие универсальные RAID-массивы с чередованием, в том числе зеркалированные, позволяют максимально ускорить ввод-вывод, не затрачивая много времени на тонкую настройку. В других ситуациях при условии быстрого восстановления и перехвата управления в случае сбоя можно обратить внимание на простые тома или даже эфемерные хранилища — это обеспечит простое управление и потребует минимальных накладных расходов.

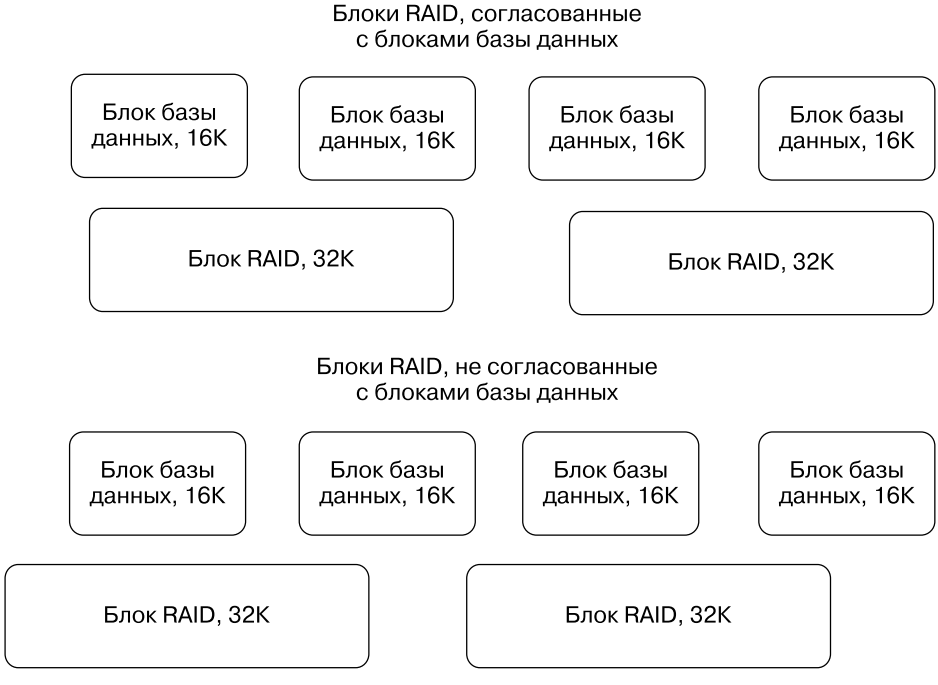


Рис. 10.1. Согласованные и несогласованные конфигурации блоков базы данных и файловой системы

Структуры типа B-деревьев

В большинстве баз данные структурированы в виде двоичного дерева, также известного как *B-дерево*. B-дерево — это структура данных, которая автоматически балансируется, сохраняя данные отсортированными. B-дерево оптимизировано для

чтения и записи блоков данных, поэтому B-деревья обычно используются в базах данных и файловых системах.

Таблицу или индекс в виде B-дерева можно представить как дерево, перевернутое вверх ногами. Началом индекса, построенного на основе ключа, является корневая страница. В качестве ключа используется один или несколько столбцов. Большинство таблиц в реляционных базах данных формируются на основе первичного ключа, который может быть определен явно или неявно. Например, первичный ключ может быть целым числом. Если приложение ищет данные по конкретному идентификатору или диапазону идентификаторов, то именно этот ключ будет использоваться для их поиска. Кроме B-дерева первичного ключа, могут быть определены вторичные индексы для других столбцов или наборов столбцов. В отличие от исходного B-дерева эти индексы хранят не всю строку, а только индексированные данные. Таким образом, эти индексы намного меньше и гораздо легче помещаются в памяти.

B-дерево называют деревом, потому что для получения нужных данных при навигации по дереву вы делаете выбор из двух и более дочерних страниц. Как только что было отмечено, страница содержит строки данных и метаданные. Эти метаданные хранят указатели на страницы, расположенные ниже по дереву, также называемые *дочерними страницами*. Корневая страница имеет две дочерние страницы или более, которые также называются *потомками*. Дочерняя страница, или узел, может быть внутренним или конечным узлом. Во внутренних узлах хранятся поворотные ключи (pivot key) и дочерние указатели; они используются для перенаправления чтения по индексу на тот или иной узел. Конечные узлы содержат ключевые данные. Согласно этой структуре создается самобалансирующееся дерево, в котором можно искать лишь на нескольких уровнях, что позволяет искать только несколько дисков, чтобы найти указатели на нужные строки. Если необходимые данные находятся внутри самого ключа, вам даже не нужно следовать указателю на строку.

Двоичное дерево допускает запись. При вставке данных в B-дерево правильный концевой узел обнаруживается в процессе операции поиска. При создании узлов они не упаковываются — остается достаточно места для вставки дополнительных данных. Если в узле есть место, то данные вставляются в нужном порядке. Если узел заполнен, то выполняется разделение. При разделении определяется новая медиана и создается новый узел, затем записи перераспределяются соответствующим образом. Данные об этой медиане вставляются в родительский узел, что может привести к дополнительному разделению, и так далее, вплоть до корневого узла. Операции изменения и удаления также начинаются с поиска правильного концевого узла, после чего выполняется изменение или удаление. Изменение может привести к разделению, если при этом увеличивается размер данных до той точки, где наступает переполнение узла. Удаление также может привести к изменению баланса.

Формирование создаваемых с нуля (greenfield) баз данных начинается с последовательных операций записи и чтения. Такие БД отличаются низкой задержкой операций записи и чтения. По мере роста базы данных разбиение приводит к тому, что

операции ввода-вывода станут случайными. В итоге увеличивается и время чтения и записи. Именно поэтому мы вынуждены настаивать на использовании в тестах реалистичных наборов данных, чтобы гарантировать, что они будут соответствовать длительной работе в реальных условиях, а не упрощенному выставочному образцу.

Однострочные операции записи требуют как минимум полной перезаписи одной страницы. В случае разделения может быть перезаписано несколько страниц. Эта сложная операция должна быть атомарной, и в то же время в случае сбоя возможны искажения и появление потерянных страниц. При выборе хранилища данных очень важно понимать, какие методы предоставляются для предотвращения этого. Вот несколько примеров таких методов:

- перед выполнением сложных операций записи на диск вносятся данные в журналы операций записи, также известные как Write Ahead Logs (WAL);
- ведется запись в журналы событий на случай восстановления;
- создаются журналы повторного выполнения операций с образами измененных данных до и после выполнения операции.

Таким образом, важнейшим параметром при настройке базы данных для главного хранилища является размер блока базы данных. Мы уже обсудили важность согласования размеров блоков БД с размерами блоков данных на диске, но этого недостаточно. Например, при использовании твердотельных накопителей (Solid-State Drive, SSD) блоки меньшего размера иногда обеспечивают гораздо лучшую производительность при обходе B-деревьев. В случае больших блоков задержка на SSD иногда превышает задержку жестких дисков (Hard Disk Drive, HDD) на 30–40 %. Поскольку в структурах B-дерева используются и чтение, и запись, то это необходимо учитывать.

Ниже приводится краткий перечень свойств и преимуществ B-деревьев:

- отличная производительность для запросов на основе диапазона;
- не самая лучшая модель для поиска в одной строке;
- ключи представлены в виде упорядоченного списка, что удобно при поиске ключей и сканировании диапазона;
- структура сводит к минимуму операции чтения страниц в случае больших наборов данных;
- если не упаковывать ключи в каждую страницу, получаем эффективные операции удаления и вставки, при этом редко приходится выполнять операции разбиения и слияния;
- производительность намного выше, если вся структура помещается в памяти.

Есть и другие варианты для индексации данных. Наиболее часто из них используется хеш-индекс.

Как уже говорилось, структура в виде B-дерева довольно популярна в реляционных базах данных. Если вы работали с такими средами, то вам, вероятно, уже встречались B-деревья. Однако есть и другие варианты хранения данных, которые активно применяются на практике. Далее мы рассмотрим структуры журналов только для записи (http://bit.ly/2zyttIz).

Отсортированные строковые таблицы и журнально-структурированные деревья со слиянием

BigTable, Cassandra, RocksDB (доступные в MySQL через MyRocks и MongoDB) и LevelDB — это примеры баз данных, в которых для основного хранилища используются отсортированные строковые таблицы (Sorted-String Table, SST). Термины SSTable и Memtable первоначально появились в документе Google BigTable (https://research.google.com/archive/bigtable-osdi06.pdf), который с тех пор является источником вдохновения для целого ряда систем управления базами данных.

В SST используется несколько файлов, внутри каждого из которых содержится набор отсортированных пар «ключ — значение». В отличие от описанного ранее блочного хранилища здесь нет необходимости тратить ресурсы на метаданные на уровне блоков и строк. Ключи и их значения непрозрачны для СУБД и хранятся в виде произвольных больших двоичных объектов (Binary Large Object, BLOB). Поскольку ключи и значения хранятся в отсортированном виде, их можно читать последовательно и рассматривать как индекс по ключу, по которому они отсортированы.

Существует алгоритм, который объединяет хранящиеся в памяти таблицы, выполняет пакетную очистку и периодическое сжатие в SST. Этот алгоритм относится к архитектуре журнально-структурированного дерева со слиянием (Log-Structured Merge, LSM, рис. 10.2). Такие деревья описаны Патриком О'Нилом (Patrick O'Neill) в статье по адресу https://www.cs.umb.edu/~poneil/lsmtree.pdf.

С помощью LSM SST-таблицы записываются путем периодических сбросов данных, которые хранились в памяти. После того как данные будут сброшены, отсортированы и записаны на диск, они больше не изменяются. Элементы таблицы соответствий «ключ — значение» не могут быть добавлены или удалены. Это эффективно для наборов данных, предназначенных только для чтения, поскольку можно перенести SST в память для быстрого доступа. Даже если SST не полностью помещается в памяти, для случайного чтения требуется минимальное количество операций поиска на диске.

Но для поддержки быстрой записи этого мало. В отличие от записи на диск запись в набор данных, размещенный в памяти, тривиальна, поскольку нужно просто поменять указатели. Таблица, хранящаяся в памяти, может принимать за-

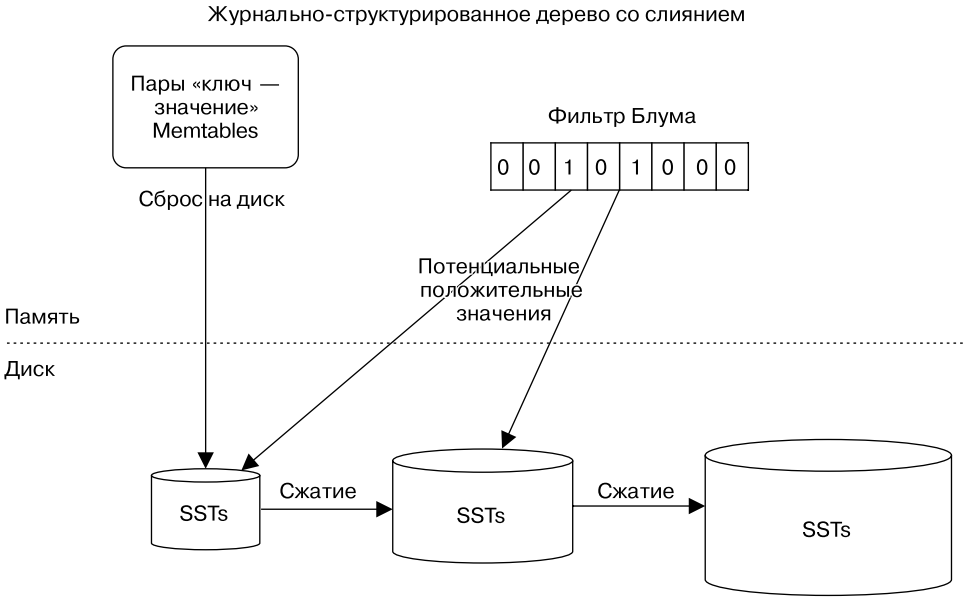


Рис. 10.2. Журнально-структурированное дерево со слиянием с использованием фильтра Блума

писи и оставаться сбалансированной. Такие таблицы также называются *memtable*. Таблицы memtable могут использоваться в качестве отправной точки запроса на чтение, перед тем как перейти к самой свежей версии SST на диске, потом следующей после нее и так далее, пока данные не будут найдены. После достижения определенного порогового значения, которое может быть выражено как время, количество транзакций или размер, memtable будет отсортирована и записана на диск.

При удалении данных, которые уже сохранены в SST, должны быть записаны значения логического удаления. Их еще называют *затирающими значениями* (tombstone). Периодически SST-таблицы объединяются, что позволяет устранить затирающие значения и сэкономить место. При вводе-выводе этот процесс слияния и сжатия бывает очень интенсивным и часто требует значительно больше памяти, чем фактический рабочий набор данных. Пока рабочие группы не привыкнут к этим новым моделям емкости, это может влиять на достижимость целевого уровня качества обслуживания (Service-Level Objectives, SLO).

Существует возможность потери данных, которую необходимо учитывать в сценариях сбоев. Пока таблица memtable не записана на диск, она уязвима для сбоев. Неудивительно, что в механизме хранения SST предусмотрены решения, подобные тем, что используются в системах на основе B-дерева, в том числе журналы событий, журналы повторных операций и регистрация записи с упреждением.

Фильтры Блума

Возможно, вы считаете, что поиск в memtable и в большом количестве таблиц SSTable может быть дорогим и медленным, если окажется, что не существует ключа записи. В этом случае вы правы! Для решения данной проблемы используется *фильтр Блума* — структура данных, которую можно применять для того, чтобы определить, существует ли ключ записи в наборе, которым в данном случае является SSTable.

В хранилищах данных, таких как Cassandra, фильтры Блума позволяют определить, существует ли запрошенный ключ записи и в какой SSTable он может находиться. Фильтр Блума предназначен для ускорения работы и поэтому иногда может приводить к ложным срабатываниям. Но в целом его использование значительно сокращает количество операций ввода-вывода при чтении. И наоборот, если фильтр Блума сообщает, что ключ записи не существует в данной SSTable, то это так и есть. Фильтры Блума обновляются при записи таблиц memtable на диск. Чем больше памяти можно выделить для фильтра, тем меньше вероятность ложного срабатывания.

Реализации

Существует несколько хранилищ данных, в которых в качестве механизма хранения используется структура LSM совместно с SSTables:

- Apache Cassandra;
- Google Bigtable;
- HBase;
- LevelDB;
- Lucene;
- Riak;
- RocksDB;
- WiredTiger.

Детали реализации для каждого хранилища данных свои, но распространение и постоянное совершенствование этого механизма хранения превратили его в важный вариант реализации хранилища для любой организации, которая занимается анализом больших наборов данных.

Перечисляя и изучая различные структуры хранения данных, мы неоднократно упоминали журналы и отмечали, что они критически важны для обеспечения надежности данных в случае сбоев. Подробно мы их рассмотрели в главе 7. Журналы также важны для репликации данных в распределенных хранилищах. Давайте подробнее рассмотрим журналы и их использование при репликации.

Индексирование

Мы уже обсуждали одну из наиболее распространенных структур индексации — B-дерево. Таблицы SST также внутренне индексируются. Существуют и другие структуры индекса, которые применяются в мире баз данных.

Хеш-индексы

Одна из самых простых реализаций индекса — *хеш-карта*. Это набор блоков, в которых содержатся результаты хеш-функции, примененной к ключу. Этот хеш указывает на место, где можно найти данные записи. Применение хеш-карты имеет смысл только для поиска по одному ключу, так как сканирование диапазона выходит чрезмерно затратным. Кроме того, для обеспечения хорошей производительности хеш должен полностью помещаться в памяти. При выполнении этих условий хеш-карты обеспечивают отличный доступ в конкретных случаях использования, в которых они применимы.

Битовые индексы

Битовый индекс хранит данные в виде битовых массивов (битовых карт). Поиск по индексу проходит путем выполнения побитовых логических операций над битовыми картами. В B-деревьях индекс лучше всего работает со значениями, которые повторяются не очень часто. Этот показатель также называется *высокой кардинальностью*. Битовый индекс работает намного эффективнее, если индексируется небольшое количество значений.

Модификации B-деревьев

Есть множество модификаций традиционного индекса в виде B-дерева. Они часто предназначены для очень специфичных вариантов использования, включая следующие.

- *Индекс по функции*. Индекс, основанный на результатах выполнения функции, примененной к индексу.
- *Обратный индекс*. Значения индексируются от конца к началу для обратной сортировки.
- *Кластерный индекс*. Этот индекс требует, чтобы записи в таблице были физически сохранены в индексированной последовательности с целью оптимизации доступа ввода/вывода. Конечные узлы кластерного индекса содержат страницы данных.

- *Пространственный индекс.* Существует несколько механизмов индексации пространственных данных. Стандартные индексы не позволяют эффективно обрабатывать пространственные запросы.
- *Поисковый индекс.* Такие индексы позволяют искать подмножества данных в столбцах. Большинство индексов не могут выполнять поиск внутри индексированного значения. Однако есть несколько типов индексов, выполняющих эту операцию, и для нее созданы целые хранилища данных, такие как ElasticSearch.

Каждое хранилище данных имеет собственный набор доступных специализированных индексов, часто предназначенных для оптимизации типичных вариантов использования в этом хранилище данных.

Индексы чрезвычайно важны для быстрого доступа к подмножествам данных. При выборе новейшего хранилища данных важно понимать его ограничения с точки зрения индексов, такие как возможность иметь несколько индексов или количество столбцов, которые можно проиндексировать, или даже то, каким образом эти индексы поддерживаются в фоновом режиме.

Журналы и базы данных

Поначалу журналы для баз данных служили обеспечению устойчивости. Они эволюционировали до механизма, используемого при репликации данных с первичного сервера на сервер реплики в целях доступности и масштабируемости. В какой-то момент были созданы целые сервисы для использования журналов с целью переноса данных между различными БД при наличии уровня преобразования между ними. Затем журналы превратились в полноценную систему обмена сообщениями, где журнальные записи играют роль событий, которые сервис подписки может использовать для выполнения отдельных видов работ для подписанных сервисов.

Имея множество вариантов применения журналов, мы хотели бы сосредоточиться в этой главе только на репликации. Обсудив, как данные могут храниться и индексироваться на локальном сервере, теперь перейдем к тому, как эти данные могут распределяться на других серверах.

Репликация данных

На протяжении всей этой книги мы предполагали, что вы будете работать главным образом с распределенными хранилищами данных. Это означает, что должен быть способ перемещения данных, записанных на одном узле, на другие узлы. На

эту тему написано много книг[1], поэтому мы сосредоточимся на хорошо известных, проверенных примерах, а не на теоретических исследованиях. Наша цель состоит в том, чтобы вы как DBRE-специалист и разработчики, которых вы поддерживаете, могли изучить предлагаемые методы репликации и понять, как они работают. Понимание достоинств и недостатков, а также знание паттернов и антипаттернов применения, связанных с вариантами репликации, крайне важно для DBRE-специалиста, архитектора, инженера-программиста или специалиста по эксплуатации, если они хотят хорошо выполнять свою работу.

Стоит перечислить несколько общих различий в архитектурах репликации. Обсуждая репликацию, мы будем называть ведущими те узлы, которые принимают записи от приложений, а ведомыми — узлы, которые получают реплицированные события для применения их к наборам данных. Наконец, мы будем называть узлами считывания те узлы, с которых приложения считывают данные.

- *Один ведущий узел*. Данные всегда отправляются на один конкретный ведущий узел.
- *Несколько ведущих узлов*. Может существовать несколько узлов, играющих роль ведущих, и каждый ведущий узел должен сохранять данные в кластере.
- *Отсутствие ведущего узла*. Ожидается, что все узлы могут принимать данные при записи.

Мы начнем с самого простого метода репликации — с одним ведущим узлом — и будем продвигаться к более сложным.

Репликация с одним ведущим узлом

Как следует из заголовка, в этой модели репликации все операции записи направляются на один ведущий узел и реплицируются оттуда. Таким образом, один из *N* узлов выбирается в качестве ведущего, а остальные являются репликами. Данные поступают от ведущего узла. Этот метод очень популярен благодаря своей простоте и позволяет гарантировать следующее:

- отсутствие конфликтов согласованности, потому что все операции записи исходят из одного узла;
- если предположить, что все операции детерминированные, они приведут к одинаковым результатам на каждом узле.

[1] *Helal A. A.* Replication Techniques in Distributed Systems. — Advances in Database Systems, 1996.

У этого метода есть ряд модификаций. Например, один ведущий узел реплицируется на несколько ретрансляционных реплик, которые, в свою очередь, имеют собственные реплики. Тем не менее существует один ведущий узел, который выполняет операции записи, что является ключевым признаком данной архитектуры. Есть несколько подходов к репликации при одном ведущем узле. Каждый подход обуславливает определенный уровень согласованности, задержки и доступности. Таким образом, выбор соответствующего варианта зависит от конкретных приложений и от того, как они используют кластеры баз данных.

Модели репликации

При репликации данных в режиме с одним ведущим узлом можно использовать три разные модели.

- *Асинхронная репликация* — оптимизация задержки за счет устойчивости.
- *Синхронная репликация* — оптимизация устойчивости за счет задержки.
- *Полусинхронная репликация* — компромисс между задержкой и устойчивостью.

