O'REILLY®

Распределенные данные

Алгоритмы работы современных систем хранения информации

Алекс Петров

Database Internals

A Deep Dive into How Distributed Data Systems Work

Alex Petrov

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Распределенные данные

Алгоритмы работы современных систем хранения информации

Алекс Петров

ПИТЕР®

Санкт-Петербург • Москва • Минск

2021

Алекс Петров

Распределенные данные. Алгоритмы работы современных систем хранения информации

Серия «Бестселлеры O'Reilly»
Перевел на русский А. Коцюба

Руководитель проекта	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Научный редактор	*А. Коцюба*
Литературный редактор	*А. Руденко*
Обложка	*В. Мостипан*
Корректоры	*Н. Петрова, М. Одинокова*
Верстка	*Е. Неволайнен*

ББК 32.973.233-018
УДК 004.65

Петров Алекс

П30 Распределенные данные. Алгоритмы работы современных систем хранения информации. — СПб.: Питер, 2021. — 336 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1640-9

Когда дело доходит до выбора, использования и обслуживания базы данных, важно понимать ее внутреннее устройство. Как разобраться в огромном море доступных сегодня распределенных баз данных и инструментов? На что они способны? Чем различаются? Алекс Петров знакомит нас с концепциями, лежащими в основе внутренних механизмов современных баз данных и хранилищ. Для этого ему пришлось обобщить и систематизировать разрозненную информацию из многочисленных книг, статей, постов и даже из нескольких баз данных с открытым исходным кодом. Вы узнаете об принципах и концепциях, используемых во всех типах СУБД, с акцентом на подсистеме хранения данных и компонентах, отвечающих за распределение. Эти алгоритмы используются в базах данных, очередях сообщений, планировщиках и в другом важном инфраструктурном программном обеспечении. Вы разберетесь, как работают современные системы хранения информации, и это поможет взвешенно выбирать необходимое программное обеспечение и выявлять потенциальные проблемы.

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

ISBN 978-1492040347 англ. Authorized Russian translation of the English edition of Database Internals
ISBN 9781492040347 © 2020 Alex Petrov
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.

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: 07.2021.
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 02.07.21. Формат 70x100/16. Бумага офсетная. Усл. п. л. 27,090. Тираж 500. Заказ

Краткое содержание

Оглавление

Питеру Хинтьенсу, от которого я получил свою первую подписанную книгу: вдохновляющему программисту распределенных систем, автору, философу и другу

Предисловие к русскому изданию

С детства я говорил на украинском и русском языках. Но ради того, чтобы как можно больше людей узнало о предмете настолько важном, как базы данных, я решил написать эту книгу на английском. Можете представить себе мой восторг, когда я узнал, что издательство «Питер», технические книги которого я помню еще с университетских лет, занимается переводом моей книги на русский.

Мы приложили все усилия для того, чтобы вы не только познакомились с устройством баз данных, но и могли использовать русский перевод книги как основу для дальнейшего изучения этой темы, и снабдили русские термины их английскими эквивалентами для того, чтобы помочь русскоязычному сообществу разработчиков баз данных стандартизировать терминологию.

Предисловие

Распределенные системы управления базами данных являются неотъемлемой частью большинства предприятий и подавляющего большинства программных приложений. Приложения отвечают за логику и взаимодействие с пользователем, в то время как системы управления базами данных (СУБД) обеспечивают целостность, согласованность и избыточность данных.

В 2000 году вы могли использовать лишь несколько разновидностей баз данных, большинство из которых были реляционными и в силу этого мало чем отличались друг от друга. Конечно, это не означает, что эти базы данных были совершенно одинаковыми, однако они имели много сходства в плане функциональности и сценариев использования.

Некоторые из этих баз данных ориентировались на *горизонтальное масштабирование* — повышение производительности и увеличение емкости за счет запуска нескольких экземпляров базы данных, действующих как единый логический блок: Gamma Database Machine Project, Teradata, Greenplum, Parallel DB2 и многие другие. Горизонтальное масштабирование и сегодня остается одним из самых востребованных свойств баз данных, что объясняется растущей популярностью облачных сервисов. Зачастую проще развернуть новый экземпляр и добавить его в кластер, чем производить *вертикальное масштабирование*, перенося базу данных на более крупную и мощную машину. Миграция часто требует больших затрат времени и усилий, что, в свою очередь, может приводить к простоям.

Примерно в 2010 году начал формироваться новый класс баз данных *с конечной согласованностью* и стали набирать популярность такие термины, как *NoSQL* и *большие данные*. За последние 15 лет сообщество разработчиков ПО с открытым исходным кодом, крупные интернет-компании и поставщики баз данных создали так много баз данных и инструментов, что можно легко запутаться, пытаясь разобраться в их специфических особенностях и сценариях использования.

В 2007 году разработчики компании Amazon опубликовали статью с описанием системы хранения данных Dynamo [DECANDIA07], которая произвела столь сильное впечатление на сообщество разработчиков баз данных, что за короткое время на свет появилось множество вариантов и реализаций этой системы. Наиболее известными среди них были такие системы хранения, как Apache Cassandra, разработанная компанией Facebook, Project Voldemort от компании LinkedIn и Riak от бывших разработчиков компании Akamai.

Сегодня ситуация в области систем хранения данных снова меняется: на смену хранилищам типа «ключ–значение», NoSQL и системам с конечной согласованностью стали приходить более масштабируемые и производительные базы данных,

способные выполнять сложные поисковые запросы с более высокой гарантией согласованности.