В моделях *асинхронной репликации* транзакция записывается в журнал ведущего узла, а затем записывается на узел и сбрасывается на диск. Отдельный процесс отвечает за доставку этих журналов подписчикам, где они сразу используются. В моделях асинхронной репликации всегда существует некоторое отставание между тем, что передается ведущему узлу, и тем, что передается ведомым узлом. Кроме того, нет гарантии, что момент записи у одного подписчика совпадет со временем записи у остальных. На практике временной промежуток между моментами записи может быть слишком мал, чтобы его заметить. Не менее часто встречаются кластеры, использующие асинхронную репликацию, для которой временной интервал между передачей ведущим и ведомым узлам составляет несколько секунд, минут или даже часов.

В моделях *синхронной репликации* транзакция, которая записывается в журнал ведущего узла, немедленно передается по сети его подписчикам. Ведущий узел *не будет фиксировать транзакцию*, пока его подписчики не подтвердят, что они выполнили запись. Таким образом гарантируется, что все узлы кластера находятся в той же точке фиксации транзакции. Это означает, что все операции чтения будут согласованы, независимо от того, с какого узла они приходят, и любой узел может стать ведущим без риска потери данных при сбое на текущем ведущем узле. Однако при этом в случае сетевой задержки или повреждения узла может возникнуть задержка записи для транзакции на ведущем узле.

Поскольку синхронная репликация может оказать значительное влияние на задержку, особенно при наличии большого количества узлов, то в качестве компромисса может быть использована *полусинхронная* репликация. В данном алгоритме

достаточно, чтобы только один узел подтвердил ведущему узлу, что он выполнил запись. При этом снижается риск влияния задержки, если один или несколько узлов окажутся в ухудшенных состояниях, и гарантируется, что по крайней мере два узла в кластере находятся в одной и той же точке фиксации. В этом режиме нет гарантии, что все узлы в кластере будут возвращать одни и те же данные, независимо от того, с какого узла считывания выполняется чтение. Зато гарантируется, что при необходимости всегда можно перевести хотя бы один узел кластера в состояние ведущего без потери данных.

Форматы журнала репликации

Для репликации с одним ведущим узлом необходимо использовать журнал транзакций. Существует несколько подходов к реализации этих журналов. У каждого из них есть свои плюсы и минусы, и во многих хранилищах данных можно использовать несколько вариантов, чтобы можно было выбрать, какой из них подходит вам лучше всего. Давайте рассмотрим их здесь.

Логические журналы. При логической репликации фактический SQL-оператор или оператор записи данных, используемый для выполнения записи, записывается и отправляется от ведущего узла к подписчикам. Это означает, что для каждого подписчика будет выполняться весь оператор.

Преимущества

- Оператор может выполнять сотни и тысячи операций записи, вместо того чтобы передавать множество данных. Оператор обычно занимает намного меньше места. Это решение может быть оптимальным при репликации через центры обработки данных, где пропускная способность сети ограничена.
- Данный подход переносим. В результате выполнения большинства SQL-операторов получаются одинаковые результаты даже в разных версиях баз данных. Это позволяет обновлять подписчиков перед обновлением ведущих узлов. Это критически важная составляющая высокодоступных подходов к обновлению в производственной среде. Без обратносовместимой репликации для обновления версий могут быть значительные простои при обновлении всего кластера.
- Поскольку файлы журналов содержат целые операторы, их также можно использовать для аудита и интеграции данных.

Недостатки

- Для выполнения оператора может потребоваться значительное время, особенно если в нем используются функции агрегирования и вычисления для выбранного набора данных, чтобы определить, какие именно данные будут записаны. Выполнение оператора может занять гораздо больше времени, чем простое

изменение записей или битов на диске. Это может вызвать задержку репликации при выполнении сериализованных прикладных процессов.

- Некоторые операторы могут оказаться недетерминированными и создавать разные результаты для одного набора данных при выполнении на разных узлах.

Примером такого подхода является логическая репликация MySQL.

> **ДЕТЕРМИНИРОВАННЫЕ ТРАНЗАКЦИИ**
>
> Под *детерминированными* мы понимаем такие транзакции, при которых результат выполнения оператора не зависит от времени и от внешних факторов. Если оператор выполняется для того же набора данных и в той же последовательности, то он, независимо от узла, должен давать один и тот же результат. Примеры недетерминированных операторов включают в себя использование локальных функций времени, таких как `now()` или `sysdate()`, или же функций, в которых используется случайная последовательность, например `order by rand()`.
>
> Аналогично наличие хранимого кода, такого как пользовательские функции, хранимые процедуры и триггеры, может привести к тому, что оператор будет недетерминированным и, следовательно, небезопасным для логической репликации.

Журналы упреждающей записи. Журнал упреждающей записи (Write-Ahead Log, WAL), также известный как журнал повтора, содержит сведения о серии событий, каждое из которых сопоставляется с транзакцией или записью. В журнале перечислены все байты, необходимые для применения транзакции к диску. В таких системах, как PostgreSQL, где используется этот метод, один и тот же журнал отправляется непосредственно всем приложениям-получателям, которые и выполняют необходимые изменения на дисках.

Достоинства

- Репликация выполняется очень быстро, поскольку синтаксический анализ и сам оператор уже выполнены. Осталось только применить изменения к диску.
- Репликация не подвержена риску воздействия недетерминированного SQL-оператора.

Недостатки

- Если интенсивность записи высока, то может потребоваться и значительная пропускная способность.
- Ограниченная переносимость, потому что формат тесно связан с механизмом хранения базы данных. Это может усложнить выполнение непрерывных обновлений, которые позволяют свести к минимуму время простоя.
- Не очень удобен для аудита.

В методе WAL часто применяются те же журналы, что были созданы для обеспечения устойчивости, и для репликации просто используется процесс доставки журналов. Это обеспечивает эффективность такого формата, но также лишает его мобильности и гибкости.

Построчная репликация. При построчной репликации (также называемой логической) операции записи заносятся в журналы репликации ведущего узла как события, указывающие на то, как изменяются отдельные строки таблицы. Указываются столбцы с новыми данными; столбцы с измененной информацией приводятся как до, так и после изменений, а также отмечаются удаленные строки. В репликах эти данные применяются не для выполнения исходного оператора, а для непосредственного изменения строк.

Достоинства

- Данный метод не подвержен риску воздействия недетерминированных SQL-операторов.
- Это компромисс по скорости между двумя предыдущими алгоритмами. Все еще требуется перевод с логического на физический уровень, но не приходится выполнять целые операторы.
- Это компромисс по переносимости между двумя предыдущими алгоритмами. Не очень удобочитаемый, но может использоваться для интеграции и проверки.

Недостатки

- В средах с интенсивной записью этот метод может требовать значительной пропускной способности.
- Не очень поддается аудиту.

Данный метод также называется *отслеживанием измененных данных* (Change Data Capture, CDC). Он применяется в SQL Server и MySQL, а также в средах хранилищ данных.

Репликация на уровне блоков. До сих пор мы говорили о методах репликации с использованием собственных механизмов базы данных. В отличие от этого репликация блочных устройств является внешним подходом. В качестве соответствующей реализации чаще всего используется распределенное реплицированное блочное устройство (Distributed Replicated Block Device, DRBD) для Linux. DRBD функционирует на уровне блочных устройств и распространяет записи не только на локальном блочном устройстве, но и на реплицированном, расположенном на другом узле.

Репликация на уровне блоков является синхронной и позволяет избежать значительных издержек при репликации операций записи. Однако на вторичном узле не может быть работающего экземпляра базы данных. Таким образом, в случае

аварийного переключения должен быть запущен новый экземпляр базы данных. Если предыдущее ведущее устройство вышло из строя без отключения чистой базы данных, то этот новый экземпляр базы должен будет выполнить восстановление так же, как если бы он был перезапущен на том же узле.

Итак, у нас есть репликация на уровне блоков — синхронная репликация с очень малыми задержками, но при этом теряется возможность использовать реплики для масштабируемости или распределения рабочей нагрузки. К счастью, метод внешней репликации, такой как репликация на уровне блоков, может применяться в сочетании с собственной репликацией, такой как логическая репликация или репликация на уровне строк. Так можно сочетать достоинства репликации с нулевой потерей данных с гибкостью, обеспечиваемой асинхронной репликацией.

Другие методы. Существуют и другие методы репликации, отделенные от журналов базы данных. В процессе извлечения, преобразования и загрузки (Extraction, Transform and Load, ETL), используемых для перемещения данных между сервисами, часто ищутся признаки новых или измененных строк, такие как идентификаторы или временные метки. Затем согласно этим признакам извлекаются данные для загрузки в другом месте.

Триггеры, которые находятся в таблицах, также позволяют загружать таблицу с изменениями для прослушивания во внешнем процессе. Эти триггеры могут просто содержать список идентификаторов для изменений или же предоставлять полную информацию об отслеживании измененных данных, как это делается при репликации на основе строк.

При выборе вариантов репликации вам потребуется подобрать сочетание параметров в зависимости от исходного хранилища данных, хранилища-приемника и инфраструктуры, существующей между этими двумя хранилищами. Мы обсудим это подробнее в следующем подразделе, посвященном использованию репликации.

Использование репликации с одним ведущим узлом

На текущем этапе развития хранилищ данных репликация чаще всего является не сколько требованием, сколько необязательной возможностью. Но и сегодня существует множество причин реализовать репликацию. Эти причины могут повлиять на архитектуру и конфигурацию. В архитектурах с одним ведущим узлом это главным образом доступность, масштабируемость, локальность и переносимость.

Доступность. Само собой разумеется, что в случае сбоя на ведущем узле базы данных требуется самый быстрый вариант восстановления, на который можно перевести трафик приложения. Наличие рабочей базы данных с полностью обновленной копией данных гораздо предпочтительнее, чем резервное копирование, при котором

необходимо сначала восстановить данные, а затем обновить их до точки отказа. Это означает, что требования к среднему времени восстановления (Mean Time to Recover, MTTR) и к потере данных должны оставаться главными приоритетами при выборе метода репликации. Синхронная и полусинхронная репликация обеспечивает наилучшие варианты без потери данных при низком MTTR, но они влияют на задержку. Достичь идеального сочетания низкого MTTR и низкой задержки с отсутствием потерь данных с помощью одной только репликации невозможно. Нужна некоторая внешняя поддержка, такая как система обмена сообщениями. В такой системе можно делать записи, дополняющие запись в хранилище данных, чтобы обеспечить восстановление данных, которые могут быть потеряны при отказе ведущего узла в асинхронно реплицируемой среде.

Масштабируемость. Один ведущий узел создает ограничение на операции ввода-вывода при записи, но узлы-последователи позволяют считывать данные и обеспечивать масштабирование, соответствующее количеству выполненных операций чтения. Для приложений с интенсивным чтением и относительно небольшим количеством операций записи несколько таких реплик создают возможность для создания большей емкости в кластере. Эта емкость ограничена, поскольку издержки на репликацию не допускают линейной масштабируемости. Тем не менее это создает возможность для увеличения пропускной способности. Для поддержки масштабируемости данные на репликах должны быть достаточно свежими, чтобы соответствовать бизнес-требованиям. Для некоторых организаций задержка репликации, присущая системам с асинхронной репликацией, является приемлемой. Однако при других требованиях синхронная репликация абсолютно необходима, независимо от влияния задержки записи.

Локальность. Репликация также является способом хранения наборов данных в различных местах, расположенных ближе к потребителям, что позволяет свести к минимуму задержку. Если у вас есть клиенты, проживающие в разных странах или даже на разных континентах, влияние удаленных запросов может оказаться значительным. Переносить целиком большие наборы данных не всегда возможно, но постепенное применение изменений позволяет поддерживать эти наборы в актуальном состоянии. Как уже упоминалось, для репликации на большие расстояния в сетях с ограниченной пропускной способностью часто требуется логическая репликация — если для управления записями на основе строк или WAL сжатия недостаточно. Современные сети и алгоритмы сжатия часто смягчают эту проблему. Кроме того, вследствие задержек полусинхронные и синхронные алгоритмы, как правило, невозможно выполнять на большие расстояния, что приводит к выбору асинхронного способа репликации.

Переносимость. В других хранилищах данных может поддерживаться ряд возможностей для данных, находящихся в ведущем узле. Журналы репликации можно использовать для записи в хранилища данных в качестве событий для потребителей в конвейере данных или для преобразования в другие хранилища

с более подходящими шаблонами запросов и индексации. Использование тех же потоков репликации, что и у имеющихся реплик, для повышения доступности и масштабирования гарантирует, что наборы данных, передаваемые от ведущего узла, будут одинаковыми. При этом более индивидуальные решения, такие как ETL на основе запросов, и подходы на основе триггеров, обеспечивают отбор соответствующих подмножеств данных, а не передачу всего потока транзакций, выходящих из журналов репликации. Эти задачи также часто не слишком требовательны к актуальности данных, что позволяет использовать подходы с меньшим влиянием на задержку.

Исходя из этих потребностей, вы и команда разработчиков должны иметь возможность выбрать один или несколько вариантов репликации. Независимо от того, какой выбор вы сделаете, в этих реплицированных средах может возникнуть ряд проблем.

Проблемы репликации с одним ведущим узлом

В любой реплицируемой среде есть множество проблемных мест. Несмотря на то что среда репликации с одним ведущим узлом является самой простой из реплицируемых сред, это никоим образом не означает, что в ней все легко и просто. В этом подразделе мы рассмотрим самые распространенные из этих проблем.

Построение реплик. В случае больших наборов данных их переносимость может быть значительно снижена. Мы уже обсуждали это в главе 7. По мере увеличения размера набора данных MTTR также увеличивается, что может привести к необходимости в большем количестве реплик или новой стратегии резервного копирования, которая бы поддерживала MTTR в допустимых пределах. Другой вариант — уменьшить размер набора данных в группе серверов, разбив его на несколько меньших наборов. Это также называется *сегментированием* (sharding). Подробнее мы обсудим это в главе 12.

Синхронизация реплик. Создание реплики — лишь первый шаг в реплицируемой среде. При использовании асинхронной репликации сохранение реплики оказывается проблемой в том случае, если среда характеризуется частыми или обширными изменениями набора данных. Как уже обсуждалось в пункте «Форматы журнала репликации» в этом разделе ранее, изменения должны журналироваться, журналы — передаваться, после чего должны применяться изменения.

Реляционные базы данных по умолчанию обычно переводят записи в линеаризованную последовательность транзакций, которая должна строго соблюдаться для обеспечения согласованности набора данных между репликами и ведущими узлами. Как правило, это приводит к необходимости сериализованных процессов, вносящих изменения в реплики одно за другим. Эти сериализованные процессы

внесения изменений часто по ряду причин не успевают за изменениями ведущего узла и отстают от него. В число этих причин входят следующие.

- Отсутствие конкурентности и параллелизма по сравнению с ведущим узлом. На репликах ресурсы ввода/вывода часто расходуются напрасно.
- Если на репликах трафик чтения не является основным, то блоки для чтения в транзакциях не находятся в памяти.
- Когда операции записи выполняются ведущим узлом, а операции чтения — репликами, конкурентность трафика чтения может повлиять на задержку записи в репликах.

Независимо от причины, такой эффект часто называют *задержкой реплики*. В некоторых средах задержка реплики может быть редкой и преходящей проблемой, которая обычно разрешается сама собой, не влияя на SLO. В других средах эти проблемы становятся повсеместными и могут привести к невозможности использования реплик для их первоначального назначения. Если это происходит, значит, рабочая нагрузка для хранилища данных стала слишком большой и ее необходимо перераспределить с помощью одного или нескольких методов. Если вкратце, то это следующие методы (подробнее будут рассмотрены в главе 12).

- *Краткосрочного действия*. Увеличьте емкость кластера, чтобы текущая рабочая нагрузка соответствовала емкости кластера.
- *Среднесрочного действия*. Разбейте функции базы данных по кластерам, чтобы рабочие нагрузки гарантированно не превышали емкость кластера. Этот метод также известен как *функциональное разбиение* или *функциональное сегментирование*.
- *Долгосрочного действия*.
  - Разбейте набор данных на несколько кластеров, что позволит удерживать рабочие нагрузки в пределах емкости кластера. Этот метод также известен как *разделение* или *сегментирование набора данных*.
  - Выберите систему управления базами данных, у которой требования к хранению, согласованности и устойчивости больше подходят для вашей рабочей нагрузки и SLO и у которой не будет проблем с масштабированием.

Как видно из описанного выбора, ни один из этих методов не будет работать при дальнейшем увеличении системы. Другими словами, они не масштабируются линейно по мере повышения нагрузки. Некоторые, такие как увеличение пропускной способности функциональных разделов, имеют более короткое время жизни, чем другие, например разделение наборов данных. Но даже разделение наборов данных в итоге достигнет своих пределов. Это означает, что необходимо учитывать и другие нюансы, чтобы гарантировать, что границы никогда не достигнут той точки, после которой решение перестанет быть применимым.

Если имеет место задержка репликации и необходимо смягчить ее воздействие, пока не будет внедрено более долгосрочное решение, можно использовать, в частности, следующие краткосрочные тактики:

- предварительную загрузку активных наборов данных реплики в память для уменьшения дискового ввода-вывода;
- снижение устойчивости реплик для уменьшения задержки записи;
- распараллеливание репликаций по схемам в базе данных (при отсутствии транзакций, действующих между схемами).

Это все краткосрочные методы, которые позволяют дать передышку, но требуют компромиссов с точки зрения неустойчивости, высоких затрат на обслуживание и потенциальных проблем с данными, поэтому их необходимо тщательно исследовать и применять только при крайней необходимости.

Восстановление после сбоя единого ведущего узла. Одной из величайших ценностей репликации является наличие других наборов данных, на которые можно переключиться, чтобы использовать их в качестве ведущих узлов в случае сбоя или при необходимости увести трафик с исходного ведущего узла. Однако это непростая операция, которая делится на ряд этапов. Ниже перечислены эти этапы в случае запланированного перехода на другой ресурс.

1. Проводится идентификация реплики, которую вы намерены передать новому ведущему узлу.
2. В зависимости от топологии можно выполнить предварительную частичную реконфигурацию кластера, чтобы переместить все реплики для репликации с кандидата на роль ведущего узла.
3. Если используется асинхронная репликация, то нужно сделать паузу в трафике приложения, чтобы кандидат на роль ведущего узла мог наверстать упущенное.
4. Следует изменить конфигурацию всех клиентов приложений и указать там новую реплику.

В случае чистого, запланированного переключения все это может показаться довольно тривиальным, если вы разработали эффективные сценарии и автоматизировали некоторые этапы. Однако если полагаться на такие механизмы восстановления в сценариях сбоев, это может привести ко многим проблемам. Незапланированное аварийное переключение может выглядеть примерно так.

1. Экземпляр базы данных ведущего узла перестает отвечать на запросы.
2. Система мониторинга контрольных сигналов пытается подключиться к ведущему узлу базы данных.
3. После 30 секунд отсутствия контрольных сигналов запускается алгоритм восстановления после сбоя.

4. Алгоритм восстановления выполняет следующие действия:
   1) находит реплику с последним временем сохранения и назначает ее кандидатом на роль основной;
   2) переподключает другие реплики к кандидату на роль ведущего узла в соответствующей точке потока журнала;
   3) следит, чтобы были подключены все реплики кластера;
   4) изменяет и сохраняет конфигурацию приложения через файл или сервис;
   5) запускает процесс перестройки новой реплики.

Здесь есть несколько переломных моментов. Мы обсудим это далее в главе 12.

Несмотря на возможные проблемы, репликация остается одной из наиболее часто используемых функций баз данных и, следовательно, становится важной частью инфраструктуры БД. Это означает, что ее необходимо включить в инфраструктуру для обеспечения надежности.

Мониторинг репликации с одним ведущим узлом

Эффективное управление репликацией требует эффективного мониторинга и оперативного контроля. Существует ряд метрик, которые необходимо собирать и интерпретировать, чтобы реплики были гарантированно эффективны для обеспечения SLO организации. К критически важным показателям, которые нужно отслеживать, относятся:

- задержка репликации;
- влияние простоев на записи;
- доступность реплики;
- согласованность репликации;
- операционные процессы.

Мы уже упоминали их в главе 4, но здесь стоит обсудить их еще раз.

Задержка репликации и простои. Чтобы понять суть потоков репликации, необходимо определить, какое относительное время требуется для выполнения операций репликации. В асинхронных средах это означает количество времени, которое прошло между операцией, выполняемой на главном устройстве, и моментом, когда эта запись была применена к реплике. Это время может сильно различаться в разные моменты, но эти данные имеют решающее значение. Есть несколько способов, которыми его можно измерить.

Как в любой распределенной системе, эти измерения на определенном уровне зависят от времени локальной машины. Если системные часы расходятся

с показателями протокола сетевого времени (Network Time Protocol, NTP), то информация может быть искажена. Мы просто не можем полагаться на локальное время на двух машинах и предполагать, что они синхронизированы. Для большинства распределенных баз данных, использующих асинхронную репликацию, это не проблема. Данные времени практически совпадают, и этого обычно достаточно. Но даже в таком случае, если эксперты будут учитывать, что время очень относительное понятие для каждого узла, это поможет решить некоторые проблемные вопросы.

Один из распространенных подходов к измерению времени между операцией вставки на ведущем узле и операцией вставки на реплике состоит в том, чтобы вставить строку с контрольным сообщением и затем измерить, через какое время она появится в реплике. Например, если вставить данные в 12:00:00 и, регулярно опрашивая реплику, увидеть, что она не получила это значение, можно предположить, что репликация остановлена. Если выполнить запрос в 12:01:00 и обнаружить, что данные за 11:59:00 существуют, но данных за 12:00:00 и более позднее время нет, можно сделать вывод, что репликация, сделанная в 12:01:00, отстает на 1 секунду. В какой-то момент эта строка появится, и тогда можно будет создать следующую строку, чтобы измерить, насколько сильно база данных отстает в данный момент.