Кому предназначена эта книга

На конференциях мне часто задают один и тот же вопрос: «Как мне узнать больше о внутреннем устройстве баз данных? Я даже не знаю, с чего начать». Авторы большинства книг по СУБД не вдаются в детали реализации подсистем хранения данных, лишь в общих чертах описывая такие методы доступа, как использование B-деревьев. Поскольку лишь очень немногие книги подробно освещают такие актуальные концепции, как различные сценарии использования B-деревьев и журналированные подсистемы хранения, я обычно рекомендую читать статьи.

Каждый, кто пробовал читать статьи, знает, что это не так просто: вам часто не хватает контекста, формулировки могут быть неоднозначными, между статьями мало или вообще нет связи и их трудно найти. Эта книга содержит краткое описание важных концепций систем управления базами данных и может служить в качестве справочника для тех, кто хотел бы копнуть глубже, или в качестве шпаргалки для тех, кто уже знаком с этими концепциями.

Хотя далеко не все хотят стать разработчиками баз данных, эта книга поможет всем, кто создает программное обеспечение, использующее СУБД: разработчикам, специалистам по обеспечению надежности, архитекторам и руководителям проектных работ.

Если ваша компания использует какой-либо компонент инфраструктуры, будь то база данных, очередь сообщений, контейнерная платформа или планировщик задач, то вам следует читать журналы изменений проекта и списки рассылки, чтобы оставаться на связи с сообществом и знать о последних изменениях. Понимание терминологии и знание внутреннего устройства позволят вам извлекать больше информации из этих источников и более продуктивно использовать свои инструменты для поиска и устранения неполадок, выявления и предотвращения рисков и узких мест. Общее понимание принципов работы СУБД будет существенным подспорьем в том случае, если что-то пойдет не так. Используя эти знания, вы сможете выдвинуть некоторую гипотезу, проверить ее, найти первопричину проблемы и предоставить ее описание другим участникам проекта.

Эта книга также предназначена пытливым умам — людей, которые любят изучать что-то без непосредственной необходимости, тем, кто любит проводить время, взламывая что-то из чистого интереса, создавая компиляторы, собственные операционные системы, текстовые редакторы и компьютерные игры, изучая языки программирования и впитывая новую информацию.

Предполагается, что читатель имеет некоторый опыт разработки серверных систем и работы с СУБД в качестве пользователя. Хорошим подспорьем в усвоении материала книги также будет наличие некоторых знаний о различных структурах данных.

Зачем мне читать эту книгу?

СУБД часто описываются в терминах тех концепций и алгоритмов, которые они реализуют, т. е. указывается, что СУБД «использует gossip-протокол для распространения членства» (см. главу 12), «реализует Dynamo» или «соответствует описанию, предоставленному в статье, посвященной протоколу Spanner» (см. главу 13). Если же разговор идет об алгоритмах и структурах данных, часто можно услышать что-то вроде: «У ZAB и Raft есть много общего» (см. главу 14), «Bw-деревья — это что-то наподобие B-деревьев, реализованных поверх журналированной подсистемы хранения» (см. главу 6), или «они используют одноуровневые указатели, как в B^{link}-деревьях» (см. главу 5).

Чтобы говорить о сложных понятиях, нужны абстракции. Кроме того, невозможно каждый раз заново обсуждать терминологию, начиная какой-либо разговор. Наличие удобного инструмента в виде общего языка помогает переключить внимание на проблемы более высокого уровня.

Одно из преимуществ изучения фундаментальных концепций, доказательств и алгоритмов состоит в том, что они никогда не устаревают. Конечно, всегда будут появляться новые алгоритмы, однако это часто происходит как результат выявления недостатков или возможностей для улучшения в классическом алгоритме. Знание истории помогает лучше понять различия новых алгоритмов и мотивы их создания.

Изучение таких вещей вдохновляет. Вы видите разнообразие алгоритмов, видите, как последовательно решались проблемы в сфере баз данных, и начинаете ценить эту работу. Еще один положительный эффект состоит в том, что разрозненные фрагменты головоломки начинают складываться у вас в голове в цельную картину, которой вы всегда сможете поделиться с другими.

Рассматриваемые темы

Эта книга не посвящена реляционным или нереляционным (NoSQL) СУБД; она посвящена алгоритмам и концепциям, используемым во всех типах СУБД, с акцентом на *подсистеме хранения данных* и на компонентах, отвечающих за распределение.

Некоторые из рассматриваемых здесь концепций, включая, в частности, планирование и оптимизацию поисковых запросов, диспетчеризацию, реляционную модель и т. д., уже освещены в ряде отличных пособий по СУБД. Некоторые из этих концепций обычно описываются с точки зрения пользователя, в то время как в этой книге акцентируется внимание на их внутреннем устройстве. Вы сможете найти ссылки на полезную литературу в конце части II и в заключительных разделах каждой главы. В этих книгах вы найдете ответы на многие из имеющихся у вас вопросов относительно баз данных.

В этой книге не рассматриваются языки запросов, поскольку не существует общего языка, который можно было бы использовать для всех рассматриваемых здесь СУБД.

Чтобы собрать материал для этой книги, я изучил более 15 книг, более 300 статей и бесчисленное количество записей в блогах, файлы исходного кода и документацию по нескольким СУБД с открытым исходным кодом. Главным критерием при отборе рассматриваемых концепций был вопрос о том, говорят ли о той или иной концепции специалисты и ученые в области баз данных? Если ответ был утвердительным, я включал соответствующую концепцию в длинный список рассматриваемых тем.

Структура книги

Есть несколько примеров расширяемых баз данных с подключаемыми компонентами (например, [SCHWARZ86]), но они довольно редки. В то же время существует множество примеров использования базами данных подключаемого хранилища. Точно так же поставщики баз данных редко говорят о выполнении запросов, но всегда готовы поговорить о том, как поддерживается согласованность в предлагаемых ими базах данных.

Наиболее существенные различия между разными СУБД касаются используемых методов *хранения* и методов *распределения* данных. (Иногда играют важную роль и различия в других подсистемах, но мы не будем их рассматривать в этой книге.) Книга разбита на две части, в которых соответственно обсуждаются подсистемы и компоненты, отвечающие за *хранение* (часть I) и *распределение* (часть II).

В **части I** рассматриваются процессы внутри узлов и основное внимание уделяется подсистеме хранения данных — центральному компоненту СУБД и одному из наиболее важных отличительных аспектов. Мы начнем с архитектуры СУБД и рассмотрим несколько способов классификации СУБД в зависимости от среды хранения данных и структуры хранилища.

Затем мы рассмотрим структуры хранения данных и попытаемся понять, чем размещаемые на диске структуры отличаются от структур, размещаемых в оперативной памяти. Мы познакомимся с концепцией B-деревьев и изучим алгоритмы эффективной работы со структурами B-деревьев на диске, включая сериализацию, постраничную компоновку и представление данных на диске. Далее мы обсудим различные варианты концепции B-деревьев, чтобы наглядно увидеть ее мощь и оценить разнообразие структур данных, созданных под ее влиянием.

Наконец, мы обсудим несколько вариантов журналированного хранилища, широко используемых для реализации файловой системы и системы хранения, включая мотивы и причины их использования.

Часть II посвящена объединению нескольких узлов в кластер баз данных. Сначала мы узнаем, почему важно понимать теоретические принципы создания отказоустойчивых распределенных систем, чем распределенные системы отличаются от одноузловых приложений и с какими проблемами, ограничениями и сложностями приходится сталкиваться в распределенной среде.

После этого мы глубоко погрузимся в распределенные алгоритмы. Начнем с алгоритмов обнаружения сбоев, которые помогают повысить производительность и стабильность, выявляя сбои и сообщая о них, а также действуя в обход отказавших узлов. Поскольку для понимания многих алгоритмов, рассматриваемых далее в книге, необходимо понимание концепции «лидера», мы изучим несколько алгоритмов выбора лидера и поговорим о том, в каких случаях их лучше использовать.

Так как одной из самых сложных задач в распределенных системах является обеспечение согласованности данных, мы последовательно рассмотрим такие концепции, как репликация, модели согласованности, возможные расхождения между репликами и конечная согласованность. Поскольку в системах с конечной согласованностью часто применяется механизм антиэнтропии для обеспечения конвергенции, а также gossip-протокол для распространения данных, мы обсудим несколько подходов к их использованию. Завершит эту часть книги рассмотрение логической согласованности в контексте транзакций базы данных и алгоритмов консенсуса.

Было бы невозможно написать эту книгу, не опираясь на исследования и публикации других авторов. По ходу чтения вы встретите множество ссылок на статьи и публикации — они будут заключены в квадратные скобки: например: [DECANDIA07]. Вы можете использовать эти ссылки, чтобы изучить соответствующие концепции более подробно.

В конце каждой главы вы найдете раздел «Итоги», содержащий список дополнительной литературы, имеющей отношение к содержанию главы.

Условные обозначения

В этой книге используются следующие типографские условные обозначения.

Курсивный шрифт

Применяется для выделения новых терминов.

`Моноширинный шрифт`

Используется для записи листингов программ, а также для выделения в тексте таких элементов, как имена переменных и функций, базы данных, типы данных, переменные среды, операторы и ключевые слова.

Так обозначаются подсказки и советы.

Так обозначаются примечания общего характера.

Так обозначаются предупреждения и предостережения.

Благодарности

Это издание не состоялось бы без сотен людей, упорно работавших над научными трудами и книгами, которые стали источниками идей, вдохновения и служили справочниками.

Я хотел бы поблагодарить всех людей, которые рецензировали рукописи и оставляли отзывы, которые помогали убедиться, что материал в этой книге является корректным, а формулировки — точными: Дмитрий Алимов, Тимур Сафин, Питер Альваро (Peter Alvaro), Карлос Бакеро (Carlos Baquero), Джейсон Браун (Jason Brown), Блейк Эгглстон (Blake Eggleston), Маркус Эрикссон (Marcus Eriksson), Франсиско Фернандес Кастаньо (Francisco Fernandez Castano), Хайди Говард (Heidi Howard), Вайдехи Джоши (Vaidehi Joshi), Максимилиан Карась (Maximilian Karasz), Стас Кельвич (Stas Kelvich), Михаил Клишин, Предраг Кнежевич (Predrag Knežević), Джоэл Найтон (Joel Knighton), Евгений Лазин, Нейт Макколл (Nate McCall), Кристофер Мейклджон (Christopher Meiklejohn), Тайлер Нили (Tyler Neely), Максим Неверов, Марина Петрова, Стефан Подковинский (Stefan Podkowinski), Эдвард Рибьеро (Edward Ribiero), Денис Рысцов, Кир Шатров, Алекс Сорокоумов, Массимилиано Томасси (Massimiliano Tomassi) и Ариэль Вайсберг (Ariel Weisberg).

И конечно, эта книга не появилась бы на свет без поддержки моей семьи: жены Марины и дочери Александры, которые поддерживали меня на каждом шагу этого пути.

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.

Пока что не вся терминология, связанная со сферой распределенных данных, устоялась в русскоязычном сообществе. Поэтому часть терминов в этой книге сопровождена их английскими вариантами.

ЧАСТЬ I

Подсистема хранения данных

Основной задачей любой СУБД является надежное хранение данных и обеспечение доступа к ним со стороны пользователей. Мы используем базы данных в качестве основного источника данных, помогающего обмениваться данными между различными частями приложений. Вместо того чтобы искать способ хранения и извлечения информации и изобретать новый способ организации данных при создании каждого нового приложения, мы используем базы данных. Таким образом, мы можем сосредоточиться на логике приложений, а не на инфраструктуре.