В случае полусинхронной или полностью синхронной репликации вы захотите узнать влияние этих конфигураций на операции записи. Оно может и будет измеряться как часть общих показателей задержки, но также можно измерить время, которое занимает передача по сети от ведущего узла к реплике, — это и будет стоимость передачи по сети синхронных операций записи.

Ниже приведены критически важные показатели, которые необходимо измерять:

- задержка во времени между ведущим узлом и репликой при асинхронной репликации;
- сетевая задержка между ведущим узлом и репликами;
- влияние задержки операций записи при синхронной репликации.

Эти показатели сложно переоценить для любого сервиса. Прокси-инфраструктуры могут использовать информацию о задержке репликации для проверки того, какие реплики базы данных достаточно актуальны, чтобы перевести на них трафик чтения в производственной среде. На уровне прокси, на котором можно отключать узлы, это не только позволяет гарантировать, что с устаревших реплик не будут выполняться операции чтения, но и дает возможность отстающим репликам наверстать упущенное, не нагружая их трафиком чтения. Конечно, в этом алгоритме необходимо учитывать, что произойдет, если будут задерживаться все реплики. Будет ли сервис работать напрямую от ведущего узла? Будет ли отключен интерфейс, пока не обновится достаточное количество реплик? Будет ли система переведена

в режим «только для чтения»? Все эти варианты могут быть эффективными, если они запланированы.

Кроме того, разработчики могут использовать информацию о задержках и простоях репликации для устранения проблем согласованности данных, снижения производительности и других ситуаций, которые могут возникнуть из-за задержек репликации.

Доступность и емкость репликации. При работе с хранилищем данных, таким как Cassandra, в котором данные распределяются синхронно, на основе коэффициента репликации, необходимо отслеживать и знать количество доступных копий, чтобы удовлетворить требования по кворуму операций чтения. Например, предположим, что у нас есть кластер с коэффициентом репликации 3. Это означает, что операция записи должна реплицироваться на три узла. Наше приложение требует, чтобы 2/3 узлов с этими данными могли возвращать результаты во время запроса. Это означает, что, если произойдет две ошибки, мы не сможем удовлетворить запросы приложений. Мониторинг доступности реплик с упреждением позволяет узнать, когда есть риск сбоя.

Аналогично даже в средах без требований к коэффициенту репликации и кворуму кластеры баз данных все равно разрабатываются с учетом того, сколько узлов должно быть доступно для достижения SLO. Мониторинг того, как размер кластера соответствует этим ожиданиям, имеет решающее значение.

Наконец, важно понимать, когда репликация полностью прекратилась. Мониторинг задержки репликации с помощью контрольных сообщений способен информировать об отставании репликации, однако он не предупредит вас, когда что-то произойдет и поток репликации прервется. Репликация может прерваться по таким причинам, как:

- разбиение сети на разделы;
- невозможность выполнения операторов на языке манипулирования данными (Data Manipulation Language, DML) при логической репликации, в частности:
  - несоответствие схемы;
  - недетерминированный SQL, вызывающий дрейф набора данных, который нарушает ограничение;
  - операции записи, случайно попавшие в реплику, что вызывает дрейф набора данных;
- изменения привилегий и других параметров безопасности;
- недостаток места для хранения данных в реплике;
- повреждение данных в реплике.

Ниже приведены примеры показателей, которые стоит отслеживать:

- реальное количество доступных копий данных в сравнении с ожидаемым количеством;
- прекращение репликации, требующее восстановления реплик;
- сетевые показатели между ведущими узлами и репликами;
- журналы изменений для схем базы данных и пользователей/привилегий;
- показатели того, сколько памяти используется журналами репликации;
- журналы базы данных, которые предоставляют более подробную информацию о проблемах, таких как ошибки репликации и повреждения.

На основе этой информации средства автоматизации могут использовать показатели доступности реплик для развертывания новых реплик в условиях нехватки ресурсов. Кроме того, операторы могут быстрее находить основные причины сбоев, чтобы определить, следует ли выполнить восстановление, или просто заменить реплику, или же необходимо устранить более системную проблему.

Согласованность репликации. Как уже обсуждалось ранее, возможны сценарии, которые могут привести к тому, что наборы данных ведущего узла и реплики окажутся несогласованными. Иногда это приводит к сбою события репликации на этапе применения, что можно заметить по прерыванию репликации. Хуже, когда искажение данных происходит незаметно и вы обнаруживаете его спустя достаточно долгое время.

Как вы, вероятно, помните, в главе 7 мы обсуждали важность процесса валидации для сохранения согласованности наборов данных с бизнес-правилами и ограничениями. Можно использовать аналогичный процесс, чтобы гарантировать идентичность данных в разных репликах. Подобно процессам валидации данных на предмет согласованности, такие процедуры зачастую довольно непросты и недешевы с точки зрения ресурсов. Это означает, что необходимо избирательно определить, какие объекты данных и как часто следует проверять.

Управлять данными, предназначенными только для добавления, такими как SST или даже таблицы в структурах B-дерева, используемые только для вставки, проще, поскольку можно создавать контрольные суммы для набора строк на основе первичного ключа или диапазона дат и сравнивать эти контрольные суммы по репликам. Пока эта проверка выполняется достаточно часто, чтобы не отставать от репликации, вы можете быть относительно уверены, что данные согласованны.

Для данных, которые допускают изменения, это может оказаться более сложным. Один из подходов состоит в том, чтобы выполнить функцию хеширования для данных на уровне БД и сохранить ее результаты после завершения транзакции в приложении. Хеш, включенный в поток репликации, будет создавать идентичные значения в каждой реплике, если данные реплицируются надлежащим образом.

Если же этого не произошло, то хеши будут разными. Асинхронное задание, которое сравнивает хеши с последними транзакциями, предупредит вас о разных значениях.

Это лишь несколько способов отслеживания согласованности репликации. Создание шаблонов для разработчиков программного обеспечения (SWE), а также системы классификации объектов данных, чтобы помочь им определить, нужно ли выделить место для таблицы в процессе валидации, позволит вам не задействовать слишком много ценных ресурсов. Выборка или даже просто выполнение валидации для последних временных окон также могут оказаться эффективными, в зависимости от типа хранимых данных[1].

Операционные процессы. Наконец, важно мониторить время и ресурсы, необходимые для выполнения операционных процессов, критически важных для репликации. Со временем, по мере роста наборов данных и появления конкурентности, эти процессы могут стать более обременительными. Если превысить определенные пороговые значения, то появляется риск того, что нарушится актуальность репликации или вы не сможете обеспечить надлежащее количество реплик в сети в любое время для поддержания трафика. Вот некоторые из этих метрик:

- размер набора данных;
- продолжительность резервного копирования;
- продолжительность восстановления реплики;
- пропускная способность сети, требуемая для резервного копирования и восстановления;
- время, необходимое для синхронизации после восстановления;
- нагрузка на узлы в производственной среде во время резервного копирования.

Передавая события с соответствующими показателями каждый раз, когда происходит резервное копирование, восстановление или синхронизация, можно формировать отчеты для оценки и, возможно, прогнозирования того, когда изменение набора данных и появление конкурентности приведут к тому, что станет невозможно задействовать существующие рабочие процессы. Можно также использовать некоторые простейшие оценки для прогнозирования того, как продолжительность или потребление ресурсов могут измениться в зависимости от изменений размера набора данных или конкурентности.

Регулярные проверки и тесты могут помочь персоналу спрогнозировать, когда их рабочие процессы перестанут масштабироваться. Это позволит вовремя увеличить емкость, перепроектировать системы или процессы либо перебалансировать распределение наборов данных, чтобы все так же обеспечивать доступность и время отклика согласно SLO.

---

[1] См. статью: *Anderson T. A.* Replication, Consistency and Practicality: Are These Mutually Exclusive? https://www.researchgate.net/publication/221215000_Replication_Consistency_and_Practicality_Are_These_Mutually_Exclusive.

Со временем неизбежно появятся новые метрики и показатели репликации данных, которые вы захотите измерять, однако на сегодняшний день перечисленных выше достаточно для обеспечения эффективной работы репликации и поддержки ваших SLO.

Репликация с одним ведущим узлом наиболее популярна благодаря своей относительной простоте. Тем не менее бывают случаи, когда этот подход не удовлетворяет локальным требованиям доступности и другим запросам. Разрешение записи в кластер базы данных из нескольких ведущих узлов позволяет уменьшить ущерб в случае сбоя ведущего узла, а ведущие узлы можно размещать в разных зонах и регионах, чтобы обеспечить более высокую производительность. Далее мы рассмотрим подходы и проблемы этого способа репликации.

Репликация с несколькими ведущими узлами

На практике есть два совершенно разных способа освободиться от установки использовать репликацию с одним узлом. Первый способ — это то, что можно назвать многонаправленной репликацией или традиционной репликацией с несколькими ведущими узлами. При таком подходе концепция роли ведущего узла сохраняется и ведущие узлы предназначены для получения операций записи и распределения их по репликам, а также по другим ведущим узлам. Как правило, два ведущих узла располагаются в разных центрах обработки данных. Второй способ — запись в любом месте. Это означает, что любой узел в кластере базы данных может в любое время эффективно выполнять чтение и запись. Затем операции записи распространяются на все остальные узлы.

Какое бы решение ни было принято, результат получается более сложным, чем при репликации с одним ведущим узлом, поскольку необходимо добавить уровень разрешения конфликтов. Когда все записи поступают в один ведущий узел, мы предполагаем, что появления конфликтующих записей, поступающих на разные узлы, не будет. Но если разрешить запись в несколько узлов, то появляется вероятность возникновения конфликтов. Это необходимо спланировать соответствующим образом, в результате чего приложение усложняется.

Варианты использования репликации с несколькими ведущими узлами

Если результатом репликации с несколькими ведущими узлами является сложность, то какие требования могут стоить таких затрат и рисков? Давайте их рассмотрим.

Доступность. Когда при асинхронной репликации с одним ведущим узлом происходит сбой ведущего узла, это обычно влияет на доступность приложения в диапазоне от 30 секунд в лучшем случае и до 30 минут или даже до 1 часа и более

в худшем случае, в зависимости от того, как спроектирована система. Это связано с необходимостью проверки согласованности репликации, восстановления после сбоя или ряда других действий.

В некоторых случаях такое нарушение обслуживания может быть просто неприемлемым и не окажется ресурсов или возможности изменить приложение, чтобы отработать отказ более незаметно. Тогда возможность изменить баланс нагрузки между узлами для операций записи, возможно, стоит неизбежных осложнений.

Местонахождение. Организации может потребоваться запустить активные сайты в разных регионах, чтобы обеспечить низкую задержку для глобальной или распределенной клиентской базы. В приложениях с интенсивным чтением обычно это можно сделать и при репликации с одним ведущим узлом, расположенным в удаленной сети.

Но если приложение интенсивно записывает данные, то задержка при отправке операций записи по этим сетям на большие расстояния может быть очень ощутимой. В этом случае наилучшим подходом может оказаться размещение ведущего узла в каждом центре обработки данных и управление разрешением конфликтов.

Аварийное восстановление. Как и в случае с локальностью и доступностью, бывают ситуации, когда приложение настолько важно, что его необходимо разделить между центрами обработки данных, чтобы обеспечить доступность даже в редких случаях сбоя на уровне ЦОД. Вы можете достичь этой цели при репликации с одним ведущим узлом, но только если дополнительный ЦОД используется для чтения, как обсуждалось ранее, или как запасной вариант. Однако лишь немногие предприятия могут себе позволить развернуть для такой цели целый центр обработки данных, поэтому часто выбирается репликация с несколькими уровнями, чтобы оба центра обработки данных могли активно принимать трафик и поддерживать работу клиентов.

Принимая во внимание большое количество инфраструктур, работающих в облачных сервисах, и учитывая глобальные требования к распространению, вам в итоге по одной из вышеупомянутых причин практически неизбежно потребуется рассмотреть возможность репликации с несколькими уровнями. Зачастую многоуровневую репликацию физически можно реализовать нативно или с помощью стороннего программного обеспечения. Проблема заключается в управлении неизбежными конфликтами, которые произойдут при этом.

Разрешение конфликтов при традиционной разнонаправленной репликации

У традиционной многонаправленной репликации много общего с репликацией с одним ведущим узлом. В сущности, операции записи просто передаются в обоих направлениях, так как вы разрешаете выполнять запись более чем одному

ведущему узлу. Звучит хорошо и соответствует всем случаям использования, которые мы только что обсудили. Но если задействовать асинхронную репликацию как единственный возможный подход в среде с медленными сетевыми соединениями, в которой есть несколько центров обработки данных, то могут быть и будут проблемы. При задержках репликации или в случае сетей с несколькими разделами (partitioned network) если приложение опирается на сохраненное состояние в базе данных, то оно будет использовать устаревшее состояние. При устранении задержки репликации или разделения сети необходимо устранить конфликты между записями, которые были созданы с использованием разных версий состояния. Как вы и ваши программисты будете решать проблему конфликтующих записей при многоуровневой архитектуре репликации? Очень осторожно. Для этого можно использовать несколько методов.

Устранение конфликтов. Путь наименьшего сопротивления — это всегда попытка избежать проблем. Бывают случаи, когда можно выполнять операции записи или перенаправлять трафик таким образом, чтобы конфликты просто не происходили. Рассмотрим несколько примеров.

- Предоставление каждому ведущему узлу подмножества первичных ключей, которые могут быть сгенерированы лишь для этого конкретного ведущего узла. Это хорошо работает для приложений, предназначенных только для вставки/добавления. В простейшем случае это может быть один ведущий узел, который выполняет запись с инкрементными ключами с нечетными номерами, и второй ведущий узел, выполняющий запись с четными номерами.
- Использование подхода по установлению сходства, при котором каждый клиент всегда направляется к определенному ведущему узлу. Можно настроить этот выбор по региону, уникальному идентификатору и пр.
- Применение вторичного ведущего узла только в целях отработки отказа, так что, в сущности, операции записи выполняются в каждый момент времени лишь на одном ведущем узле, но для простоты поддерживается топология с несколькими ведущими узлами.
- Разделение на уровне приложений, когда в каждом регионе размещаются полные стеки приложений, чтобы исключить необходимость межрегиональной репликации по схеме «ведущий — ведущий».

Конечно, это не значит, что такой подход будет работать всегда просто потому, что выполнена соответствующая настройка. Возможны ошибки конфигурации, ошибки балансировки нагрузки и человеческие ошибки, которые могут привести к нарушению репликации или повреждению данных. Таким образом, вам все равно нужно быть готовыми к случайным конфликтам, даже если они редки. И, как мы уже говорили, чем реже ошибка, тем опаснее она может быть.

Кто последний, тот и прав. В случае, когда нельзя избежать потенциальных конфликтов записи, необходимо решить, как ими управлять, когда они все же произойдут. Один из самых распространенных алгоритмов, изначально реализованных

в хранилищах данных, — это Last Write Wins (LWW). В случае LWW, когда возникает конфликт двух записей, считается верной запись с более свежей меткой времени. Это может показаться довольно простым, но при использовании меток времени есть некоторые нюансы.

Метки времени — маленькая сладкая ложь

В большинстве серверных часов используется физическое время, которое выдается функцией gettimeofday(). Эти данные предоставляются оборудованием и NTP. Есть множество причин, по которым время иногда может двигаться назад, а не вперед, в том числе следующие:

- аппаратные проблемы;
- проблемы виртуализации;
- NTP не включен или на предыдущих серверах установлено неверное время;
- корректировочные (високосные) секунды.

Корректировочные секунды — это ужас. Длительность суток в POSIX составляет 86 400 секунд. Однако в действительности сутки не всегда длятся 86 400 секунд. Корректировочные секунды предназначены для того, чтобы сохранять одинаковую длительность суток, пропуская или удваивая счет секунд. Это может вызывать огромные проблемы, и Google распределяет время в сутках так, чтобы оно оставалось равномерным (http://bit.ly/2zzUTO6).

Бывают случаи, когда метод LWW относительно безопасен. Если можно выполнять неизменяемые операции записи, потому что известно корректное состояние данных во время записи, то использование LWW допустимо. Но если при выполнении записи вы полагаетесь на состояние, которое было прочитано в ходе транзакции, то вы сильно рискуете потерять данные в случае разделения сети.

Примерами хранилищ данных с реализациями LWW являются Cassandra и Riak. Фактически в публикации Dynamo (http://bit.ly/2zyOVwP) метод LWW рассматривается как один из двух вариантов, предлагаемых для обработки конфликтов обновления.

Специальные варианты разрешения конфликтов. Вследствие ограничений базовых алгоритмов, которые опираются на временные метки, часто приходится использовать дополнительные специальные способы. Многие репликаторы позволяют выполнять собственный код при обнаружении конфликта после записи. Логика, обеспечивающая автоматическое разрешение конфликтов записи, может быть довольно обширной, но и в этом случае остаются возможности для ошибок.

Используя оптимистическую репликацию, которая позволяет записывать и реплицировать все изменения, можно разрешить фоновым процессам, приложению или даже пользователям самим решать, что делать для разрешения этих конфликтов. Это может быть всего лишь предложение выбрать ту или иную версию объекта данных. В качестве альтернативы можно предложить полное слияние данных.

Бесконфликтно реплицируемые типы данных. Из-за сложной логики в специальном коде для разрешения конфликтов многие организации не соглашаются на такую работу и не готовы к рискам. Однако существует класс структур данных, которые созданы для эффективного управления записями из нескольких реплик в случае разных временных меток или сетевых проблем. Эти типы данных называются *бесконфликтными, реплицируемыми типами данных* (Conflict Free, Replicated Datatypes, CRDT). CRDT обеспечивают строгую окончательную согласованность (strong eventual consistency), поскольку данные всегда могут быть синхронизированы без конфликтов. На момент написания этой книги CRDT эффективно внедряются в Riak и используются в очень крупных реализациях онлайн-чата и онлайн-ставок.

Как видим, разрешение конфликтов в средах с несколькими ведущими узлами возможно, но это непростая задача. Сложность, связанная с распределенностью систем, весьма реальна, и при их разработке требуются значительные затраты времени и сил. Кроме того, полноценные реализации этих подходов могут быть невозможны в хранилищах данных, которые наиболее оптимально подходят для вас и вашей организации. Так что будьте очень осмотрительны, прежде чем нырять в кроличью нору репликации с несколькими ведущими узлами.

Репликация с записью в любом месте

Существует подход, альтернативный традиционной разнонаправленной репликации. В репликации с записью в любом месте нет ведущих узлов. Любой узел может принимать операции чтения и записи. Примерами такого подхода к репликации являются системы на основе Dynamo, такие как Riak, Cassandra и Voldemort. У этих систем есть ряд свойств, которые мы сейчас рассмотрим более подробно:

- согласованность «в конечном счете»;
- кворумы чтения и записи;
- нестрогие кворумы;
- антиэнтропия.

Разные системы различаются в реализациях, но их совокупность формирует подход к репликации без ведущих узлов, если только приложение допускает неупорядоченные записи. Обычно есть настраиваемые параметры, которые помогают изменить поведение этих систем для лучшего соответствия вашим потребностям, но наличие неупорядоченных записей неизбежно.

Согласованность «в конечном счете». Выражение «согласованность в конечном счете» (eventual consistency) часто используется применительно к классу хранилищ данных, известному как NoSQL. В распределенных системах возможны проблемы с сервером или сетью. Эти системы созданы распределенными для того, чтобы обеспечить постоянную доступность, но ценой согласованности данных. Если узел не

работает в течение нескольких минут, часов или даже дней, то быстро возникает расхождение между его данными и данными других узлов[1].

Когда системы восстанавливаются, они выполняют разрешение конфликтов на основании методов, описанных в предыдущем подразделе, включая следующие:

- LWW с использованием меток времени или «векторных часов»[2];
- специальный код;
- бесконфликтные реплицируемые типы данных.

Нет никакой гарантии, что данные остаются согласованными на всех узлах в любой момент, однако в конечном счете они будут согласованы. При создании хранилищ данных вы сами выбираете, сколько копий данных необходимо иметь, чтобы обеспечить кворум при сбоях.

С учетом сказанного, еще необходимо доказать, что итоговая согласованность действительно работает. Есть много возможностей потерять данные: как из-за неправильного понимания используемых методов разрешения конфликтов и результатов их применения, так и из-за ошибок. Jepsen — отличный набор тестов, который показывает, как можно эффективно протестировать отсутствие расхождений в данных в распределенном хранилище. Дополнительную информацию на эту тему вы найдете в следующих источниках:

- Jepsen. Distributed Systems Safety Research (https://jepsen.io/);
- *Fowler M.* Eventual Consistency (http://bit.ly/2zxs6JS);
- *Bailis P., Ghodsi A.* Eventual Consistency Today: Limitations, Extensions and Beyond (http://queue.acm.org/detail.cfm?id=2462076).

Кворумы чтения и записи. Один из главных критериев репликации с записью в любом месте — понимание того, сколько узлов должно быть доступно для отправки или приема данных с целью обеспечения согласованности. На уровне клиента или базы данных обычно есть возможность определить кворум. Традиционно кворумом называется минимальное количество членов некоего собрания, необходимое для ведения дел в этой группе. В случае распределенных систем это означает минимальное количество узлов чтения или записи, требуемое для обеспечения согласованности данных.

Например, в кластере, состоящем из трех узлов, можно допустить сбой одного узла. Это означает, что требуемый кворум для чтения и записи равен 2. При принятии решений о кворумах есть простая формула. Предположим, что N — количество узлов в кластере, R — количество доступных узлов для чтения, а W — количество

---

1 *Vogels W.* Eventually Consistent // practice. http://stanford.io/2zxRXBu.