Для большей компактности вместо громоздкого термина «система управления базами данных» на протяжении этой книги мы будем в основном употреблять соответствующую аббревиатуру (СУБД) или просто выражение «база данных», имея в виду то же самое.

База данных представляет собой модульную систему, включающую в себя несколько составных частей: транспортный уровень, который принимает запросы, обработчик запросов, который выбирает наиболее эффективный способ выполнения запросов, подсистему выполнения, которая производит выполнение операций, и подсистему хранения (см. раздел «Архитектура СУБД» на с. 25).

Подсистема хранения данных (или ядро СУБД) — это программный компонент СУБД, отвечающий за хранение, извлечение и управление данными в памяти и на диске, предназначенный для работы с постоянной, долговременной памятью каждого узла [REED78]. В то время как базы данных могут отвечать на сложные запросы, подсистемы хранения рассматривают данные более детально и предлагают простой API для манипуляций с данными, позволяющий пользователям создавать, обновлять, удалять и извлекать записи. СУБД можно рассматривать как приложение, которое надстроено поверх подсистемы хранения и предлагает некоторую схему данных, язык запросов, индексацию, транзакции и много других полезных функций.

Для обеспечения гибкости и ключи, и значения могут быть произвольными последовательностями байтов без заданной формы. Их классификация и семантика представления определяются в подсистемах более высокого уровня. Например, вы можете использовать `int32` (32-разрядное целое число) в качестве ключа в одной из

таблиц и `ascii` (строку в кодировке ASCII) — в другой; с точки зрения подсистемы хранения оба ключа являются просто сериализованными записями.

Такие подсистемы хранения, как BerkeleyDB (https://databass.dev/links/92), LevelDB (https://databass.dev/links/93) и ее потомок RocksDB (https://databass.dev/links/94), LMDB (https://databass.dev/links/95) и ее потомок libmdbx (https://databass.dev/links/96), Sophia (https://databass.dev/links/97), HaloDB (https://databass.dev/links/98) и многие другие, были разработаны независимо от тех СУБД, в которые они теперь встроены. Использование существующих подключаемых подсистем хранения позволило разработчикам СУБД обойти этап их создания и сосредоточиться на других подсистемах.

В то же время четкое разделение между компонентами СУБД открывает возможность переключаться между различными подсистемами, потенциально более подходящими для конкретных сценариев использования. Например, MySQL, популярная система управления базами данных, имеет несколько подсистем хранения (https://databass.dev/links/99), включая InnoDB, MyISAM и RocksDB (https://databass.dev/links/100) (в составе MyRocks (https://databass.dev/links/101)). MongoDB позволяет переключаться между подсистемами хранения WiredTiger (https://databass.dev/links/102), In-Memory и (теперь уже устаревшей) MMAPv1 (https://databass.dev/links/103).

Сравнение баз данных

Выбор той или иной СУБД может иметь долгосрочные последствия. Если есть вероятность того, что база данных не подойдет из-за проблем с производительностью, обеспечением согласованности или выполнением операций, лучше выяснить это на раннем этапе цикла разработки, так как миграция на другую систему может оказаться нетривиальной. В некоторых случаях для этого потребуются существенные изменения в коде приложения.

У каждой СУБД есть свои сильные и слабые стороны. Для снижения риска дорогостоящей миграции можно потратить некоторое время перед принятием решения о выборе конкретной базы данных, чтобы быть уверенным в ее соответствии потребностям вашего приложения.

Попытка сравнить базы данных исходя из их компонентов (например, какую подсистему хранения они используют, каким образом данные совместно используются, реплицируются и распределяются и т. д.), их рейтинга (субъективная оценка, сделанная консалтинговыми агентствами, такими как ThoughtWorks (https://www.thoughtworks.com/de/radar), или сайтами со сравнением баз данных, такими как DB-Engine (https://db-engines.com/de/ranking) или Database of Databases (https://dbdb.io/)) или языка реализации (C++, Java или Go и т. д.) может привести к неверным и поспешным выводам. Такие методы можно использовать только для предварительного сравнения, и они могут быть такими же грубыми, как выбор между HBase и SQLite, поэтому даже поверхностное понимание того, как работает каждая база данных и что находится внутри нее, поможет принять более взвешенное решение.

Каждое сравнение должно начинаться с четкого определения цели, потому что даже малейшая предвзятость может полностью свести на нет все исследование. Если вам нужно, чтобы база данных хорошо подходила для имеющихся у вас (или ожидаемых) рабочих нагрузок, то лучше всего будет смоделировать эти рабочие нагрузки для различных СУБД, измерить важные для вас показатели производительности и сравнить результаты. Некоторые проблемы, особенно когда речь заходит о производительности и масштабируемости, начинают проявляться только через некоторое время или по мере роста емкости. Для выявления потенциальных проблем лучше всего провести длительные тесты в среде, максимально точно имитирующей реальное рабочее окружение.

Моделирование реальных рабочих нагрузок не только помогает понять, как работает база данных, но и позволяет научиться ее использовать и отлаживать, а также выяснить, насколько дружелюбно и полезно сложившееся вокруг нее сообщество. Выбор базы данных всегда определяется сочетанием этих факторов, и производительность часто оказывается не самым важным аспектом: лучше использовать базу данных, которая медленно сохраняет данные, а не быстро их теряет.

Чтобы сравнить базы данных, полезно тщательно изучить сценарий использования и определить текущие и ожидаемые переменные:

- размеры схемы данных и записей;
- количество клиентов;
- типы запросов и паттерны доступа;
- скорость выполнения запросов на чтение и запись;
- ожидаемые изменения в любой из этих переменных.

Знание этих переменных может помочь ответить на следующие вопросы:

- Поддерживает ли база данных необходимые запросы?
- Способна ли база данных обрабатывать тот объем данных, который мы планируем хранить?
- Сколько операций чтения и записи может обрабатывать один узел?
- Сколько узлов должна включать в себя система?
- Как расширить кластер с учетом ожидаемых темпов роста?
- В чем заключается процесс обслуживания?

Получив ответы на эти вопросы, вы сможете построить тестовый кластер и смоделировать свои рабочие нагрузки. Для большинства баз данных уже есть инструменты стресс-анализа, которые можно применить для воспроизведения конкретных сценариев использования. Если в экосистеме базы данных нет стандартного инструментария стресс-анализа для создания реалистичных рандомизированных рабочих нагрузок, это может стать настораживающим знаком. Если что-то мешает вам использовать инструменты, поставляемые вместе с самой базой, можно попро-

бовать один из существующих инструментов общего назначения или реализовать его с нуля.

Если тесты показывают положительные результаты, часто полезно ознакомиться с кодом базы данных. При изучении кода обычно лучше сначала выявить составные части базы данных, понять, как найти код различных компонентов, а затем пройтись по ним. Имея даже приблизительное представление о кодовой базе СУБД, вы сможете лучше разбираться в создаваемых ею записях журнала и ее параметрах конфигурации, а также выявлять проблемы в использующем ее приложении и даже в самом коде СУБД.

Было бы здорово, если бы мы могли использовать базы данных как черные ящики и никогда не заглядывать в них, но практика показывает, что рано или поздно возникает ошибка, сбой, регрессия производительности или какая-то другая проблема и лучше быть готовым к этому. Если вы знаете и понимаете внутренние компоненты базы данных, то можете снизить бизнес-риски и повысить шансы на быстрое восстановление.

Одним из популярных инструментов для тестирования, оценки производительности и сравнения является Yahoo! Cloud Serving Benchmark (YCSB) (https://databass.dev/links/104). YCSB предлагает инфраструктуру и общий набор рабочих нагрузок, которые могут быть применены к различным хранилищам данных. Как и все средства общего назначения, этот инструмент следует использовать с осторожностью, так как можно с легкостью прийти к неправильным выводам. Чтобы провести справедливое сравнение и принять обоснованное решение, необходимо потратить достаточно времени, чтобы понять реальные условия, в которых должна функционировать база данных, и соответствующим образом адаптировать сравнительные тесты.

СРАВНИТЕЛЬНЫЙ ТЕСТ TPC-C

Совет по оценке производительности обработки транзакций (Transaction Processing Performance Council, TPC) предлагает набор сравнительных тестов, которые поставщики баз данных используют для сравнения и рекламирования производительности своих продуктов. TPC-C — это тест обработки транзакций в реальном времени (Online Transaction Processing, OLTP), представляющий собой смесь транзакций только для чтения и обновления, которые имитируют распространенные рабочие нагрузки приложений.

TPC-C тестирует производительность и корректность выполняемых конкурентных транзакций. Основным показателем производительности является пропускная способность: количество транзакций, которые СУБД может обрабатывать в минуту. Выполняемые транзакции должны сохранять ACID-свойства и соответствовать (не противоречить) набору свойств, определяемых самим тестом.

Этот тест не относится к какому-либо конкретному бизнес-сегменту, но предоставляет абстрактный набор действий, важных для большинства приложений, для которых подходят базы данных OLTP. Он включает в себя несколько таблиц и объектов, таких как склады, товарные запасы, заказчики и заказы, и определяет макеты таблиц, сведения о транзакциях, которые могут быть выполнены с этими таблицами, минимальное количество строк в таблице и ограничения на уровень устойчивости данных.

Это не означает, что сравнительные тесты можно использовать *только* для сравнения баз данных. Сравнительные тесты могут быть полезны для определения и проверки положений соглашения об уровне обслуживания[1], определения системных требований, планирования производительности и многого другого. Чем больше вы узнаете о базе данных до ее использования, тем меньше времени придется тратить при эксплуатации.

Выбор базы данных — это долгосрочное решение, и потому желательно отслеживать выход новых версий, понимать, что именно изменилось и почему, и иметь некоторую стратегию обновления. Новые выпуски обычно содержат улучшения и исправления ошибок и проблем безопасности, но могут содержать и новые ошибки, вести к снижению производительности или неожиданному поведению, поэтому новые версии тоже важно тестировать перед их развертыванием. Проверка того, как разработчики базы данных проводили обновления ранее, может дать вам хорошее представление о том, чего ожидать в будущем. Если предыдущие обновления проходили гладко, это еще не гарантирует, что также будет и в дальнейшем, однако если они вызывали затруднения, это может быть признаком того, что будущие обновления тоже дадутся нелегко.

Понимание преимуществ и недостатков

В качестве пользователей мы можем видеть, как базы данных ведут себя в различных условиях, но при разработке баз данных мы должны принимать решения, оказывающие непосредственное влияние на это поведение.

Проектирование подсистемы хранения, безусловно, сложнее, чем просто реализация структуры данных из учебника: здесь имеется много деталей и пограничных случаев, с которыми обычно не удается справиться с первого раза. Нужно спроектировать физический макет данных и организовать указатели, принять решение о формате сериализации, понять, как данные будут удаляться при сборке мусора, как подсистема хранения вписывается в семантику СУБД в целом, выяснить, как заставить ее работать в конкурентном окружении, и, наконец, убедиться, что мы не будем терять данные ни при каких обстоятельствах.