2 *Baldoni R., Raynal M.* Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems // Distributed Systems Online. http://bit.ly/2zylMll.

узлов для записи. Тогда, если $(R + W) > N$, у вас есть эффективный кворум, чтобы гарантировать хотя бы одну корректную операцию чтения после записи.

В нашем примере с тремя узлами это означает, что требуется как минимум два узла чтения и два узла записи, учитывая, что $(2 + 2) > 3$. Если потерять два узла, то останется всего 1 + 1, что равно 2. Это меньше 3, и, таким образом, у вас нет кворума и кластер не должен возвращать данные чтения. Если при чтении двух узлов приложение получит два разных результата (либо на одном узле данные отсутствуют, либо данные двух узлов расходятся), то будет выполнено восстановление с использованием выбранных методов разрешения конфликтов. Это называется *разрешением конфликтов при чтении*.

Это далеко не все, что вам следует изучить для понимания кворумов и всей теории и практики распределенных систем. Рекомендуем ознакомиться со следующими статьями:

- *The Load, Capacity and Availability of Quorum Systems* (http://bit.ly/2xjZNRZ);
- *Quorum Systems: With Applications to Storage and Consensus*.

Нестрогие кворумы. Однажды наступит время, когда у вас будут работающие узлы, но в них не окажется данных, необходимых для получения кворума. Предположим, что узлы N1, N2 и N3 настроены на запись, причем N2 и N3 отключены, но доступны N1, N4 и N5. На данном этапе система должна перестать разрешать запись этих данных, пока один узел не будет введен в кластер и не восстановится кворум. Однако если вам важнее продолжить получать записи, то можно допустить нестрогий кворум для записи. Это означает, что другой узел может начать получать записи. Как только N2 или N3 возвратятся в кластер, данные могут быть переданы им обратно в процессе *направленной отправки* (hinted hand-off).

Кворумы — это компромисс между согласованностью и доступностью. Очень важно понимать, как в хранилище данных на самом деле реализованы кворумы. Вы должны понимать, когда допускается нечеткий кворум и какие кворумы могут привести к строгой согласованности. Информация из разных источников иногда вводит в заблуждение, поэтому тестирование реализаций должно стать частью вашей работы.

Антиэнтропия. Еще одним инструментом для поддержания итоговой согласованности является антиэнтропия. Хранилище данных на основе Dynamo способно достаточно эффективно поддерживать согласованность в период между восстановлением чтения и направленной отправкой. Но если данные считываются не очень часто, то несоответствия могут сохраняться достаточно долго. В будущем это может подвергнуть приложение риску получить устаревшие данные в случае аварийного переключения. Таким образом, должен существовать метод синхронизации данных в дополнение к обычному механизму. Этот процесс называется антиэнтропией.

Примером антиэнтропии является дерево Меркла, которое реализовано в Riak, Cassandra и Voldemort. Это сбалансированное дерево хешей объектов. Создавая иерархические деревья, фоновый процесс антиэнтропии может быстро выявить различие значений между узлами и восстанавливать их. Эти хеш-деревья изменяются при записи и регулярно очищаются и восстанавливаются, чтобы можно было свести к минимуму риск пропуска противоречивых данных.

Антиэнтропия критически важна для хранилищ данных, в которых содержится много редко используемых данных. Это хорошее дополнение к направленной отправке и восстановлению чтения. Убедившись, что для хранилищ данных активирована антиэнтропия, вы обеспечите максимально возможную согласованность в распределенном хранилище данных.

В деталях реализации этих систем есть существенные различия, однако рекомендации в общем сводятся к обсуждавшемуся ранее. Если предполагать, что ваше приложение способно справляться с неупорядоченными записями и устаревшими операциями чтения, система репликации без ведущего узла может обеспечить отличную отказоустойчивость и масштабируемость.

Изучив три наиболее распространенных подхода к реплицируемым хранилищам данных, вы и ваши коллеги должны получить общее представление о методах распределения данных между несколькими системами. Это позволяет проектировать системы, которые отвечают потребностям вашей организации в доступности, масштабе, производительности и локальности данных.

Резюме

Эта глава представляет собой ускоренный курс по хранилищам данных. Мы рассмотрели разные этапы работы с хранилищем, начиная с того, как данные записываются на диск, и до того, как они перемещаются по кластерам и центрам обработки данных. Это основа архитектуры баз данных. Рассмотрев эту информацию, пусть и на весьма общем уровне, мы еще сильнее углубимся в характеристики хранилищ данных, чтобы помочь вам и вашим командам разработчиков выбрать подходящую архитектуру конкретно для вашей организации.

11 Справочник по видам хранилищ данных

С технической точки зрения хранилище данных — это именно то, что следует из названия: хранилище данных, связанное с ним программное обеспечение и структура, позволяющая хранить, изменять данные и получать к ним доступ. Но здесь мы будем говорить о хранилищах данных, которые используются в современных организациях для достижения их целей в тех случаях, когда очень большое количество пользователей получает доступ к очень большим объемам данных в условиях конкурентности.

Обычно читатели пользуются справочниками по видам различных представителей флоры, фауны и других природных объектов, чтобы их идентифицировать. Наша цель в этой главе — помочь вам понять характеристики различных хранилищ данных. Мы надеемся, что, вооружившись этой информацией, вы сможете ближе познакомиться с разными видами хранилищ данных, лучше понимая различные варианты их использования, а также нюансы их обслуживания.

В этой главе мы начнем с определения характеристик и категорий хранилища данных, имеющих большое значение для разработчиков приложений, которые записывают и используют данные. Затем мы подробнее рассмотрим категории, которые больше заинтересуют архитекторов и операторов хранилищ данных. Мы считаем, что любой, кто занимается разработкой, проектированием или эксплуатацией хранилищ данных, должен знать обо всех особенностях конкретного хранилища данных. Однако мы не ставили себе задачу дать вам исчерпывающее описание, поскольку наверняка найдется хранилище данных, которое где-нибудь используется, но которого не окажется в этом описании. Вместо этого мы приведем хорошую подборку хранилищ данных и перечислим инструменты для дальнейшего изучения темы, в зависимости от ваших потребностей и задач.

Концептуальные особенности хранилища данных

Существует множество способов классификации хранилищ данных. Выбранные вами методы классификации будут зависеть от ваших задач и способов взаимодействия с хранилищем данных. *Вы создаете функции в приложениях, которые запрашивают, хранят и изменяют данные? Вы запрашиваете и анализируете данные для принятия решения? Вы проектируете системы, на которых будет работать база данных? Вы управляете базой данных, настраиваете или контролируете ее?* Каждой роли соответствует свой вид базы и хранящихся в ней данных.

В мире ORM и бессерверных архитектур, предоставляющих API, наблюдается тенденция к абстрагированию хранилищ данных от потребителей, которые их используют. Мы с этим не согласны. Понимание каждой характеристики и соответствующего хранилища данных, которое вы (или кто-то еще) выбираете, имеет решающее значение для продуктивной работы. Бесплатного сыра не бывает, и каждая привлекательная функция подразумевает достижение некоего компромисса. Критически важно гарантировать, что команды, работающие с хранилищами данных, будут полностью осведомлены об этом.

Модель данных

Для большинства разработчиков программного обеспечения (SWE) модель данных является одной из самых важных категорий. То, как структурированы данные и каким образом управляются отношения между ними, имеет решающее значение для тех, кто создает приложения на основе этих данных. Это также существенно влияет на управление изменениями и миграциями БД, поскольку разные модели часто по-разному управляют такими изменениями.

В этом подразделе описаны четыре основных варианта моделей данных: реляционная, «ключ — значение», документная и навигационная, или графовая, модель. У каждой из них своя сфера применения, свои ограничения и особенности. Реляционная модель до сих пор была самой популярной. Учитывая, что такие хранилища выпускались огромным количеством производителей очень долгое время, эту модель можно считать наиболее понятным, стабильным и наименее рискованным из доступных вариантов.

Реляционная модель

Реляционная модель существует с того момента, когда ее первоначально предложил Э. Ф. Кодд (https://ru.wikipedia.org/wiki/Кодд,_Эдгар), опубликовавший свою статью *A Relational Model of Data for Large Shared Data Banks* («Реляционная модель данных

для больших совместно используемых банков данных») в 1970 году. Годом ранее он описал эту модель в корпоративной документации IBM. Поскольку в книге мы стремимся не столько дать вам полную информацию, сколько помочь понять системы, с которыми вы работаете сегодня, сосредоточимся на применении реляционных систем в современных организациях.

Главный принцип моделей реляционных баз данных состоит в том, что данные представляются в виде множества отношений, основанных на уникальных ключах, которые являются основными идентификаторами для фрагмента данных. Реляционная модель подразумевает согласованность данных между таблицами с ограничениями на отношения, количество элементов, значения и требованиями к существующим или несуществующим определенным атрибутам. Реляционная модель формализована и характеризуется различными уровнями строгости этой формализации, также известными как *нормальные формы*. Реальность такова, что многие из этих теоретических требований отодвигаются на второй план, когда в игру вступают производительность и конкурентность[1].

Хорошо известны такие реляционные базы данных, как *Oracle*, *MySQL*, *PostgreSQL*, *DB2*, *SQL Server* и *Sybase*. Есть и специализированные альтернативы: *Google Spanner*, *Amazon RedShift*, *NuoDB* и *Firebird*. Многие из этих альтернативных систем классифицируются как NewSQL. Они считаются подклассом систем управления реляционными базами данных, которые стремятся преодолеть некоторые барьеры конкурентности и масштабирования, гарантируя при этом согласованность. Мы подробнее обсудим их далее в главе.

Реляционная модель предлагает хорошо изученный подход к поиску данных. Благодаря поддержке объединений (джойны, joins), отношений «один ко многим» и «многие ко многим» разработчики получают высокий уровень гибкости в определении модели данных. Однако это же может привести к гораздо более сложным подходам при эволюции схемы БД, поскольку добавление, изменение или удаление таблиц, связей и атрибутов может потребовать непростой синхронизации и перемещения различных частей. Это чревато дорогостоящими и рискованными изменениями, как было описано в главе 8.

Многие команды разработчиков программного обеспечения предпочитают использовать систему объектно-реляционного управления (Object Relational Management, ORM), чтобы упростить работу, сопоставляя реляционную модель с объектной, определенной на программном уровне. Такие ORM могут быть отличными инструментами для ускорения разработки, но их использование также приводит к некоторым проблемам для команды DBR-инженеров.

---

[1] *Codd E. F.* The relational model for database management: version 2 // ACM Digital Library. http://dl.acm.org/citation.cfm?id=77708.

ORM И ВЫ

За последнее десятилетие ORM-системы стали совершеннее, и вам как DBRE-специалисту больше не нужно с ними осторожничать, как раньше. Тем не менее остаются ошибки, которые необходимо учитывать.

- ORM привязывают операции чтения и записи к таблицам, что усложняет любое количество оптимизаций части рабочей нагрузки, поскольку влияет на всю рабочую нагрузку.
- ORM могут задерживать выполнение транзакций намного дольше, чем следовало бы, что сильно влияет на итоговый расход ресурсов из-за широкого использования моментальных снимков.
- ORM могут генерировать огромное количество ненужных запросов.
- ORM могут генерировать сложные и плохо работающие запросы.

Помимо этих очевидных сложностей, одна из самых больших проблем заключается в том, что ORM абстрагирует базу данных, что исключает возможность совместной работы, необходимой для масштабирования, причем независимо от количества администраторов баз данных (DBA) в организации, которые работают с ORM. ORM позволяет игнорировать ограничения и усложняет логику. Кроме того, DBR-инженерам труднее понимать особенности взаимодействия приложения с хранилищами данных[1].

Из-за этого многие разработчики и архитекторы программного обеспечения считают реляционные системы негибкими и не позволяющими повысить скорость разработки. Однако это далеко не всегда так, и далее в главе мы приведем список достоинств и недостатков таких систем и опровергнем некоторые мифы о них.

Модель «ключ — значение»

Модель «ключ — значение» позволяет хранить данные в виде словаря или хеша. Словарь аналогичен таблице и может содержать любое количество объектов. В каждом объекте может храниться любое количество атрибутов или полей. Как и в реляционной базе данных, эти записи однозначно идентифицируются по ключу. В отличие от реляционных баз данных, здесь нет возможности создать соответствия между объектами на основе этих ключей.

Хранилище данных типа «ключ — значение» видит объект как блок данных. По сути, этот блок ничего не знает о данных, которые содержит, поэтому каждый

[1] *Ireland C. et. al.* A Classification of Object-Relational Impedance Mismatch // IEEE Xplore. http://bit.ly/2zymmQ3.

объект может иметь разные поля, вложенные объекты и пр. Такое разнообразие дорого обходится, в том числе подразумевая вероятность несогласованности, поскольку такие правила не применимы на уровне обычного общего хранилища. Аналогично недоступны возможности хранилища в отношении типов данных и индексации. С другой стороны, отсутствуют многие издержки, связанные с управлением различными типами данных, ограничениями и отношениями. Если для приложения все это некритично, то можно использовать хранилище такого типа.

Хранилища «ключ — значение» бывают самыми разными. Одним из примеров таких хранилищ является Dynamo. В 2007 году издательство Amazon опубликовало статью *Dynamo*, где были перечислены методы для создания высокодоступного распределенного хранилища данных. Среди основанных на Dynamo систем — *Aerospike* (http://bit.ly/2zy49Ch), *Cassandra* (http://bit.ly/2zyAucm), *Riak* (http://bit.ly/2zxKUsJ) и *Voldemort* (http://stanford.io/2zxtrk3). К другим реализациям хранилищ типа «ключ — значение» относятся *Redis*, *Oracle NoSQL Database* и *Tokyo Cabinet*.

Документная модель

С технической точки зрения документная модель основана на модели «ключ — значение». Отличие документной модели состоит в том, что база данных содержит метаданные о структуре документа. Это позволяет оптимизировать тип данных, вторичную индексацию и пр. В документ-ориентированных хранилищах вся информация об объекте содержится в одном месте, а не распределена между таблицами. Это позволяет извлекать все данные одним вызовом, без объединений, которые, хотя и просты, могут потреблять значительное количество ресурсов. Также обычно устраняется потребность в уровне ORM.

С другой стороны, это означает, что документ-ориентированные хранилища требуют денормализации, если существуют разные представления требуемого объекта. Это может привести к излишнему увеличению размера базы и проблемам согласованности. Кроме того, для управления данными нужны сторонние инструменты, поскольку схема БД больше не является самодокументирующейся системой[1].

Управление данными

Управление данными — это управление доступностью, целостностью и безопасностью данных, которые сохраняет и использует организация. Перед введением новых атрибутов данных их следует тщательно изучать и документировать. Использование JSON для хранения данных позволяет слишком легко вводить новые атрибуты.

[1] *Vera H. et. al.* Data Modeling for NoSQL Document-Oriented Databases. http://ceur-ws.org/Vol-1478/paper17.pdf.

Навигационная модель

Развитие навигационных моделей начиналось с иерархических и сетевых баз данных. Сегодня, говоря о навигационных моделях, мы почти всегда подразумеваем графовую модель данных. В графовой базе данных для представления и хранения данных и связей между объектами используются узлы, ребра и свойства. Узел содержит данные об определенном объекте, ребро представляет собой отношение объекта к другому объекту, а свойства позволяют указывать дополнительные данные об узле. Поскольку отношения сохраняются непосредственно как часть данных, можно легко переходить по ссылкам. Часто за один вызов можно получить весь граф.

Графовые хранилища, подобно документ-ориентированным, более точно соответствуют структуре объектно-ориентированных приложений. При их использовании нет необходимости в объединениях, и они могут оказаться более гибкими с точки зрения эволюции модели данных. Конечно, это актуально только для тех данных, которые идеально подходят для запросов, соответствующих графу. Традиционные запросы могут оказаться гораздо менее производительными[1].

Каждая из этих моделей подойдет для конкретного вида приложений. Мы сведем воедино различные варианты и опишем возможные компромиссы. Для начала рассмотрим поддержку транзакций и особенности реализации.

Транзакции

Способ, которым хранилище данных обрабатывает транзакции, также является важной характеристикой. Транзакция — это фактически логическая единица работы базы данных, которую можно считать неделимой. Для обеспечения согласованности в хранилище данных все операции в транзакции должны либо выполняться, либо откатываться.

Возможность верить, что все операции транзакции будут либо зафиксированы, либо откатаны, значительно упрощает логику обработки ошибок в приложениях на основе БД. Эти гарантии транзакционной модели позволяют разработчикам не принимать во внимание определенные виды сбоев и конкурентность, которые могут потребовать значительных ресурсов и циклов разработки.

Если прежде вы работали преимущественно с традиционными реляционными хранилищами данных, то, вероятно, принимаете существование транзакций как должное. Это объясняется тем, что почти все эти хранилища данных основаны на представленной IBM в 1975 году модели ACID, которая будет описана далее. Все

[1] *Stonebraker M., Held G.* Networks, Hierarchies and Relations in Data Base Management Systems. http://bit.ly/2zxrbcq.

операции чтения и записи считаются транзакциями, и для их выполнения используется базовая архитектура совместного доступа к базам данных.

ACID

Аббревиатура ACID описывает требования к БД: *атомарность* (Atomicity), *согласованность* (Consistency), *изолированность (изоляция)* (Isolation) и *устойчивость* (Durability). Эту аббревиатуру ввели в 1983 году Андреас Рейтер (Andreas Reuter) и Тео Хардер (Theo Härder) (https://en.wikipedia.org/wiki/ACID#cite_note-2), опираясь на работу Джима Грея (Jim Gray) (https://en.wikipedia.org/wiki/Jim_Gray_(computer_scientist)), который учел атомарность, согласованность и устойчивость, но не учел изолированность. Эти четыре свойства описывают основные требования к транзакционной системе, которые повлияли на многие аспекты разработки в СУБД.

При работе с хранилищем данных крайне важно понять, как в нем определены и реализованы эти концепции, потому что для различных хранилищ характерны разные особенности. Зная об этом, нам следует разобрать каждое свойство в отдельности и рассмотреть варианты, которые встречаются на практике[1].

Атомарность

Под атомарностью понимается гарантия того, что вся транзакция будет либо зафиксирована и записана в хранилище данных, либо откатана. В атомарной базе данных невозможны такие действия, как частичная запись или откат. В этом контексте атомарность не относится к атомарным операциям, которые встречаются при разработке программного обеспечения. Этот термин обозначает гарантию изолированности от конкурентных процессов, наблюдающих за работой *в процессе*, а не только до и после получения результатов.

Есть много причин, по которым транзакция может завершиться неудачно и потребовать отката. Например, клиентский процесс может прервать промежуточную транзакцию или, возможно, может прерваться соединение из-за сбоя сети. Аналогичным образом, при сбоях базы данных, сбоях сервера и многих других операциях может потребоваться откат частично завершенной транзакции.

В PostgreSQL это реализовано с помощью журнала `pg_log`. Транзакции записываются в `pg_log`, и им присваивается одно из состояний: *«в процессе»*, *«зафиксирована»* или *«прервана»*. Если клиент отменит или откатит транзакцию, она будет помечена как прерванная. Внутренние процессы также будут периодически помечать транзакции как прерванные, если у них нет связанных внутренних процессов.

---

[1] *Vieira M. et al.* Timely ACID Transactions in DBMS. http://bit.ly/2zyR2Rh.

Важно отметить, что операции записи можно считать атомарными, только если атомарными являются базовые операции записи страницы (сектора, блока) диска. Существуют значительные разногласия по поводу атомарности блочных операций записи. Большинство современных дисков разрешает продолжать запись в сектор даже при сбое диска. Но, в зависимости от уровней абстракции между физическим диском и фактической записью на диск, есть риск потерять при этом данные.

Согласованность

Гарантия согласованности — это гарантия того, что любая транзакция переведет базу данных из одного допустимого состояния в другое. При записи транзакции предполагается, что она не может нарушить определенные правила. С технической точки зрения согласованность определяется не на уровне базы данных, а на уровне приложения. Однако традиционные БД предоставляют разработчикам инструменты для обеспечения такой согласованности. Эти инструменты характеризуются наличием некоторых ограничений и триггеров. К ограничениям могут относиться внешние ключи с каскадированием, ограничения на ненулевые значения, ограничения уникальности, ограничения типов и длины данных и даже ограничения на определенные значения, разрешенные в конкретном поле.

Немного удручает, что согласованность повсеместно используется в базах данных и программном обеспечении. В теореме CAP тоже используется термин «согласованность», но в другом значении. Этот же термин вы услышите при обсуждении хеширования и репликации.

Изолированность

Гарантия изолированности — это обещание того, что одновременное выполнение транзакций приведет к тому же состоянию, которое возникло бы, если бы эти транзакции выполнялись последовательно. В базах данных, отвечающих ACID, это реализовано сочетанием методов, которые могут включать в себя блокировки записи, блокировки чтения и моментальные снимки. В совокупности это называется *конкурентным управлением*. На практике существует несколько типов конкурентного управления, которые могут привести к различным видам поведения в базе данных. Более строгие версии могут значительно повлиять на производительность конкурентных транзакций, в то время как более слабые способны привести к повышению производительности за счет меньшей изолированности.

Стандарт ANSI/ISO SQL определяет четыре возможных уровня изоляции транзакции. Каждый уровень потенциально может обеспечить разные результаты для одной и той же транзакции. Эти уровни определены с точки зрения трех возможных ситуаций, которые могут быть разрешены или запрещены.

- *«Грязное» чтение*. При «грязном» чтении есть вероятность прочитать незафиксированные («грязные») данные, которые записываются в другой транзакции от другого клиента.
- *Неповторяющееся чтение*. При неповторяющемся чтении в контексте транзакции, если дважды выполнить одну и ту же операцию чтения, есть вероятность получить другие результаты, основанные на других одновременных действиях в базе данных.
- *Фантомное чтение*. При фантомном чтении в контексте транзакции, если дважды выполнить одну и ту же операцию чтения, данные, возвращаемые во второй раз, будут отличаться от тех, что были получены в первый раз. В отличие от неповторяющегося чтения при фантомном чтении данные, которые однажды были запрошены, не изменяются, но по запросу возвращается больше данных, чем в первый раз.

Чтобы избежать этих явлений, можно использовать четыре возможных уровня изоляции.

- *Чтение незафиксированных данных (Read Uncommitted)*. Это самый низкий уровень изоляции. Здесь разрешены «грязное» чтение, «грязная» запись, неповторяющееся и фантомное чтение.
- *Чтение фиксированных данных (Read Committed)*. На этом уровне изоляции цель состоит в том, чтобы избежать операций «грязного» чтения и «грязной» записи. Другими словами, не должно быть возможности читать или перезаписывать незафиксированные данные. Некоторые базы данных избегают «грязной» записи, блокируя запись для выбранных данных. Блокировки записи удерживаются до фиксации данных, а блокировка чтения снимается после выбора данных. «Грязное» чтение обычно выполняется путем сохранения двух копий данных, записываемых в транзакции: одной — из более старых зафиксированных данных, которые будут использоваться для чтения из других транзакций, и второй — из данных, которые были записаны, но не зафиксированы.

  Однако при изолированном режиме чтения зафиксированных данных по-прежнему можно выполнять неповторяющиеся операции чтения. Если незафиксированные данные будут прочитаны один раз, а затем после фиксации прочитаны снова, то вы увидите разные значения в контексте вашей собственной транзакции.
- *Повторяемое чтение (Repeatable Reads)*. Чтобы достичь уровня изоляции при чтении зафиксированных данных и избежать неповторяющегося чтения, необходимо реализовать дополнительные элементы управления. Если в базе данных используются блокировки для управления параллельным выполнением операций, клиент должен будет сохранять блокировки чтения и записи до конца транзакции. В итоге остается возможность фантомного чтения. Как можно догадаться, такой подход, основанный на блокировке, сложен и может

привести к значительному падению производительности в высококонкурентных системах.

Другой способ добиться этого — использовать изоляцию моментальных снимков. При изоляции моментальных снимков после запуска транзакции клиент увидит образ базы данных, актуальный на текущий момент. В этом моментальном снимке не будут отображаться дополнительные операции записи, что позволит получить согласованные, повторяемые операции чтения при длительных запросах. Для изоляции моментального снимка используется блокировка записи, но не блокировки чтения. Цель состоит в том, чтобы гарантировать, что при чтении не блокируются источники записи, и наоборот. Поскольку для этого требуется более двух копий, это называется *управлением параллельным доступом с помощью многоверсионности* (Multiversion Concurrency Control, MVCC).

При изоляции моментального снимка в случае повторяемого чтения все еще возможны расхождения при записи. В этой ситуации можно разрешить две записи в один и тот же столбец или несколько столбцов, расположенных подряд, из двух разных источников, которые прочитали обновляемые столбцы. Это приводит к появлению строк, которые содержат данные из двух транзакций.

- *Упорядочиваемый уровень, или сериализуемость (Serializable)*. Это самый высокий уровень изоляции, который предназначен для того, чтобы избежать всех вышеупомянутых явлений. Как и при повторяемом чтении, если используются блокировки для управления параллельным выполнением операций, то во время транзакции сохраняются блокировки чтения и записи. Однако здесь есть ряд дополнений, и стратегия блокировки называется двухфазной блокировкой (2-phase locking, 2PL).

  Двухфазная блокировка может быть общей или монопольной. Несколько читателей могут иметь общие блокировки чтения. Однако, чтобы можно было получить монопольную блокировку записи, все общие блокировки чтения после фиксации должны быть сняты. Точно так же, если происходит запись, общие блокировки чтения не могут быть получены. В этом режиме в средах с высоким уровнем параллелизма транзакции могут часто простаивать в ожидании блокировки. Это явление называется *взаимоблокировкой* (deadlock). Кроме того, для запросов, использующих диапазоны в условиях WHERE, также должны устанавливаться блокировки диапазона. В ином случае происходят операции фантомного чтения.

  2PL может существенно влиять на задержку транзакций. Когда много транзакций находятся в режиме ожидания, время отклика всей системы может значительно увеличиться. Таким образом, многие системы на практике не реализуют сериализуемость и придерживаются повторяемого чтения.

  Подход без блокировки основан на изоляции моментальных снимков и называется последовательной изоляцией моментальных снимков (Serial Snapshot Isolation, SSI). Он представляет собой «оптимистичную» сериализацию, при

которой база данных ожидает фиксации, чтобы выяснить, не произошли ли какие-либо действия, приводящие к проблемам сериализации, чаще всего это конфликт записи. Использование такого подхода позволяет значительно уменьшить время отклика в системах, для которых нехарактерны нарушения параллельного выполнения операций. Однако если такие нарушения наблюдаются регулярно, то постоянные откаты и повторные попытки могут значительно влиять на время отклика.

Поскольку каждый уровень изоляции строже последующего в том смысле, что ни один более высокий уровень не допускает действие, запрещенное на более низком уровне, стандарт разрешает СУБД выполнять транзакцию на уровне изоляции, более строгом, чем запрашиваемый. Таким образом, например, транзакция «чтение фиксированных данных» может фактически выполняться на уровне изоляции повторяемого чтения.

ИЗМЕНЧИВОСТЬ ПРИ ИЗОЛЯЦИИ

Как уже упоминалось, хранилища данных значительно различаются между собой по реализации стандартов изоляции ANSI.

- PostgreSQL — имеет уровни чтения фиксированных данных, повторяемого чтения и сериализуемости. Использует SSI для сериализации.
- Oracle — есть только возможность чтения зафиксированных данных и сериализуемости. Уровень сериализуемости ближе к повторяемому чтению, чем к действительному уровню сериализуемости.
- MySQL с InnoDB — имеет уровни чтения фиксированных данных, повторяемого чтения и сериализуемости. Использует 2PL для сериализации, но не находит потерянные обновления[1].

Здесь мы лишь слегка затронули тему изоляции. Далее в главе мы предложим вам несколько отличных ресурсов для более подробного изучения этой темы.

Устойчивость

Гарантия устойчивости подразумевает, что после того, как транзакция будет выполнена, она так и останется успешно завершенной. Транзакция остается устойчивой независимо от того, случится ли сбой питания, сбой базы данных, аппаратный сбой или возникнет любая другая проблема. Очевидно, при работе с БД мы не можем

[1] *Schwartz B.* If Eventual Consistency Seems Hard, Wait Till You Try MVCC. http://bit.ly/2zyNy1m.

гарантировать, что используемое оборудование обеспечит такую устойчивость. Как уже говорилось в главе 5, мы можем предполагать, что база данных синхронизирована с диском, хотя в реальности это далеко не всегда так.

Устойчивость тесно связана с атомарностью, поскольку без первой невозможно обеспечить вторую. Во многих базах данных предусмотрен *журнал упреждающей записи* (Write-Ahead Log, WAL), где фиксируются все записи до того, как они будут записаны на диск. Этот журнал используется для отмены или для повторного применения транзакции. Если происходит сбой, то при перезапуске база данных может сверить этот журнал с данными системы и определить, нужно ли отменить, завершить или проигнорировать транзакцию.

Подобно уровням изоляции, бывают ситуации, когда устойчивость можно и нужно ослаблять, чтобы обеспечивать соответствие требованиям производительности. Для достижения абсолютной устойчивости сброс данных на диск должен происходить при каждом завершении транзакции. Это весьма затратная процедура, которая не требуется для всех транзакций и записей. Например, в MySQL можно настроить периодический сброс журнала `Innodb`, а не после завершения каждой транзакции. Точно так же можно поступить с журналами репликации[1].

Несмотря на то что мы ограничились довольно общим обзором, вам должно быть понятно, как много деталей скрыто и воспринимается как должное в транзакционных системах. Будучи DBRE-специалистом, вы должны хорошо разбираться в реализациях. Зачастую детали этих реализаций не совсем понятны из документации, и углубленные тесты с использованием таких инструментов, как *Jepsen* (https://github.com/jepsen-io/jepsen) и *Hermitage* (https://github.com/ept/hermitage), помогут вам в ваших исследованиях.

Надеемся, приведенной информации вам будет достаточно, чтобы выбрать подходящую конфигурацию и понять, когда стоит ослабить устойчивость или использовать более мягкую изоляцию. В то же время могут оказаться важными знания о том, когда значения по умолчанию, предлагаемые базой данных, не соответствуют потребностям ваших приложений.

BASE

Поскольку инженеры всегда искали альтернативы традиционным реляционным системам, термин BASE начал использоваться как альтернатива для ACID. Аббревиатура BASE расшифровывается как *Basically Available, Soft state and Eventual consistency* — «базовая доступность, неустойчивое состояние, согласованность в конечном счете». Эта концепция описывает нетранзакционные системы, которые

[1] *Sears R., Brewer E.* Segment-Based Recovery: Write-ahead loggin revisited. http://www.vldb.org/pvldb/2/vldb09-583.pdf.

являются распределенными и могут иметь довольно нестандартные возможности репликации и синхронизации. В отличие от систем ACID, когда система активна и принимает трафик, у систем BASE может не быть четкого состояния. Аналогично, поскольку нет необходимости управлять параллельным выполнением операций для транзакций, пропускная способность записи и параллельность в BASE-системах могут быть значительно выше за счет атомарности, изоляции и согласованности[1].

Рассмотрев модели данных и транзакционные модели, доступные для хранилищ данных, мы изучили их особенности, наиболее важные для разработчиков. Однако существует множество других характеристик, которые необходимо учитывать при выборе не только базы данных, но и всей операционной экосистемы и инфраструктуры, на основе которой используются эти БД (табл. 11.1).

Таблица 11.1. Основные концептуальные характеристики хранилища данных

Характеристика	MySQL	Cassandra	MongoDB	Neo4J
Модель данных	Реляционная	Ключ — значение	Документная	Навигационная
Готовность модели	Готова	2008	2007	2010
Отношения между объектами	Внешние ключи	Нет	DBRefs	Ядро к модели
Атомарность	Поддерживается	На уровне разделения	На уровне документа	На уровне объекта
Согласованность (в узле)	Поддерживается	Не поддерживается	Не поддерживается	Строгая согласованность
Согласованность (в кластере)	На основании репликации	Итоговая (настраивается)	Итоговая	Поддержка XA-транзакций
Изоляция	MVCC	Возможность сериализации	Чтение незафиксированных данных	Чтение зафиксированных данных
Устойчивость	Для DML, но не для DDL	Поддерживается, настраивается	Поддерживается, настраивается	Поддерживается, WAL

Здесь все описано очень кратко, и при выборе хранилища данных для вашего приложения следует придирчиво изучить функциональность поддерживаемых свойств в ходе тестирования эффективности этих систем. Но даже с этими оговорками есть определенные особенности, знание которых поможет сделать подходящий выбор

[1] *Roe C.* The Question of Database Transaction Processing: An ACID, Base, NoSQL Primer. http://bit.ly/2zw5Obr.

для вашего приложения. Далее мы рассмотрим внутренние характеристики хранилища данных, чтобы получить полную картину.

Внутренние характеристики хранилища данных

Есть множество способов описать и классифицировать хранилища данных. Используемая модель данных и транзакционная модель напрямую определяют архитектуру и логику приложения. Поэтому они, как правило, находятся в центре внимания разработчиков, которые хотят обеспечить высокую скорость и гибкость. Внутренние, архитектурные реализации этих баз данных часто остаются для разработчика «черными ящиками». Однако они имеют решающее значение при выборе подходящего хранилища данных с целью длительного использования.

Хранилище

Мы подробно рассмотрели хранилища в главе 10. Каждое хранилище предусматривает один или несколько возможных вариантов размещения данных на диске. Это часто реализовано в форме систем хранения. Механизм хранения определяет чтение и запись данных, блокировку, параллельный доступ к данным и все остальные процессы, необходимые для управления структурами данных, такими как индексы B-дерева, журнально-структурированные деревья со слиянием (Log Structured Merge, LSM) и фильтры Блума.

В некоторых базах данных, таких как MySQL и MongoDB, предлагается несколько вариантов механизмов хранения. Например, в MongoDB можно использовать MMap, в MMap — WiredTiger либо структуры LSM или RocksDB, основанные на деревьях LSM. Реализации механизмов хранения значительно различаются, но их характеристики обычно можно разбить на следующие категории:

- производительность операций записи;
- производительность операций чтения;
- устойчивость операций записи;
- размер хранилища.

Изучение механизмов хранения на основе этих характеристик поможет определить, какой из них выбрать для вашего хранилища данных. Часто приходится идти на компромисс, выбирая между производительностью операций чтения/записи и устойчивостью. Есть также функции, которые можно реализовать для увеличения устойчивости механизма хранения. Понимание этого и, конечно же,

сравнительный анализ и проверка достоверности утверждений об устойчивости имеют первостепенное значение.

Вездесущая теорема CAP

При обсуждении этих характеристик часто ссылаются на *теорему Эрика Брюера* (Eric Brewer), или *теорему CAP* (рис. 11.1). Она гласит, что в любой распределенной системе можно обеспечить не более двух из трех свойств или гарантий: согласованность (Consistency), доступность (Availability) или устойчивость к разделению (Partition tolerance). Как и в случае с ACID, эти термины являются обобщенными. Каждое из этих свойств на практике может иметь или не иметь место. Часто систему обозначают как CP или AP, что означает, что она обеспечивает только два конкретных свойства из трех. Тем не менее если вы углубитесь в изучение таких систем, то обнаружите, что реализации каждого конкретного свойства в них являются неполными, поскольку удается достичь лишь частичной доступности или согласованности[1].

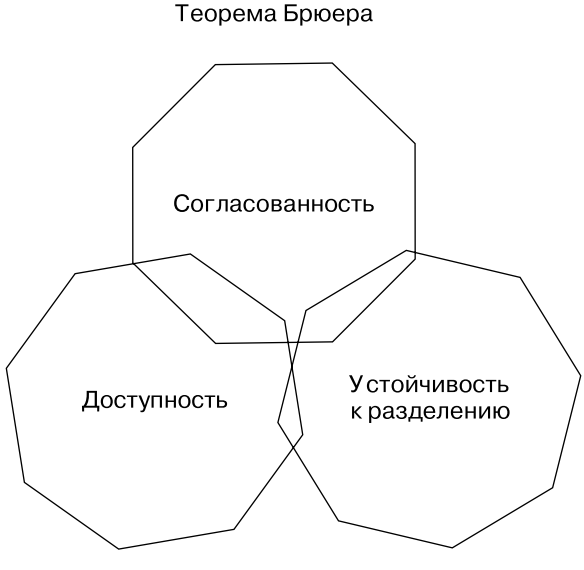


Рис. 11.1. Визуализация теоремы CAP, или теоремы Брюера

Теорема CAP призвана помочь разработчикам понять компромисс между согласованностью или доступностью. Разделение сети в распределенных системах неизбежно. Сети по своей сути ненадежны. В этом случае узлы с одной стороны

[1] *Kleppmann M.* Please stop calling databases CP or AP. http://bit.ly/2zxUA6k.

разделения неизбежно потеряют согласованность, если позволят обновлять состояние. Если согласованность в приоритете, то одна часть сети должна стать недоступной. Рассмотрим каждый из этих терминов подробнее, чтобы понять, что эти понятия могут означать.

Согласованность

Напомним, что мы обсуждали согласованность (буква C в аббревиатуре ACID) в подразделе, посвященном транзакциям. Досадно, что согласованность в ACID отличается от согласованности в CAP. В ACID согласованность означает, что транзакция сохраняет все правила и ограничения базы данных. В CAP согласованность означает *линеаризуемость*. Линеаризуемость гарантирует, что множество операций, выполняемых над объектом в распределенной базе данных, будет происходить в режиме реального времени. Поскольку операции могут быть прочитаны и записаны, это означает, что они должны появляться в той последовательности, в какой выполняются остальными пользователями в системе. Линеаризуемость выступает гарантией последовательной согласованности в реальном времени[1].

Согласованность в ACID, как и согласованность в CAP, не может поддерживаться при разделении сети. Это означает, что транзакционные хранилища данных на основе ACID в случае разделения сети могут гарантировать непротиворечивость, только жертвуя доступностью[2]. BASE-системы были разработаны, помимо прочего, для того, чтобы справляться с разделениями сети без ущерба для доступности.

Доступность

Доступность в теореме CAP относится к способности обрабатывать запросы. Обычно большинство распределенных систем могут обеспечивать согласованность и доступность. Но перед разделением сети, когда одно подмножество узлов отделяется от другого, необходимо принять решение о том, как сохранить доступность за счет согласованности. Конечно, ни одна система не способна поддерживать 100%-ную доступность в течение длительного времени, и это соответствует тому, что мы говорили ранее о доступности как о непрерывном процессе.

Устойчивость к разделению

Разделение сети — это временный или постоянный разрыв соединения, который приводит к нарушению связи между двумя подмножествами сетевой инфраструктуры.

---

1 *Bailis P.* Linearizability versus Serializability. http://bit.ly/2v08Ymd.

2 *Brewer E.* CAP Twelve Years Later: How the 'Rules' Have Changed. http://bit.ly/2psQjuC.

По сути, при этом часто создаются два кластера меньшего размера. Каждый из них может считать себя единственным существующим кластером и допускать продолжение записи. Это приводит к появлению двух различающихся наборов данных и также называется *разделением ведущих узлов* (split brain).

Теорема CAP была опубликована для того, чтобы помочь находить компромисс между согласованностью и доступностью в распределенных хранилищах данных. На практике разделения сети занимают небольшое количество времени в жизненном цикле хранилища данных. Согласованность и доступность могут и должны предоставляться одновременно. Однако при разделении сети система должна иметь возможность обнаруживать это разделение и управлять им, чтобы восстановить согласованность и доступность.

Следует отметить, что в теореме CAP никак не учитывается время отклика (задержка) и производительность. Время отклика может быть столь же важным, как и доступность, а слишком большое время отклика тоже может быть потенциальной причиной проблем согласованности. Достаточно длительное время отклика может превысить допустимый предел, и это вынудит систему перейти в состояние отказа, связанное с разделением сети. В отношении задержки существует компромисс, на который идут гораздо чаще, чем для достижения согласованности и доступности. Фактически еще одна важная причина, по которой возникли системы BASE и движение NoSQL, заключалась в требованиях к повышению производительности при масштабировании.

Теперь, когда мы ознакомились с теоремой CAP, обсудим, как это может повлиять на нашу систематизацию баз данных. Можно было бы предпочесть концепцию CP концепции AP, но мы уже обсуждали, насколько упрощен такой подход. Вместо этого рассмотрим, как распределенные системы поддерживают согласованность и доступность.

Компромисс между согласованностью и задержкой

Распределенная система считается строго согласованной, если все ее узлы видят все транзакции в той же последовательности, в которой те были записаны. Другими словами, такая система линеаризуема. В теореме CAP отдельно обсуждается, как распределенное хранилище данных способствует согласованности или доступности в случае разделения сети. Однако согласованность требуется на протяжении всего жизненного цикла хранилища данных, и ее нельзя рассматривать только через парадигму CAP[1].

Всем хотелось бы иметь строго согласованное распределенное хранилище данных, но мало кто на самом деле согласится с последствиями, связанными с задержкой и доступностью. Итак, компромиссы сделаны. В этом подразделе мы рассмотрим

[1] *Abadi D.* Consistency Tradeoffs in Modern Distributed Database System Design. http://bit.ly/2zxYLiH.

достигнутые компромиссы и то, как они влияют на общую согласованность в кластере. Это позволит нам четко оценить хранилище данных и определить, соответствует ли оно нашим потребностям.

При записи данных в узле распределенного хранилища данные должны быть реплицированы, чтобы гарантировать доступность. Как уже отмечалось, эти данные можно воспроизвести следующими способами.

- Отправлять записи на все узлы одновременно, синхронно.
- Отправлять записи на один узел, играющий роль основного. Репликация выполняется асинхронно, полусинхронно или синхронно.
- Отправлять записи на любой узел, который играет роль основного только для этой транзакции. Репликация выполняется асинхронно, полусинхронно или синхронно.

При записи в любой узел кластера есть риск нарушить целостность без использования какого-либо координирующего процесса, такого как Paxos, который способен эффективно упорядочивать записи. Данный процесс, по сути, увеличивает задержку транзакции. Это один из способов обеспечения высокой согласованности за счет задержки. И наоборот, если задержка важнее упорядоченности, то на текущем этапе согласованность может быть принесена в жертву. Это компромисс *между задержкой и упорядоченностью*.

При отправке операций записи на один узел, откуда они должны передаваться на другие узлы, существует вероятность того, что первичный узел, принимающий записи, окажется недоступен из-за отключения/сбоя или из-за недопустимой нагрузки, которая может привести к простоям. В таких случаях повторение операции или ожидание увеличивает задержку. Однако можно настроить балансировщики нагрузки или прокси-серверы для отправки операций записи на другой узел после истечения определенного времени ожидания. Но запись в другом узле может привести к проблемам согласованности, поскольку возможны конфликты, если исходная транзакция будет уже обработана, но не предоставит подтверждение этому. Количество повторных попыток или увеличенное время ожидания повлияют на задержку и в итоге на доступность при сохранении согласованности. Это компромисс *повторных попыток при первичных простоях*.