Помимо того что вам нужно принять решения в отношении множества различных вещей, ситуация осложняется еще и тем, что в большинстве случаев эти решения подразумевают нахождение некоторого компромисса. Например, если мы будем сохранять записи в том же порядке, в котором они вносятся в базу данных, их можно будет сохранять быстрее, но если при этом их нужно будет извлекать в лексикографическом порядке, то их потребуется пересортировывать перед возвращением

[1] Соглашение об уровне обслуживания (service-level agreement, SLA) — это обязательство поставщика услуг относительно качества предоставляемых услуг. Помимо прочего, такое соглашение может включать информацию о задержке, пропускной способности, дрожании, а также количестве и частоте отказов.

результатов клиенту. Как вы увидите на протяжении этой книги, существует много различных подходов к разработке подсистемы хранения и каждая реализация имеет свои собственные плюсы и минусы.

При рассмотрении различных подсистем хранения мы будем обсуждать все их преимущества и недостатки. Если бы существовала подсистема хранения, идеальным образом подходящая для каждого возможного сценария использования, то все бы просто использовали его. Но поскольку такой подсистемы хранения не существует, мы должны подходить к выбору очень тщательно, принимая в расчет ожидаемые рабочие нагрузки и сценарии использования.

Существует множество различных подсистем хранения, использующих разные структуры данных и реализованных на разных языках, начиная с таких низкоуровневых языков, как C, и заканчивая такими высокоуровневыми языками, как Java. В то же время все подсистемы хранения сталкиваются с одинаковыми проблемами и ограничениями. Это можно сравнить с планированием городской застройки — планируя застройку под конкретное количество людей, вы можете увеличивать город в вертикальном или горизонтальном направлении. Хотя в обоих случаях город сможет принять одинаковое количество людей, эти подходы подразумевают совершенно разный образ жизни. При вертикальном росте застройки люди будут жить в квартирах и высокая плотность населения, вероятно, приведет к высокой интенсивности транспортного движения. С другой стороны, при менее плотном размещении люди, вероятно, будут жить в домах, но дальше от места работы.

Аналогичным образом проектные решения, принимаемые разработчиками подсистем хранения, делают их более подходящими для различных целей: одни из них обеспечивают минимальное время чтения и записи, другие — максимальную плотность (количество хранимых данных, приходящееся на каждый узел), а третьи — максимальную простоту эксплуатации.

В разделе «Итоги» в конце каждой главы вы найдете ссылки на полное описание применяемых в реализации алгоритмов и другую полезную информацию. Чтение этой книги должно хорошо подготовить вас для эффективного использования этих источников и дать прочное понимание того, какие имеются альтернативы для описываемых в них концепций.

ГЛАВА 1

Введение и обзор

СУБД служат различным целям: некоторые используются в основном для временных «горячих» данных, некоторые служат в качестве долговременного хранилища «холодных» данных, некоторые подходят для сложных аналитических запросов, некоторые позволяют получать доступ к значениям только по ключу, некоторые оптимизированы для хранения данных временных рядов, а некоторые эффективно хранят большие двоичные объекты. Сначала мы рассмотрим и обозначим различия, начав с краткой классификации и обзора, поскольку это поможет оценить масштаб дальнейшей дискуссии.

Терминология иногда может быть неоднозначной и трудной для понимания без принятия во внимание полного контекста. Примером могут служить различия между *колоночными* хранилищами и хранилищами *с широкими столбцами*, которые имеют очень мало общего между собой, или взаимосвязь *кластеризованных* или *некластеризованных индексов* и *индексированных таблиц.* Цель данной главы состоит в том, чтобы дать однозначное и точное определение таких терминов.

Мы начнем с обзора архитектуры СУБД (см. раздел «Архитектура СУБД» на с. 25) и обсудим отдельные ее компоненты и их функции. После этого мы осветим различия между различными СУБД с точки зрения используемой среды хранения (см. раздел «Резидентные и дисковые СУБД» на с. 27) и структуры (см. раздел «Колоночные и строчные СУБД» на с. 30).

Эти два способа разделения являются далеко не единственными вариантами классификации СУБД. Например, СУБД часто разделяют на три основные категории:

Базы данных для обработки транзакций в реальном времени (*online transaction processing, OLTP*)

Они обрабатывают большое количество поступающих со стороны пользователя запросов и транзакций. Запросы часто бывают предопределенными и с коротким жизненным циклом.

Базы данных для аналитической обработки данных в реальном времени (*online analytical processing, OLAP*)

Они обрабатывают сложные агрегаты данных. OLAP-системы часто используются для аналитики и построения хранилищ данных и способны обрабатывать сложные произвольные специальные запросы с длительным жизненным циклом.

Гибридная транзакционно-аналитическая обработка (*hybrid transactional and analytical processing, HTAP*)

Эти базы данных сочетают в себе свойства хранилищ OLTP и OLAP.

Существует и множество других терминов и способов классификации: хранилища типа «ключ–значение», реляционные базы данных, документоориентированные хранилища и графовые базы данных. Эти понятия здесь не определяются, так как предполагается, что читатель хорошо знает и понимает их. Поскольку концепции, которые мы здесь обсуждаем, широко применимы и используются в большинстве упомянутых типов хранилищ в том или ином качестве, полная систематика не является необходимой или важной для дальнейшего обсуждения.

Часть I этой книги посвящена структурам хранения и индексирования, поэтому нам необходимо понять высокоуровневые подходы к организации данных и взаимосвязь файлов данных и индексных файлов (см. раздел «Файлы данных и индексные файлы», с. 34).

Наконец, в разделе «Буферизация, неизменяемость и упорядочение» на с. 39 мы обсудим три широко используемых метода разработки эффективных структур хранения данных, а также поговорим о том, как применение этих методов влияет на проектирование и реализацию структур.

Архитектура СУБД

Не существует общего шаблона для проектирования СУБД. Каждая база данных строится немного по-разному, а границы компонентов довольно трудно обнаружить и определить. Даже если эти границы существуют на бумаге (например, в проектной документации), в коде кажущиеся независимыми компоненты могут оказаться связанными из-за оптимизации производительности, обработки граничных случаев или применения определенных архитектурных решений.