При чтении с узла также возможны простои или недоступность. Передача операций чтения на другие узлы в асинхронно реплицируемых средах может привести к устареванию операций чтения и, следовательно, к проблемам согласованности. Увеличение времени ожидания и повторных попыток снижает риск появления противоречивых результатов, но это происходит за счет увеличения задержки. Это компромисс *повторов при первичных простоях чтения*.

При синхронной записи на все узлы как упорядоченной последовательностью процессов, так и в ходе репликации вы также получите дополнительную задержку

вследствие ожидания всех транзакций, направленных на другие узлы. Если эти узлы находятся в перегруженной сети или интенсивно обмениваются данными, то задержка может быть очень большой. Это компромисс *между синхронной репликацией и задержкой*. Для смягчения этого компромисса можно использовать полусинхронную репликацию. Она уменьшает потенциальное влияние задержки за счет уменьшения количества узлов и сетевых соединений, которые могут повлиять на репликацию. Полусинхронная репликация — это тоже компромисс, поскольку, увеличивая задержку, вы повышаете риск потери данных, жертвуя доступностью. Это компромисс между *полусинхронной доступностью и задержкой*.

Каждый из этих компромиссов открывает возможности для настройки системы в сторону лучшей согласованности или уменьшения задержки. Эти компромиссы имеют решающее значение в тех случаях, когда в системах не бывает разделения сети и просто обслуживаются запросы.

Доступность

Кроме согласованности и ее связи с задержкой, у нас есть доступность. Бывает доступность при разделении сети, как в теореме CAP. Но есть также ежедневная доступность в условиях сбоев узла, многоузловой сети или всего кластера. Обсуждая доступность в распределенных системах, мы хотим поговорить не просто о доступности, а об *урожайности* (yield) и *урожае* (harvest). *Урожайность* (http://mauricio.github.io/pwl-harvest-yield/#/20) означает способность получить ответ на ваш вопрос. *Урожай* описывает полноту набора данных. Вместо того чтобы просто задать вопрос о том, работает ли система, вы можете оценить, какой подход лучше использовать при неудаче — уменьшить урожайность или урожай.

Первый вопрос, который нужно задать себе в отношении распределенной системы, — допустимо ли снижение урожая в пользу поддержания урожайности. Например, допустимо ли доставлять 75 % данных в запросе, если пропускная способность узла сократилась на 25 %? Если предоставлять большое количество результатов поиска, это может быть приемлемым. Тогда это обеспечивает большую устойчивость к сбоям, что может означать снижение коэффициентов репликации в кольце Cassandra. Аналогично, если урожай должен быть близок к 100 %, нужно распространять больше копий данных. Это означает не только большее количество реплик, но и больше зон доступности, в которых должны существовать реплики.

Это также можно увидеть в декомпозиции приложений на подсистемы. Независимо от того, проявляется ли это в функциональном разделении или на уровне микросервисов, в результате один отказ может быть изолирован от остальной части системы. Такое разделение часто требует вмешательства программистов, но это пример снижения урожая для поддержания урожайности.

Понимание механизмов хранения и того, как в хранилище данных реализуются компромиссы согласованности, доступности и задержки, позволяет изучить

хранилище данных «изнутри», что дополняет концептуальные особенности, которые мы уже рассмотрели. Инженеры и архитекторы, отвечающие за производительность и функциональность приложения, больше всего заинтересованы в концептуальных характеристиках. Инженеры по эксплуатации и базам данных часто стремятся к тому, чтобы внутренние характеристики (табл. 11.2) соответствовали целевым показателям качества обслуживания (SLO), которые диктуются бизнесом.

Таблица 11.2. Сводная таблица внутренних характеристик хранилища данных

Характеристика	**MySQL**	**Cassandra**	**MongoDB**	**Neo4J**
Механизмы хранения	Плагины, в основном B-деревья	Только LSM	Плагины, B-деревья и LSM	Собственное графовое хранилище
Распределенная согласованность	Главный приоритет — согласованность	Итоговая согласованность, вторичная после доступности	Главный приоритет — согласованность	Главный приоритет — согласованность
Распределенная доступность	Вторичная, после согласованности	Главный приоритет — доступность	Вторичная, после согласованности	Вторичная, после согласованности
Задержка	Настраивается в зависимости от устойчивости	Оптимизируется для операций записи	Настраивается в зависимости от согласованности	Оптимизируется для операций чтения

Резюме

Надеемся, что в этой главе мы привели хороший список характеристик во всем их разнообразии, чтобы вы могли выбрать подходящее хранилище данных. Информация будет полезна для вас независимо от того, новое у вас приложение или уже существующее либо вы только получили запрос от команды разработчиков на поиск новейшего и наилучшего хранилища данных. Теперь, когда мы перешли от обычных устройств хранения к хранилищам данных, пришло время рассмотреть архитектуры и конвейеры данных.

12 Примеры архитектур данных

Теперь, когда мы перешли к механизмам хранения и отдельным хранилищам данных, расширим наше представление о том, как эти хранилища могут вписаться в мультисистемные архитектуры. Архитектуры, которые включают в себя только одно хранилище данных, встречаются редко. В реальности обычно имеется несколько способов сохранения данных, несколько потребителей этих данных и несколько производителей. В этой главе мы представим вам очаровательный маленький образчик архитектурных компонентов, которые часто используются для реализации хранилищ данных, а затем опишем несколько управляемых данными архитектур, которые можно встретить на практике. Мы также обсудим, для каких задач подходят эти архитектуры.

Это далеко не исчерпывающее описание, однако оно должно стать отличным обзором экосистемы и дать вам представление о том, что именно нужно искать. Прочитав главу, вы поймете, в чем заключается эффективное использование этих компонентов и каким образом они могут влиять на сервисы обработки данных — как положительно, так и отрицательно.

Архитектурные компоненты

Каждый из этих компонентов входит в сферу повседневных обязанностей инженера по надежности баз данных (DBRE). Прошли те времена, когда мы могли игнорировать компоненты, окружающие экосистему данных. Каждый из них оказывает определенное влияние на общую доступность сервиса, целостность и согласованность данных. Их невозможно игнорировать при разработке сервисов и операционных процессов.

Внешние хранилища данных

Внешняя база данных — это суть того, что мы обсуждали в книге. Пользователи приложений обычно запрашивают, вставляют и изменяют данные в хранилищах,

имея некий уровень доступа к данным. Исторически многие приложения создавались так, как будто эти базы данных доступны всегда. Это означает, что каждый раз, когда внешние хранилища данных не работают или настолько заняты, что работают слишком медленно и это влияет на уровень качества обслуживания клиентов, приложения становятся непригодными для использования.

Традиционно эти системы назывались системами интерактивной транзакционной обработки (OnLine Transactional Processing, OLTP). Для них было характерно множество быстрых транзакций, поэтому они предназначались для очень быстрых запросов, обеспечивали целостность данных при высокой степени параллелизма и масштабирования в зависимости от количества транзакций, которые они могут обрабатывать одновременно. Ожидается, что все данные будут обрабатываться в реальном времени, со всеми необходимыми деталями для поддержки сервисов, которые их используют. Каждый пользователь или транзакция ищет небольшое подмножество данных. Это означает, что шаблоны запросов имеют тенденцию фокусироваться на поиске и доступе к какому-то небольшому подмножеству из большого множества данных. Для этого решающее значение имеют эффективное индексирование, изоляция и параллельный доступ, поэтому такие задачи обычно решаются с помощью реляционных систем.

Внешнее хранилище данных также характеризуется тем, что данные в него вносятся главным образом самими пользователями. Есть также хранилища, ориентированные на пользователя и в основном предназначенные для аналитики. В таких хранилищах традиционно используется интерактивный анализ данных (OnLine Analytics Processing, OLAP).

Мы уже обсуждали различные характеристики большинства таких хранилищ данных: структуру хранения, модель данных, парадигмы ACID/BASE и компромиссы между доступностью, согласованностью и задержкой. Кроме того, необходимо учитывать общую пригодность к эксплуатации и то, каким образом хранилища интегрируются с остальной частью экосистемы. К основным характеристикам относятся следующие:

- низкая задержка записи и запросов;
- высокая доступность;
- низкое среднее время восстановления (MTTR);
- возможность масштабирования в зависимости от трафика приложения;
- простая интеграция с приложением и эксплуатационными сервисами.

Как вы, вероятно, догадались, это довольно серьезные требования для любой архитектуры. Их редко удается выполнить без помощи других компонентов инфраструктуры, которые мы рассмотрим далее.

Уровень доступа к данным

Приложение часто разбивается на уровень представления и уровень бизнес-логики. Уровень бизнес-логики — это то, что также называют уровнем доступа к данным (Data Access Layer, DAL). Он обеспечивает упрощенный доступ к постоянным хранилищам данных, используемым для чтения и записи компонентов приложения. Часто проявляется как набор объектов с атрибутами и методами, которые ссылаются на хранимые процедуры или запросы. Такая абстракция скрывает сложность хранилища данных от разработчиков программного обеспечения (SWE).

Примером DAL является использование DAO (Data Access Objects). DAO предоставляют интерфейсы доступа к базе данных, соответствующие обращениям приложений к базе данных. Не затрагивая слой хранения данных, разработчики могут отдельно тестировать доступ к данным. Подобным образом можно подставить заглушки вместо баз данных и протестировать приложение. Принято считать, что при этом требуется гораздо больше кодирования в Java Database Connectivity (JDBC) или в аналогичных интерфейсах. Тем не менее, будучи приближен к базе данных, он позволяет писать эффективный код, когда требуется обеспечить производительность с помощью специальных методов. Другой нередко проявляющийся недостаток заключается в том, что это требует от разработчика более глубокого понимания схемы. Хотя мы думаем, что это скорее достоинство и чем лучше разработчики понимают схему, тем лучше для всех.

Еще одним примером DAL является объектно-реляционное отображение (Object-Relational Mapper, ORM). Как мы уже рассказывали, ORM нам не нравится по ряду причин. Однако у него есть некоторые преимущества, включая кэширование и аудит. Очень важно понимать, что именно используют ваши команды программистов и какая гибкость или ограничения вводятся при кодировании и оптимизации доступа к данным.

Прокси базы данных

Уровень прокси базы данных находится между серверами приложений и внешними хранилищами данных. Некоторые прокси расположены на уровне 4 (Layer 4, L4) сетевого транспортного уровня и используют доступную там информацию, чтобы принять решение, как распределять запросы, поступающие от серверов приложений на серверы баз данных. Сюда входят IP-адреса и порты источника и приемника в заголовке пакета. Функциональность L4 позволяет распределять трафик по определенному алгоритму, но при этом не принимаются во внимание другие факторы, такие как загрузка или задержка репликации.

Уровни 4 и 7

Говоря об уровнях, мы имеем в виду уровни модели взаимодействия открытых систем (Open Systems Interconnection, OSI). Эта модель определяет сетевой стандарт.

Прокси уровня 7 (L7) работает на самом высоком уровне сетевого транспортного уровня. Это также называется уровнем приложения или в данном случае уровнем HTTP. На уровне L7 прокси-сервер имеет доступ к значительно большему количеству данных из пакета TCP. Прокси-серверы L7 понимают протокол базы данных и протокол маршрутизации и имеют широкие возможности по настройке.

Их функции могут включать в себя следующее:

- проверку работоспособности и перенаправление на работоспособные серверы;
- разделение операций чтения и записи и передачу операций чтения в реплики;
- переписывание запросов, которые нельзя настроить в коде, с целью оптимизации.
- кэширование и возвращение кэшированных результатов запроса;
- перенаправление трафика на реплики, на которых нет задержек;
- генерирование метрик по запросам;
- фильтрацию трафика брандмауэром в зависимости от типов запросов или хостов.

Вся эта функциональность, конечно, дорого обходится. Ценой компромисса в этом случае является время ожидания. Таким образом, то, какой прокси-сервер выбрать — L4 или L7, — зависит от потребностей вашей команды в функциональности и запросов по времени ожидания. Прокси-сервер может смягчить последствия технических недоработок, исправляя их на другом уровне. Но это также может привести к тому, что недоработки будут игнорироваться в течение более длительного времени и приложение окажется сложнее переносить на другие хранилища данных.

Доступность

Одной из основных функций прокси-сервера является возможность перенаправления трафика в случае сбоя узла. Если этот узел является репликой, то прокси-сервер может проверить его работоспособность и отключить узел. При первичной ошибке или ошибке записи, если допускается только один модуль записи, прокси-сервер может остановить трафик, чтобы обеспечить безопасное переключение при сбое. В любом случае использование эффективного прокси-уровня позволит значительно снизить MTTR сбоя — при условии, что вы настроили прокси-слой на устойчивость к ошибкам. В противном случае вы просто добавите новую точку отказа.

Целостность данных

Если прокси-сервер просто перенаправляет трафик, это мало влияет на целостность данных. Однако есть некоторые возможности улучшить ситуацию. В среде с асинхронной репликацией прокси-сервер L7 может отключить все реплики, которые отстают в репликации. Это уменьшает вероятность возврата в приложение устаревших данных.

С другой стороны, если прокси-сервер кэширует данные, чтобы уменьшить задержку и увеличить емкость узлов базы данных, есть вероятность возврата устаревших данных из этого кэша, если только их надежно не аннулировать после записи. Эту и другие проблемы кэширования мы обсудим в подразделе, посвященном кэшированию.

Масштабируемость

Имея хороший уровень прокси, можно значительно улучшить масштабирование. Мы уже обсуждали паттерны масштабирования, которые включают в себя распределение операций чтения между несколькими репликами. Без прокси тоже можно выполнять элементарное распределение нагрузки, но без учета нагрузки и задержки и, следовательно, не так эффективно. Использование прокси-сервера для распределения операций чтения — очень эффективный подход для распределения большой рабочей нагрузки при чтениях. Это предполагает, что у организации достаточно денег, чтобы заплатить за все эти реплики, и налажена эффективная автоматизация для управления ими.

Еще одной задачей, при которой прокси-уровень может улучшить масштабируемость, является сбрасывание нагрузки (load shedding). Многие серверы баз данных страдают от большого количества одновременных подключений. Прокси-уровень может играть роль очереди соединений, удерживая большое количество соединений, в то же время позволяя только определенному числу из них выполнять работу в базе данных. Это может показаться нелогичным из-за увеличения задержки вследствие параллелизма, однако ограничение количества соединений и работы может обеспечить большую пропускную способность.

Время отклика

Время отклика выступает решающим фактором при добавлении еще одного уровня в поток транзакций. Прокси уровня L4 добавляет минимальную задержку, но на уровне L7 она увеличивается гораздо больше. С другой стороны, существуют способы смягчить этот эффект и сократить ожидание: кэширование регулярно выполняемых запросов, избегание чрезмерно загруженных серверов и переписывание неэффективных запросов. Тип компромисса зависит от приложений,

и именно вам, архитекторам и инженерам, придется принимать эти решения. Как и с большинством компромиссов, мы рекомендуем стремиться к простоте, а не к значительной функциональности, если только это не абсолютно необходимое условие. Простота и меньшее время отклика могут оказаться невероятно важны для организации.

Теперь, когда мы рассмотрели уровни доступа к данным и уровни прокси-серверов — уровни, которые помогают установить связь между приложением и базой данных, поговорим о приложениях, которые функционируют начиная от базы данных и далее. Это системы, которые потребляют, обрабатывают, преобразуют данные и, как правило, формируют полезный результат и генерируют прибыль, используя эти внешние хранилища данных.

Системы обработки событий и сообщений

Данные не существуют сами по себе. Поскольку транзакции происходят в основном хранилище данных, то после регистрации транзакции может быть выполнено любое количество действий. Другими словами, транзакции выступают как события. Ниже приводятся примеры действий, которые могут потребоваться после транзакции:

- передать данные дальше, для выполнения аналитики и помещения в хранилище;
- выполнить заказы;
- обнаружить мошенничество и проверить транзакцию;
- загрузить данные в кэши или сети доставки контента (Content Delivery Networks, CDN);
- откорректировать и опубликовать параметры персонализации.

Это всего лишь несколько примеров возможных действий, которые могут быть инициированы после транзакции. Системы обработки событий и сообщений построены так, чтобы использовать данные из хранилищ и публиковать эти события для последующих процессов с целью воздействия на эти данные. Программное обеспечение для обмена сообщениями и событиями обеспечивает обмен данными между приложениями с помощью асинхронных сообщений. Эти системы создают сообщения на основе того, что они обнаруживают в хранилище данных. Затем сообщения используются другими приложениями.

Существуют разнообразные приложения, которые выполняют эту функцию. На момент написания книги самым популярным был сервис Apache Kafka, который функционирует как распределенный журнал. Kafka допускает значительное горизонтальное масштабирование на уровне поставщика, потребителя и топика.

К другим системам относятся RabbitMQ, Amazon Kinesis и ActiveMQ. В простейшем случае это может быть задача типа «извлечь, преобразовать и загрузить» (ETL) или задачи, при которых хранилище данных постоянно или периодически опрашивается на предмет появления новых данных.

Доступность

Система обработки событий может положительно повлиять на доступность хранилища данных. А именно, выведя за пределы хранилища события и их обработку, мы исключаем для хранилища один режим активности. Это уменьшает загруженность ресурсов и снижает уровень параллелизма, что может положительно влиять на доступность основных сервисов. Это также означает, что обработка событий может происходить даже в периоды пиковой активности, поскольку не приходится беспокоиться о нарушениях в работе производственных систем.

Целостность данных

Один из самых больших рисков при перемещении данных между системами — риск повреждения и потери данных. В распределенной шине передачи сообщений с любым количеством источников и потребителей данных валидация этих данных становится большой проблемой. Для данных, которые нельзя потерять, потребитель должен создать их копию в той или иной форме и записать ее обратно в шину. Затем получатель данных аудита может прочитать эти сообщения и сравнить их с исходными. Это большая работа в плане кодирования и потребляемых ресурсов. Но это абсолютно необходимо сделать для тех данных, которые вы не можете позволить себе потерять. Конечно, можно сделать выборку данных, которые допустимо частично потерять. Если обнаружится потеря данных, то должен быть предусмотрен способ уведомить последующие процессы о том, что необходимо повторно обработать определенное сообщение. Каким образом это произойдет, зависит от потребителя.

Не менее важно убедиться, что механизм хранения событий или сообщений достаточно устойчив, чтобы обеспечить надежное хранение в течение всего времени жизни сообщения. Если возможна потеря данных, то есть и проблема с их целостностью. Противоположностью этого является дублирование. Если данные можно продублировать, то событие будет обработано повторно. Когда невозможно гарантировать, что обработка будет идемпотентной и, следовательно, может быть повторно запущена с теми же результатами в том случае, если событие придется обработать повторно, тогда лучше использовать хранилище данных, которое можно индексировать соответствующим образом для управления дубликатами.

Масштабируемость

Как только что обсуждалось, при извлечении событий из внешнего хранилища данных и их последующей обработке снижается общая нагрузка на базу данных. Это разделение рабочей нагрузки как шаг в направлении к масштабированию. Разделяя независимые рабочие нагрузки, мы устраняем нежелательное взаимное влияние разнотипных рабочих нагрузок.

Время отклика

Освобождение внешних хранилищ данных от обработки событий пойдет только на пользу — это позволит уменьшить количество потенциальных конфликтов, что, в свою очередь, может снизить время отклика для внешних приложений. Однако время, необходимое для передачи событий из внешнего хранилища данных в систему обработки событий, увеличивает задержку при обработке этих событий. Асинхронный характер этого процесса означает, что приложения должны создаваться так, чтобы допускать задержку в процессе обработки.

Итак, теперь, когда мы разобрались, как добраться до хранилищ данных, как связать данные из внешнего хранилища с последующими потребителями, рассмотрим некоторые из этих потребителей.

Кэши и устройства памяти

Мы уже обсудили, насколько невероятно медленно предоставляется доступ к диску по сравнению с доступом к памяти. Вот почему мы стремимся сделать так, чтобы все наборы данных из хранилищ соответствовали структурам памяти, таким как буферные кэши, вместо того чтобы считывать данные с диска. При этом во многих организациях просто не предусмотрен бюджет на хранение набора данных в памяти. В случае данных, размер которых слишком велик для кэша хранилища данных, стоит обратить внимание на системы кэширования и хранилища, оптимизированные для обработки данных в оперативной памяти (in-memory datastores).

Системы кэширования и хранилища данных для обработки в памяти, по сути, весьма похожи. Их назначение — хранение данных не на диске, а в оперативной памяти, что обеспечивает быстрый доступ при чтении. Если данные редко изменяются и вы допускаете возможность эфемерного хранения в памяти, это может оказаться отличным вариантом. Многие хранилища данных для обработки в памяти обеспечивают длительное хранение благодаря копированию данных на диск с помощью фоновых процессов, которые асинхронно запускаются из транзакции. Но это повышает риск потери данных в случае сбоя до того, как они будут сохранены.

Хранилища данных для обработки в памяти часто имеют дополнительные функции, такие как использование расширенных типов данных, возможность репликации и отработка отказов. Современные хранилища для обработки данных в памяти также оптимизированы для доступа к данным в оперативной памяти, что может оказаться быстрее, чем даже работа с набором данных в реляционной системе, которая полностью помещается в кэше базы данных. Кэши баз данных по-прежнему должны проверять актуальность своих хранилищ и соблюдать требования к параллельному доступу и ACID. Таким образом, хранилище данных в памяти может оказаться наиболее подходящим вариантом для систем, требующих наименьшего времени отклика.