Источники, описывающие архитектуру СУБД (например, [HELLERSTEIN07] [WEIKUM01] [ELMASRI11] и [MOLINA08]), определяют компоненты и отношения между ними поразному. Архитектура, показанная на рис. 1.1, демонстрирует некоторые общие элементы этих представлений.

СУБД используют *модель «клиент — сервер»*, в которой экземпляры СУБД (*узлы*) играют роль серверов, а экземпляры приложений — роль клиентов.

Запросы клиентов поступают через *транспортную подсистему*. Поступающие запросы чаще всего оформлены на каком-либо языке запросов. Транспортная подсистема также отвечает за связь с другими узлами кластера баз данных.

После получения запроса транспортная подсистема передает его обработчику запросов, который анализирует, интерпретирует и проверяет его. Далее проводятся проверки управления доступом, так как для их полного выполнения необходима интерпретация запроса.

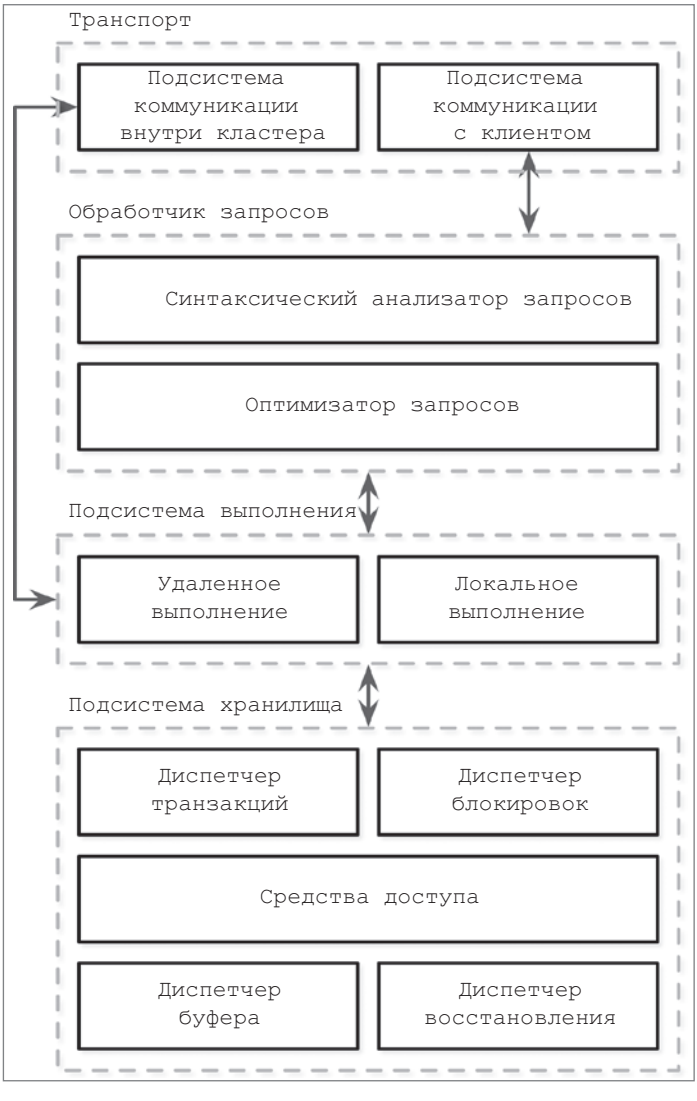

Рис. 1.1. Архитектура СУБД

Анализируемый запрос передается *оптимизатору запросов*, который сначала устраняет невозможные и избыточные части запроса, а затем пытается найти наиболее эффективный способ его выполнения на основе внутренней статистики (мощность индекса, приблизительный размер пересечений и т. д.) и размещения данных (какие узлы в кластере содержат данные и какие затраты требуются для их передачи). Оптимизатор обрабатывает реляционные операции, необходимые для разрешения запросов, обычно представленные в виде дерева зависимостей, и проводит оптимизации, такие как упорядочение индексов, оценка мощности и выбор средств доступа.

Запрос обычно представлен в виде *плана выполнения* (или *плана запроса*): последовательности операций, которую необходимо выполнить для того, чтобы результаты запроса считались полными. Поскольку один и тот же запрос может быть выполнен с помощью различных планов выполнения, которые могут отличаться по эффективности, оптимизатор выбирает наиболее эффективный план из всех доступных.

План выполнения обрабатывается *подсистемой выполнения*, которая собирает результаты выполнения локальных и удаленных операций. *Удаленное выполнение* обычно сводится к записи и чтению данных на других узлах кластера, а также выполнению репликации.

Локальные запросы (поступающие непосредственно от клиентов или от других узлов) выполняются *подсистемой хранения данных*, которая включает в себя несколько компонентов с четко заданными функциями:

Диспетчер транзакций

Производит планировку транзакций и гарантирует, что они не оставят базу данных в логически несогласованном состоянии.

Диспетчер блокировок

Блокирует объекты базы данных для выполняемых транзакций, чтобы конкурентные операции не нарушали физическую целостность данных.

Средства доступа (структуры для хранения данных)

Управляют доступом и организацией данных на диске. Средства доступа включают в себя неупорядоченные файлы и такие структуры хранения, как B-деревья (см. раздел «Вездесущие B-деревья», с. 51) или LSM-деревья (см. раздел «LSM-деревья», с. 148).

Диспетчер буферов

Кэширует страницы данных в памяти (см. раздел «Организация буферизации данных», с. 98).

Диспетчер восстановления

Ведет журнал операций и восстанавливает состояние системы в случае сбоя (см. раздел «Восстановление», с. 106).