Существует три подхода к загрузке кэша данными. Первый подход состоит в том, чтобы поместить данные в кэш после того, как они будут записаны в постоянное хранилище наподобие реляционной БД. Второй вариант — одновременная запись в кэш и долговременное хранилище в режиме двойной записи. Такой подход не очень надежен из-за возможности сбоя одной из двух записей. Для выполнения этой работы требуется также проводить валидацию после записи или двухэтапную фиксацию. Последний подход — сначала записать в кэш, а затем разрешить асинхронное сохранение на диске. Такой вариант также называется *сквозной записью* (write-through). Обсудим, как каждый из этих подходов может повлиять на систему баз данных.

Доступность

Использование кэша может положительно повлиять на доступность, позволяя продолжать чтение даже при сбое хранилища данных. Для приложений с интенсивным чтением это может быть очень полезным. Однако если в системе кэширования произойдет увеличение емкости и/или сокращение задержек либо случится сбой, то данные, предназначенные для длительного хранения, могут оказаться не соответствующими трафику, отправляемому обратно. Поддержание доступности на уровне кэширования становится таким же важным, как обеспечение доступности в хранилище данных, а это означает, что управление усложняется в два раза.

Еще одна важная проблема — «несметная орда». Она проявляется, когда у всех серверов кэша есть фрагмент данных, к которому обращаются очень часто и который аннулирован из-за записи или превышения времени отклика. Когда это происходит, большое количество серверов одновременно отправляют запросы на чтение в постоянное хранилище, чтобы обновить кэш. Это может привести к конкурентному резервному копированию, которое чревато перегрузкой постоянного хранилища данных. Когда это происходит, может оказаться невозможным обслуживать операции чтения из кэша или постоянно хранимых данных.

Есть несколько способов управления «несметной ордой». Довольно легко можно гарантировать, что простои кэша не будут затрагивать друг друга, хотя это и не очень масштабируемый подход. Более управляемый вариант состоит в добавлении уровня прокси-кэша, который бы ограничивал прямой доступ к хранилищу данных. На этом этапе будет уровень постоянного хранения, уровень прокси-кэша и уровень кэша. Как видим, масштабирование может быстро стать довольно сложным.

Целостность данных

В системах кэширования может быть сложно обеспечивать целостность данных. Кэшированные данные являются в каком-то смысле статичной копией постоянно изменяющихся данных в хранилище. Поэтому необходимо найти компромисс между допустимой частотой обновления данных и, соответственно, скрытой нагрузкой на постоянное хранилище, с одной стороны, и тем, какова вероятность того, что все, кто будет обращаться к кэшу, получит устаревшие данные, с другой стороны.

Помещая в кэш сохраненные данные, вы должны быть готовы к тому, что у вас появятся устаревшие данные. Этот подход лучше всего работает в отношении более или менее статичных данных, которые редко приходится аннулировать и получать повторно. К ним относятся наборы поисковых данных, такие как геокоды, метаданные приложения и контент, предназначенный только для чтения, скажем новостные статьи.

При одновременном вводе данных в постоянное хранилище и в кэш вы исключаете возможность устаревания данных. Однако при этом по-прежнему нужно проверять, что данные не устарели. Непосредственно после записи и затем периодически необходимо продолжать проверки, чтобы убедиться, что вы предоставляете потребителю правильные данные.

Наконец, при первой записи данных в кэш с последующей записью в постоянное хранилище следует иметь средства восстановления в случае сбоя кэша, прежде чем он сможет переслать запись на уровень постоянного хранения. Одно из возможных решений — журналирование всех операций записи и обработка их как событий, которые могут привести к запуску кода валидации. Конечно, на этом этапе в значительной степени воспроизводится та же сложность, которую предоставляют многие самодельные хранилища данных. Таким образом, приходится тщательно взвесить, стоит ли сквозная запись той сложности, которая необходима для валидации и поддержания целостности хранилищ данных.

Масштабируемость

Одной из основных причин использования кэширования и баз данных в оперативной памяти является масштабирование в соответствии с рабочей нагрузкой, создаваемой чтением данных. Поэтому действительно, добавив уровень кэширования, можно эффективно увеличить степень масштабирования за счет сложности среды. Но, как обсуждалось выше, в таком случае успешное достижение целевых показателей качества обслуживания (SLO) начинает зависеть от этого уровня кэширования. Если серверы кэша выйдут из строя и информация на них окажется недействительной или поврежденной, то вы больше не сможете рассчитывать, что хранилище данных сохранит операции чтения приложения.

Время отклика

Использование уровня кэширования не только обеспечивает масштабируемость, но и позволяет уменьшить задержку чтения. Это отличное применение кэша или технологии хранения данных в оперативной памяти, но если кэш-сервер выйдет из строя, то вы больше не сможете напрямую отслеживать состояние долговременных хранилищ без рабочего кэш-сервера. Стоит запланировать и выполнять периодические тесты для трафика чтения, обходя кэш в тестовых средах, чтобы увидеть, как на уровне постоянного хранения данных одновременно обрабатываются рабочие нагрузки записи и чтения. Если есть вероятность отказа долговременного хранилища в производственной среде, то вы, вероятно, захотите проверить эти сбои в среде тестирования.

Кэширование и использование хранилищ данных для обработки в памяти характерны для многих успешных архитектур, основанных на данных. Они хорошо сочетаются с промежуточным ПО (middleware), управляемым событиями, способны существенно повысить масштабируемость и производительность приложения. Однако они создают дополнительный уровень управления с точки зрения эксплуатационных расходов, риска отказов и риска целостности данных. Это часто упускают из виду (хотя так делать нельзя) из-за того, что многие системы кэширования чертовски просты в реализации. Ваша задача как DBRE-специалиста — гарантировать серьезное отношение в организации к ответственности за обеспечение доступности и целостности этой подсистемы.

Каждый из рассмотренных компонентов играет важную роль в обеспечении доступности, масштабируемости и расширенной функциональности ваших хранилищ данных. Но каждый из них также увеличивает архитектурную сложность, эксплуатационные зависимости, риск потери данных и проблемы их целостности. Теперь, когда мы рассмотрели некоторые отдельные компоненты, разберем несколько архитектур, используемых для передачи данных через клиентские приложения в хранилище и далее к многочисленным последующим сервисам.

Архитектуры данных

Рассматриваемые в этой главе архитектуры данных — примеры наборов систем, управляемых данными и предназначенных для приема, обработки и доставки данных. В каждом из этих примеров мы обсудим основные принципы, способы использования и требуемые компромиссы. Наша цель — дать вам представление о том, каким образом хранилища данных и связанные с ними системы, обсуждавшиеся в этой книге, функционируют в реальном мире. Разумеется, это просто примеры. Реальные приложения всегда будут сильно отличаться от описанных, в зависимости от потребностей организации.

Лямбда и каппа

Лямбда (Lambda) — это архитектура для обработки больших данных в реальном времени. Сегодня она повсеместно применяется во многих организациях. Каппа (Kappa) — это паттерн реагирования, предполагающий стремление к простоте и использованию преимуществ новейшего программного обеспечения. Сначала рассмотрим первую архитектуру, а затем обсудим ее изменения.

Лямбда-архитектура

Лямбда-архитектура предназначена для обработки значительных объемов данных; обработка выполняется быстро, что позволяет обслуживать запросы почти в реальном времени, а также поддерживать длительные вычисления. Лямбда-архитектура состоит из трех уровней: пакетной обработки (batch processing), обработки в реальном времени (real-time processing) и уровня запросов (query layer), или обслуживающего уровня (рис. 12.1).

Если данные записываются во внешнее хранилище, то можно использовать распределенный журнал, такой как Kafka, чтобы создать распределенный и неизменяемый журнал для уровней обработки лямбда-архитектуры. Некоторые данные записываются непосредственно в журналы сервисов, а не в хранилище данных. Потоковые уровни принимают эти данные.

У лямбда-архитектуры есть два уровня обработки, поэтому она поддерживает быстрые запросы с «достаточно хорошей» быстрой обработкой, а также позволяет выполнять более полные и точные вычисления. Уровень пакетной обработки часто выполняется с помощью запросов MapReduce, время ожидания которых просто недопустимо для запросов, выполняемых в реальном или почти в реальном времени. Типичным хранилищем данных для пакетного уровня является распределенная файловая система, такая как Apache Hadoop. Затем MapReduce создает пакетные представления из основного набора данных.

Рис. 12.1. Лямбда-архитектура

Уровень обработки в реальном времени обрабатывает потоки данных с той скоростью, с которой они поступают, не требуя полноты или абсолютной точности. Он предоставляет компромисс между качеством данных и задержкой для передачи последних данных приложению. Задача этого уровня — заполнить пробел в данных, задерживающихся на пакетном уровне. После завершения пакетной обработки данные с уровня реального времени заменяются данными с пакетного уровня. Обычно это выполняется с помощью технологии Apache Storm или Spark с поддержкой хранилища данных с малой задержкой наподобие Apache Cassandra.

Наконец, есть еще обслуживающий уровень, который возвращает данные приложению. Он содержит пакетные представления, созданные на пакетном уровне, и выполняет индексацию для обеспечения запросов с низкой задержкой. Этот уровень реализуется с использованием HBase, Cassandra, Apache Druid или другого аналогичного хранилища данных.

Кроме результатов с низкой задержкой для уровня реального времени, у этой архитектуры есть и другие преимущества, не в последнюю очередь то, что входные данные в основном наборе данных остаются неизменными. Это позволяет обрабатывать данные при изменении кода и бизнес-правил.

Самый важный недостаток этой архитектуры состоит в том, что приходится поддерживать две отдельные базы кода: одну для реального времени и одну для уровней пакетной обработки. Эта сложность связана с гораздо более высокими

затратами на обслуживание и риском проблем с целостностью данных, если две базы кода не будут постоянно синхронизированы. Появились фреймворки, способные компилировать код как на уровне реального времени, так и на уровне пакетной обработки. Кроме того, есть сложности с эксплуатацией и обслуживанием обеих систем.

Надо признать, что с момента появления лямбда-архитектуры обработка в реальном времени значительно продвинулась вперед. В современных потоковых системах уже нет причин, по которым нельзя было бы дать такие же сильные семантические гарантии, что и в пакетных процессах, не жертвуя при этомзадержкой.

Каппа-архитектура

Исходная концепция каппа-архитектуры (рис. 12.2) была впервые описана Джеем Крепсом (Jay Kreps), когда он работал в LinkedIn. В лямбда-архитектуре для хранения данных используется реляционное хранилище данных или хранилище NoSQL. В каппа-архитектуре хранилище данных представляет собой неизменяемый журнал, предназначенный только для добавления данных, такой как Kafka. Уровень обработки в реальном времени передает данные через вычислительную систему и направляет их во вспомогательные хранилища для обслуживания. Каппа-архитектура исключает систему пакетной обработки, ожидая, что система потоковой передачи сможет выполнить все преобразования и вычисления.

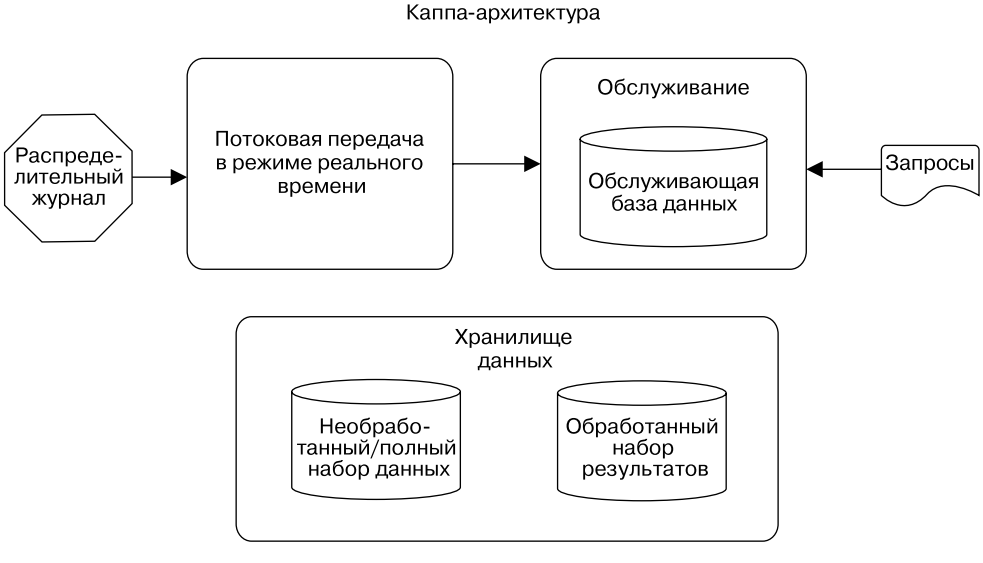


Рис. 12.2. Каппа-архитектура

Одно из самых больших достоинств каппа-архитектуры — снижение сложности и эксплуатационных расходов, по сравнению с лямбда-архитектурой, за счет исключения уровня пакетной обработки. Каппа-архитектура также предназначена для уменьшения сложностей при миграциях и реорганизациях. Если вы захотите повторно обработать данные, то сможете запустить повторную обработку, протестировать ее и переключиться на нее.

Лямбда- и каппа-архитектура — это примеры паттернов, которые можно использовать для обработки и представления больших объемов данных в режиме реального времени. Далее мы рассмотрим некоторые типовые архитектурные решения, основанные на данных и созданные как альтернативы традиционному подходу к приложениям, которые обращаются к своим хранилищам напрямую.

Порождение событий

Порождение событий — это архитектурный паттерн, который полностью меняет способ извлечения и вставки сохраненных данных. В этом случае уровень абстракции хранилища данных снижается, что обеспечивает гибкость при создании и восстановлении представлений данных.

Архитектурный паттерн порождения событий предполагает, что изменения объектов сохраняются в виде последовательности изменений состояния. Когда состояние изменяется, в журнал добавляется новое событие. В традиционном хранилище данных изменения деструктивны — предыдущее состояние заменяется текущим. В модели порождения событий, поскольку все изменения записываются, текущее состояние приложения может быть восстановлено путем воспроизведения событий из журнала. Такое хранилище данных называется *хранилищем событий*.

Это нечто большее, чем просто новый журнал для изменений. Порождение событий и распределенные журналы — это новый паттерн моделирования данных. Порождение событий дополняет традиционное хранилище, такое как реляционное или хранилище типа «ключ — значение», предоставляя более низкий уровень хранения данных, оперирующий событиями, а не значениями с состоянием, которые могут быть перезаписаны.

Хранилище событий функционирует не только как распределенный журнал событий и база данных записей, но и как система сообщений, как было показано ранее в этой главе. Последующие процессы и рабочие потоки могут на них подписаться. Когда сервис сохраняет событие в хранилище событий, оно доставляется всем заинтересованным подписчикам. Хранилище событий можно реализовать в виде реляционного или NoSQL-хранилища данных либо в распределенном журнале, таком

как Kafka. Существует даже специальное хранилище событий EventStore — проект с открытым исходным кодом, в котором хранятся неизменяемые записи и которое рассчитано только для добавления данных. Выбор будет во многом определяться скоростью изменения и тем временем, в течение которого все события должны храниться до следующего моментального снимка и сжатия.

Порождение событий имеет ряд преимуществ. В отличие от деструктивных изменений данных здесь достаточно провести аудит жизненного цикла объекта. Такое решение также значительно облегчает отладку и тестирование. Даже если кто-то случайно удалит таблицу или фрагмент данных в хранилищах событий, все равно останется распределенный журнал, по которому можно воссоздать все таблицы или одну из них для определенной сущности (entity). Однако здесь есть ряд проблем. Не в последнюю очередь это управление изменениями схемы сущностей, потому что изменение может сделать недействительными предыдущие сохраненные события. Кроме того, может оказаться сложным воссоздавать внешние зависимости при воспроизведении потока событий.

Благодаря преимуществам порождения событий данный паттерн может найти применение во многих магазинах, несмотря на то что в их приложениях вместо хранилища событий все еще используются более традиционные хранилища данных.

CQRS

Существует подход, отражающий естественный переход от использования хранилища событий в качестве вспомогательной абстракции хранилища к использованию хранилища событий в качестве основного уровня хранения данных. Это CQRS — *разделение ответственности на команды и запросы* (Command-Query-Responsibility Segregation). В основе CQRS лежит идея о том, что одни и те же данные могут быть представлены для потребления с использованием нескольких моделей или представлений. Такая необходимость в различных моделях исходит из концепции, что в разных предметных областях разные цели и эти цели требуют разных контекстов, языков и в итоге разных представлений данных.

CQRS можно реализовать, используя порождение событий. Распределенный журнал изменений состояния событий позволяет рабочим процессам, подписавшимся на эти события, создавать эффективные представления, которые затем можно использовать. CQRS может также обеспечивать некоторые другие полезные варианты поведения. Вместо того чтобы просто создавать новые представления, можно создавать представления этих данных в разных хранилищах,

оптимизированных для тех шаблонов запросов, которые их используют. Например, если данные — это текст, который вы хотите найти, то, поместив его в поисковую систему, например в ElasticSearch, для одного представления, можно получить оптимизированное представление для приложения поиска. Можно также создать независимые паттерны масштабирования для каждой совокупности данных. Применяя для запросов хранилища данных, оптимизированные для чтения, и журнал только для добавления данных, оптимизированный для операций записи, вы сможете эффективно использовать CQRS для распределения и оптимизации рабочих нагрузок.

Однако для этой архитектуры есть вероятность излишнего усложнения. Вполне возможно выделить данные, для которых требуется только одно представление, или разделить их на большее количество представлений, чем это необходимо. С целью ограничения сложности необходимо применять такой механизм только для тех данных, которые действительно требуют комплексного многостороннего подхода.

Если сделать так, чтобы записи или команды возвращали достаточно данных для эффективного поиска номеров новых версий своих моделей, также можно снизить сложность приложения. Удобно, если команды возвращают сообщение об успешном или неудачном завершении, список ошибок и номер версии, который можно использовать для получения итоговой версии модели. Можно даже возвращать данные как часть команды — это может не вполне соответствовать теории CQRS, зато сильно упростит всем жизнь.

Не стоит смешивать CQRS с паттерном порождения событий. Порождение событий как основной механизм хранения данных довольно сложен. Важно убедиться, что таким образом представлены только те данные, которые хранятся в виде наборов изменений состояния. CQRS можно реализовать используя представления, флаги базы данных и внешнюю логику или любым другим способом, более простым, чем собственно паттерн порождения событий.

Итак, мы просто перечислили здесь архитектуры, управляемые данными, с которыми удобно работать или проектировать приложения. Ключевой особенностью каждой из них является определение жизненного цикла данных и поиск эффективного хранилища и способа передачи для пересылки этих данных каждому компоненту системы. Данные могут быть представлены любым количеством способов, и в большинстве современных организаций наверняка придется реализовать несколько моделей представления данных при сохранении целостности основного набора данных.

Резюме

Надеемся, что после рассмотрения этих примеров вы отметили себе некоторые возможности для дальнейшего расширения функциональности, увеличения доступности, масштабирования и повышения производительности своих хранилищ данных. Есть множество возможностей погрязнуть в сложных архитектурах, обслуживание которых требует огромных затрат. Наихудший случай, конечно же, — это потеря целостности данных, которая может стать реальной проблемой на протяжении всей вашей карьеры.

Завершая эту главу, мы подходим к концу книги. В последней главе мы соберем все вместе и приведем некоторые рекомендации относительно того, как вы можете продолжать совершенствовать свои знания и повышать культуру обеспечения надежности баз данных в своих организациях.

13 Как обосновать и внедрить DBRE

На протяжении всей книги мы пытались показать, как со временем менялся инжиниринг баз данных. Мы перечислили области эксплуатации и разработки, в которые должен быть вовлечен инженер по обеспечению надежности баз данных (DBRE), и рассказали о том, с чего начать. Наконец, мы попытались рассказать вам о современных экосистемах хранения и репликации, о хранилищах данных и архитектурах, чтобы расширить ваш кругозор.

Мы уверены, что есть насущная необходимость уделить особое внимание надежности как одной из сфер ответственности DBR-инженера, ведь база данных — это то место, где риск и хаос просто недопустимы. Многое из того, что сейчас стало обычным явлением в нашей повседневной работе, — виртуализация, инфраструктура как код, контейнеры, бессерверные вычисления и распределенные системы — возникло в тех вычислительных областях, где риск был допустимым. Теперь, когда эти технологии используются повсеместно, специалисты, которые управляют одним из самых ценных ресурсов организации — данными, — должны найти способы включить базы данных в эти парадигмы.

Значительная часть этой работы все еще относится к области скорее желаемого, чем действительного. Любая организация очень дорожит своими данными и не хочет ими рисковать. Таким образом, то, как мы преподносим эти концепции остальной части организации или как реагируем на действия других, — зависит только от нашего профессионального уровня. Недостаточно иметь видение и намерение — чтобы достичь успеха, необходимо найти способы представить это видение другим.

В этой главе мы расскажем, как ввести культуру обеспечения надежности баз данных в вашей организации сейчас и в будущем. В процессе рассмотрения и обсуждения DBRE к вам будет приходить понимание, в каких направлениях и как это можно применить к широкому кругу решаемых задач.

Культура обеспечения надежности баз данных

Что такое культура обеспечения надежности баз данных и как ее внедрять?

Есть много процессов, которые, как считается, относятся к культуре обеспечения надежности, однако при этом не относятся к базам данных как таковым, в том числе:

- создание безобвинительных (не предусматривающих поиска виновных) отчетов о проблемах («постмортемов»);
- автоматизация повторяющейся, рутинной работы;
- структурированное и рациональное принятие решений.

Все это действительно имеет смысл, и каждый сотрудник организации, занимающийся эксплуатацией или обеспечением надежности информационных систем (SRE), должен постоянно над этим работать. Но на что мы, как DBRE-специалисты или люди, которые хотят ими стать, должны обратить особое внимание в своей деятельности? В этой главе мы рассмотрим подходы, которые позволят нам быть вовлеченными в процесс, внедрять культуру обеспечения надежности и донести знания в области DBRE до остальных работников организации.

Разрушение барьеров

Администратор баз данных, который держится обособленно от других специалистов, взаимодействующих с хранилищем данных, попросту не может работать успешно. Чтобы быть эффективными, нам необходимо быть активными членами команды и партнерами. Это неизбежно связано с той проблемой, что роль базы данных обычно не очень понятна. На протяжении всей книги мы подчеркивали, что просто невозможно в полной мере донести понимание роли базы данных до всех разработчиков и инженеров по эксплуатации, которые будут с ней работать.

В некоторых областях DBRE-специалист может оказаться весьма эффективным членом универсальной команды. Каждый раз, когда DBR-инженера привлекают к смежным с его основными обязанностями работам, ему необходимо поставить цель — предоставить экспертные знания в его области, устранить ограничения на ресурсы DBRE или узнать больше о работе других, чтобы все могли лучше выполнять задачи, поставленные в организации.

Процесс проектирования архитектуры

Разумеется, люди с большим опытом работы с базами данных должны активнее принимать участие в процессе проектирования архитектуры на всех уровнях, особенно на этапе создания структуры системы. DBRE-специалист может предоставить исключительно ценную информацию о том, как выбрать правильное

хранилище данных, желательно такое, которое уже было хорошо протестировано и апробировано. Как было изложено в главах 7 и 11, DBRE-специалист должен хорошо разбираться в хранилищах данных, которые могли бы быть введены в эксплуатацию в организации.

В более крупных организациях, которым требуется самообслуживание для создания и развертывания сервисов, DBRE-специалист обладает особыми полномочиями по определению списка сервисов хранения, переданных на самообслуживание. Сотрудничая с техническими отделами, DBR-инженер может помочь поддержать утвержденный порядок использования сервисов из этого списка, работоспособность которых была подтверждена на множестве тестов нагрузочной способности, граничных случаев, масштабирования, надежности и целостности данных. Порой он также может допустить самостоятельное создание и развертывание сервисов, не входящих в этот список (и, соответственно, не проходивших строгие проверки) или находящихся в этом каталоге на другом уровне, но при условии изменения соглашений об уровне обслуживания (SLA). Рассмотрим примеры.

- *Уровень 1.* Хранилище протестировано для базовых сервисов в реальной производственной среде. Использование паттернов самообслуживания означает, что специалисты по эксплуатации и DBR-инженеры могут обеспечить SLA о не более чем 15-минутном интервале реагирования на критические ошибки и наиболее неотложные проблемы[1] и гарантируют максимальные SLA по доступности, задержке, пропускной способности и устойчивости. Реагирование может состоять в решении проблемы или (чаще) ее распознавании и перенаправлении соответствующим специалистам (escalation).
- *Уровень 2.* Хранилище протестировано в производственной среде для некритических сервисов. Это означает, что специалисты по эксплуатации и DBRE смогут обеспечить 30-минутные SLA времени реагирования на события Sev1 и гарантируют сохранение работоспособности, но со снижением целевых показателей качества обслуживания (SLO) по доступности, задержке, пропускной способности и устойчивости.
- *Уровень 3.* В производственной среде хранилище не тестировалось. Специалисты по эксплуатации и DBRE обеспечивают реагирование настолько оперативно, насколько это возможно, но никаких гарантий SLO не дается. Хранилище должно полностью поддерживаться командой разработки программного обеспечения.

Если в организации не поддерживаются подобные технологии самообслуживания, то DBRE-специалист должен приложить больше усилий к тому, чтобы все специалисты знали о тех методах, которые используются при выборе хранилищ данных и архитектур, и о том значении, которое имеет этот выбор. Несмотря на

[1] Severity 1 error, *Sev1. — Примеч. науч. ред.*

то что преждевременная оптимизация всегда рискованна, одна из самых ценных вещей, которые предоставляет DBRE, — это гарантии того, что выбор архитектуры хранилища данных не помешает будущему развитию сервиса, если однажды это хранилище достигнет критической точки масштабирования.

Во многих организациях любой проект начинается с составления обязательного контрольного списка (технического задания), который включает в себя также обзор базы данных задолго до того, как проект будет запущен в промышленную эксплуатацию. Однако в реальной жизни с таким отношением к проекту вы можете слишком долго ждать, и в какой-то момент может оказаться просто поздно что-либо менять. В этот момент важно искусно проявить дипломатию. Нужно найти время, чтобы оценить возможности разных хранилищ данных и донести до остальных рекомендуемые методы, компромиссы и модели для наиболее распространенных хранилищ данных или тех хранилищ, которые будут использоваться в будущем в организации, — это важный способ показать всем, в чем польза от DBRE-специалиста. Это также отличный шанс поработать с разработчиками и архитекторами, собрать и эти данные и сделать их доступными для того, чтобы привлечь к ним больше внимания. Если делать это каждый раз, когда в обсуждении архитектуры появляется новое хранилище данных, которое вы еще не рассматривали, то ваши коллеги будут охотнее вкладывать ресурсы в оценку хранилища данных как части проекта, обращаясь к вам как куратору и консультанту. Главное, чтобы люди увидели в привлечении DBRE пользу, а не ограничения.

Существуют показатели, которые можно использовать, чтобы измерить, в правильном ли направлении движутся DBRE-специалисты и организация в целом. Вот несколько примеров таких показателей.

- В каком количестве архитектурных проектов участвовали DBRE-специалисты или использовались утвержденные ими паттерны и типовые решения? *(Постмортем.)*
- Какое хранилище было использовано и развернуто? *(Постмортем.)*
- Сколько часов работы DBRE-специалиста затрачено на каждом этапе или суммарно в ходе формирования требований к системе? *(Постмортем.)*
- Показатели доступности, пропускной способности и задержки, сгруппированные по уровням и механизмам хранения для обеспечения надежности.

Разработка базы данных

Как и в случае с архитектурой, DBRE-специалисты, участвующие в разработке баз данных с самых ранних этапов цикла проектирования, могут стать фактором, способствующим успеху проекта. Мы подробно обсуждали эту возможность и пользу от такого участия в главе 8. Одно из самых больших препятствий для этого — разработчики, которые забывают обсуждать свои проекты с DBRE. Кроме

того, разработчики могут посчитать, что им такая помощь не нужна. Одна из самых больших побед в этой борьбе — помочь команде разработчиков увидеть пользу от работы, выполняемой совместно с DBRE-специалистами.

Задействование DBRE либо на полный рабочий день, либо только в определенные моменты в жизненном цикле проекта; работа совместно с разработчиками программного обеспечения — это прекрасная возможность помочь программистам увидеть пользу от взаимодействия с DBRE. Даже если DBR-инженер не особенно силен в написании кода, его вклад в построение моделей данных, создание доступа к базе данных и использование функций может оказаться неоценимым, а также он может помочь выстроить отношения между подразделениями. Сотрудничество разработчиков с DBRE-специалистами, которые выполняют оценку и реализацию приложений, управляемых данными или в любой момент доступны для предоставления помощи, может также способствовать развитию отношений, взаимопониманию, распространению межгрупповых знаний, что ускоряет разработку.

Мы также обсудили важность распространения рекомендованных методов и типовых решений для функций, которые создает разработчик и которые требуют интеграции с уровнем данных. Например, можно составить контрольные списки для используемых моделей, запросов или функций, в отношении которых программист должен указывать, использовал ли он определенные паттерны и типовые решения. Эти контрольные списки могут сигнализировать о требованиях, сценариях или функциях, которые, возможно, придется пересмотреть, прежде чем передавать продукт в промышленную эксплуатацию.

Существуют показатели, которые можно использовать для оценки того, насколько DBRE-специалисты и организация в целом продвинулись в этом направлении. Вот некоторые их примеры:

- количество часов, в течение которых программисты и DBR-инженеры вели совместную разработку;
- требования и пользовательские сценарии, в формировании которых участвовали DBR-инженеры;
- привязанность таких показателей функционирования, как задержка и устойчивость, к использованию DBRE;
- дежурства совместно с разработчиками.

Миграции рабочей среды

Спору нет — все хотят, чтобы миграции проходили без ошибок. Но нередко частота новых развертываний превышает время, необходимое DBRE-специалисту для подготовки и поддержки процесса развертывания. В итоге незавершенные задания объединяются в большие наборы изменений, хрупкие и нестабильные,

что может повлечь значительный риск, или же миграции выполняются без тщательного контроля со стороны DBRE. Как обсуждалось в главе 8, эффективный способ справиться с этой ситуацией состоит в том, чтобы создать процесс и инструменты, позволяющие разработчикам сделать оптимальный выбор: что может быть реализовано с помощью обычных механизмов развертывания, что следует поручить реализовать DBRE-специалистам, а что DBRE-специалисты должны проверить в случае сомнений.

Самый простой первый шаг — постепенно создать библиотеку эвристик, которые бы указывали, какие изменения являются безопасными, а какие — нет. При этом узкое место не исчезнет сразу, однако создание соглашений для сборки такой библиотеки совместно с DBRE-специалистами положит начало процессу. Периодический пересмотр и проверка результатов применения эвристик и рекомендаций, а также анализ успешных или неуспешных завершений изменений могут оказаться эффективным средством контроля данного процесса. Это можно сделать частью обычного послеаварийного анализа изменений, как успешных, так и неудачных.

Еще один способ постепенно увеличивать автономность отделов разработки — построить базу данных паттернов миграции, которые могут быть эвристически применены к предстоящим изменениям. Если делать это во взаимодействии с разработчиками, то со временем, работая совместно с ними над изменениями, вы сможете создать постоянно обновляемый документ, который разработчики будут не только использовать, но и сами дорабатывать. Здесь, как всегда, решающее значение имеет выполнение послеаварийного анализа и обзоров (постмортемов) для подтверждения успешности таких моделей.

Эту идею можно развить дальше, определив ограничения для реализаций, которые, согласно эвристикам и паттернам миграции, могут быть выполнены разработчиками. Эти ограничения дадут уверенность всем — разработчикам, службе эксплуатации, DBR-инженерам и руководству. Чем больше успеха вы будете демонстрировать и чем сильнее укрепите доверие, тем дальше это может зайти. Предоставив разработчикам самостоятельность, вы, вероятно, обнаружите, что ваши отношения с этими специалистами станут гораздо более здоровыми, поскольку вы докажете, что такие отношения приносят скорее пользу, чем являются препятствием. Продолжая делиться своими знаниями в области хранения данных и доступа к базам данных, вы повысите их эффективность и улучшите взаимодействие между вами. Это можно сделать индивидуально, работая в паре, или организовав обучение, например в форме семинаров, обмена знаниями или с помощью документов.

Никакая активизация общения не изменит тот факт, что некоторые миграции придется выполнять DBRE-специалисту. Но и в этом случае существуют подходы, которые можно использовать для обучения и поддержки других работников. Опять

же, отличным вариантом в этом случае может быть сотрудничество с инженерами в процессе планирования и выполнения миграции. Кроме того, может быть очень полезным сотрудничество с инженерами по эксплуатации, поскольку чем больше людей сможет помочь и в итоге выполнить сложные промышленные реализации, тем лучше.

Даже без значительной автоматизации у вас и команды DBRE есть множество способов вносить изменения более надежно и безошибочно, не останавливая конвейер разработки. После того как будет получено доверие и достигнута хорошая повторяемость ручных процессов, вы можете продвинуться дальше, вводя новые технологии, инструменты и код.

Существуют показатели, которые можно использовать для оценки того, насколько хорошо работают в этом направлении отдельные DBRE-специалисты и подразделение в целом. Вот несколько примеров таких показателей:

- количество часов совместной работы программистов, специалистов по эксплуатации и DBRE в ходе миграции;
- процент миграций, требующих вмешательства DBRE;
- количество успешных и неуспешных миграций и их влияние.

Проектирование и развертывание инфраструктуры

В разделе, посвященном архитектуре, мы обсудили работу с разработчиками по выбору проверенных и надежных хранилищ данных. Точно так же необходимо постоянно работать с персоналом по эксплуатации и инфраструктуре, чтобы у этих специалистов было все нужное не только для размещения хранилищ данных, но и для их развертывания и обслуживания. В главе 5 мы подробно обсудили различные стороны и этапы этой работы, а в главе 6 — программное обеспечение и инструменты, необходимые для управления этими инфраструктурами при масштабировании. Но мы все еще находимся на ранних стадиях этого процесса для хранилищ данных, особенно распределенных.

Как и в случае с производственными реализациями и предоставлением инженерам-программистам большей автономии, основная часть внедрения таких отношений и процессов сводится к постепенному, поэтапному формированию доверия. Первые шаги, которые могут принести значительную пользу команде DBRE и всей организации, — это использование единых репозиториев и систем управления версиями для хранения сценариев, конфигураций и документации. Затем можно поработать с командой специалистов по эксплуатации, чтобы начать настройку и развертывание пустых хранилищ данных с использованием средств управления конфигурацией и оркестрации. Впоследствии вам предстоит еще заполнить эти хранилища реальными данными, но и это — уже шаг вперед.

При этом, сотрудничая со специалистами по эксплуатации, можно проверить корректность конфигурации, протестировать безопасность, выполнить нагрузочное тестирование и даже более сложные тесты на целостность и репликацию данных. Чем глубже вы ознакомите всю команду с тем, как работают базы данных и как они ломаются, тем лучше. Проверка доступности и поведения при сбоях также является критически важным тестом, который может потребоваться и другим подразделениям.

Наконец, можно начать ставить задачи оперативного реагирования персоналу по эксплуатации и даже ведущим разработчикам по управлению их инфраструктурой. Благодаря тому что вы и ваша команда зеркально отражаете их и сотрудничаете с ними, они могут быстро обрести уверенность в работе с этими инфраструктурами и свести риски к минимуму. Только когда команда по-настоящему будет уверена в том, что знает все изнутри и снаружи, можно начать автоматизировать более рискованные компоненты, такие как загрузка данных, изменение репликации и отработка отказов ведущего узла.

Существуют следующие показатели, которые можно использовать для оценки того, насколько хорошо работают в этом направлении отдельные DBRE-специалисты и отдел в целом:

- количество компонентов инфраструктуры, которые настраиваются путем управления конфигурацией;
- количество компонентов инфраструктуры, интегрированных в платформы оркестрации;
- количество успешных и неудачных инициализаций;
- показатели потребления ресурсов — по всем подсистемам, используемым хранилищами данных;
- доля дежурных смен под руководством не DBRE-специалистов;
- инциденты, не управляемые DBRE-специалистами, и среднее время восстановления (MTTR);
- количество инцидентов, потребовавших обращения к DBRE.

Значительная часть успеха этой работы заключается в налаженных взаимоотношениях, взаимопонимании, доверии и обмене знаниями. Мы знаем, что многие администраторы баз данных привыкли работать изолированно, но теперь вы и ваша команда сможете вынести свою работу над базами данных на свет. Больше не должно быть малопонятной и пугающей работы, которой готовы заниматься только самые смелые или самые глупые инженеры. Ключевым условием для этого является постоянная открытость, наращивание доверия и взаимодействие с другими службами.

Принятие решений на основе данных

Доверие невозможно сформировать, не имея исчерпывающей информации о последствиях изменений. Цикл Деминга для планирования, выполнения, проверки и действия требует наблюдаемости и контролируемости, что мы рассмотрели в главе 4. Прежде чем собирать базовые показатели при каждом изменении, не забудьте четко сформулировать те их них, которые служат для определения успешности процедуры, а затем наконец выделите время на тщательный анализ результатов.

Использование знаний о SLO организации, как было показано в главах 2 и 3, имеет решающее значение для понимания изменений, которые необходимо внести, а также показателей и результатов, позволяющих показать остальным потенциальную пользу от изменений и значимость прикладываемых усилий.

Надеемся, вы работаете в организации, которая уже осознала важность принятия решений на основе данных и, таким образом, уже внедрила платформу для мониторинга, процессы анализа, а также дисциплину, которой она неуклонно следует. Мы также надеемся, что в вашей организации уже определены четкие адекватные SLO для принятия решений. Но если все не так, то вам нужно начать именно с этого, чтобы иметь возможность продвигаться к более глубоким и потенциально более далекоидущим изменениям.

Целостность и возможность восстановления данных

В главе 7 мы обсудили важность обеспечения целостности данных и возможности их восстановления после потери или повреждения. Слишком часто организации рассматривают это как обязанность DBRE-специалистов, но мы знаем, что силами одних только DBRE эта задача невыполнима. DBR-инженеры неизменно оказываются чемпионами по количеству выпадающих на них задач по обеспечению целостности данных. Ваша постоянная ответственность — убедить разработчиков в важности выделения ресурсов для создания технологических процессов проверки данных и API для восстановления. И если вы счастливо преодолеете барьеры между архитектурой, разработкой программного обеспечения и неучастием DBRE в ранних этапах разработки, то обнаружите, что построили прочные отношения и установили доверие с разработчиками и можете постепенно создавать общую базу кода и знаний, необходимых для реализации эффективного процесса обеспечения целостности данных.

Убедить их будет не просто. Как показывает наш опыт, большинство разработчиков считают, что целостность данных — это зона ответственности исключительно DBRE-специалистов. И некоторые подразделения будут препятствовать разработке процессов валидации и API восстановления. Тогда вам придется искать решения «для бедных», которые будут лишь собирать сведения о проблемах

целостности данных. Аналогично оценка и фиксация усилий и затрат на ручное восстановление данных может стать весомым аргументом для руководства, чтобы начать выделять ресурсы для разработки и внедрения API восстановления и процессов валидации.

Как видите, успешное движение в сторону повышения надежности баз данных требует постепенных и комплексных организационных изменений. Выбор направлений, в которых ваши усилия принесут наибольшую пользу для всех, — это навык, требующий здравого смысла и практики. Последующие «точечные» изменения, приводящие к последовательным улучшениям и способствующие укреплению доверия, придадут импульс дальнейшему движению. Но все это требует времени, доверия и множества экспериментов в отношении того, что лучше сработает для тех уровней риска, которые характерны для вашей организации.

Резюме

Мы хотели бы поблагодарить вас за то, что вы нашли время прочитать эту книгу. Мы обе очень увлечены построением карьеры в одной из самых утомительных и неблагодарных областей техники. Несмотря на то что значительная часть этой книги рассчитана на перспективу и все еще проходит проверку в реальных условиях, мы уверены, что продвижение DBRE может принести огромную пользу сервисам, управляемым данными, и занимающимся ими отделам.

Надеемся, что вы вдохновитесь на изучение этих процессов в вашей организации и будете жаждать узнать больше. Мы постарались предоставить вам дополнительные источники для чтения и изучения, поскольку эти подходы и технологии весьма гибки и подвижны. Но самое главное — мы надеемся, что помогли вам увидеть возможность перенести проверенную временем роль DBA в современный мир и в будущее. Роль DBA не исчезает и, независимо от того, являетесь ли вы в этой области новичком или опытным ветераном, мы хотим, чтобы у вас впереди была долгая карьера, поскольку вы принесете пользу любой организации, частью которой являетесь.

Об авторах

Лейн Кэмпбелл (Laine Campbell) работает старшим управляющим (Senior Director) по проектированию в компании Fastly. Она также была основателем и генеральным директором PalominoDB/Blackbird — консалтинговой службы, обслуживающей базы данных ряда компаний, включая Obama for America, Activision Call of Duty, Adobe Echosign, Technorati, Livejournal и Zendesk. Она имеет 18-летний опыт эксплуатации баз данных и масштабируемых распределенных систем.

Черити Мейджорс (Charity Majors) является генеральным директором и соучредителем компании honeycomb.io. Сочетая в себе точность агрегаторов журналов, метрики скорости временных рядов и гибкость APM (Application Performance Metrics), honeycomb представляет собой первый в мире действительно аналитический сервис нового поколения. Ранее Черити была специалистом по эксплуатации в Parse/Facebook, управляя огромным парком наборов реплик MongoDB, а также Redis, Cassandra и MySQL. Она также тесно сотрудничала с командой RocksDB в Facebook, участвуя в разработке и развертывании первой в мире установки Mongo + Rocks с использованием API подключаемого модуля хранения.

Об обложке

Животное, изображенное на обложке книги, — это суффолк, или суффолкская лошадь, или суффолкская гнедая. Эта английская порода ломовых лошадей всегда имеет гнедую масть.

Суффолк был выведен в XVI веке для сельскохозяйственных работ. Породу активно разводили в начале XX столетия, однако в середине века она потеряла популярность из-за механизации сельского хозяйства. Суффолки имеют рост от 1,65 до 1,75 метра и весят около 900 килограммов; они невысоки, но более массивны, чем лошади других британских племенных пород, таких как клайсдейлы или шайры.

Многие животные, изображенные на обложках книг издательства O'Reilly, находятся под угрозой исчезновения, однако все они важны для мира. Чтобы больше узнать о том, как им можно помочь, посетите сайт animals.oreilly.com.

Лейн Кэмпбелл, Черити Мейджорс

Базы данных. Инжиниринг надежности

Перевела с английского *Е. Сандицкая*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*С. Сиротко*
Литературный редактор	*Н. Рощина*
Художественный редактор	*А. Михеева*
Корректоры	*Е. Павлович, О. Андриевич*
Верстка	*Г. Блинов*

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: 03.2020. 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 26.02.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 24,510. Тираж 700. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87