Вместе диспетчеры транзакций и блокировок отвечают за управление параллелизмом (см. раздел «Управление параллелизмом», с. 111): они гарантируют логическую и физическую целостность данных, обеспечивая при этом максимально эффективное выполнение конкурентных операций.

Резидентные и дисковые СУБД

СУБД хранят данные в оперативной памяти или на диске. *Резидентные СУБД* (или *СУБД в оперативной памяти*) хранят данные *главным образом* в оперативной памяти, а диск используют для восстановления и ведения журнала. *Дисковые СУБД* хранят

бóльшую часть данных на диске и используют память для кэширования содержимого диска или в качестве временного хранилища. Системы обоих типов в той или иной степени используют диск, но резидентные базы данных хранят содержимое почти исключительно в ОЗУ.

Доступ к памяти был и остается на несколько порядков быстрее доступа к диску[1], поэтому есть смысл использовать память в качестве основного хранилища. Такой подход становится все более экономически целесообразным, поскольку цены на память снижаются. Однако цены на оперативную память по-прежнему остаются высокими по сравнению с постоянными устройствами хранения данных, такими как твердотельные накопители и жесткие диски.

Резидентные СУБД отличаются от дисковых не только основной средой хранения данных, но и тем, как они организованы и какие структуры данных и методы оптимизации они используют.

Использование памяти в качестве основного хранилища данных в таких базах данных обусловлено главным образом высокой производительностью, сравнительно низкими затратами на доступ и высокой гранулярностью доступа. С точки зрения программирования работа с оперативной памятью также представляет гораздо меньше сложностей, чем работа с диском. Операционные системы абстрагируют управление памятью и позволяют нам мыслить в терминах выделения и освобождения областей памяти произвольного размера. На диске же мы должны вручную управлять ссылками на местонахождение данных, форматами сериализации, освобожденной памятью и фрагментацией.

Основными ограничивающими факторами роста количества резидентных баз данных являются непостоянство (т. е. недостаточная долговечность) оперативной памяти и высокая стоимость. Поскольку содержимое ОЗУ не является постоянным, программные ошибки, отказы, аппаратные сбои и перебои в подаче электроэнергии могут привести к потере данных. Существуют способы обеспечения долговечности, такие как использование источников бесперебойного питания и оперативной памяти с резервным аккумуляторным питанием, но они требуют дополнительных аппаратных ресурсов и более высокого уровня квалификации обслуживающего персонала. На практике решающую роль часто играет то, что диски проще в обслуживании и гораздо меньше стоят.

Ситуация, вероятно, будет меняться по мере роста доступности и популярности технологии энергонезависимой памяти (Non-Volatile Memory, NVM) [ARULRAJ17]. Энергонезависимое хранилище уменьшает или полностью устраняет (в зависимости от конкретной технологии) асимметрию между временем чтения и записи, дополнительно улучшает производительность чтения и записи и обеспечивает доступ с байтовой адресацией.

---

[1] Сравнение задержек доступа к диску и памяти и многих других важных параметров, приведенных за несколько лет, и их графическое представление см. на https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html.

Долговечность в резидентных хранилищах

Резидентные СУБД сохраняют резервные копии на диске, чтобы обеспечить долговечность и не допустить потери часто меняющихся данных. Хотя некоторые базы данных хранят данные исключительно в памяти, без каких-либо гарантий долговечности, мы не будем обсуждать их в рамках этой книги.

Прежде чем операция будет считаться завершенной, ее результаты должны быть записаны в последовательно организованный журнал предзаписи. Более подробно мы рассмотрим журналы упреждающей записи в разделе «Восстановление» на с. 106. Чтобы избежать применения полного содержимого журнала во время запуска или после сбоя, резидентные хранилища поддерживают *резервную копию*. Резервная копия поддерживается в виде упорядоченной структуры на диске; при этом изменения часто вносятся в нее асинхронно (без привязки к запросам клиентов) и применяются пакетами для уменьшения числа операций ввода-вывода. Во время восстановления содержимое базы данных может быть получено из резервной копии и журналов.

Записи журнала обычно применяются к резервной копии пакетами. После обработки пакета записей журнала резервная копия содержит *моментальный снимок* базы данных, который соответствует некоторому моменту, и все предшествующее содержимое журнала может быть удалено. Этот процесс называется *созданием контрольных точек*. При таком подходе сокращается время восстановления за счет поддержания дисковой базы данных в предельно актуальном состоянии с помощью записей журнала, без необходимости в блокировке доступа клиентов на время обновления резервной копии.

Не стоит думать, что резидентная база данных представляет собой что-то вроде дисковой базы данных с огромным страничным кэшем (см. раздел «Организация буферизации данных» на с. 98). При кэшировании страниц в памяти формат сериализации и способ представления данных несут с собой дополнительные издержки и не позволяют достичь той же степени оптимизации, которую могут обеспечить резидентные хранилища.

Дисковые базы данных используют специализированные структуры хранения данных, оптимизированные для доступа к диску. Поскольку в памяти сравнительно быстро осуществляются переходы по указателям, произвольный доступ к памяти осуществляется намного быстрее, чем произвольный доступ к диску. Дисковые структуры хранения данных часто имеют вид широкого и низкого дерева (см. подраздел «Деревья для дисковых хранилищ», с. 46), в то время как резидентные хранилища могут использовать больше разновидностей структур данных и способны на такие виды оптимизации, которые невозможно или очень сложно реализовать на диске [MOLINA92]. Точно так же на диске особого внимания требует обработка данных переменного размера, в то время как в памяти это обычно решается путем обращения к значению с помощью указателя.

В случае некоторых сценариев использования уместно предположить, что весь набор данных поместится в памяти. Некоторые наборы данных ограничены природой

вещей реального мира, которые они представляют: записи учащихся для школ, записи клиентов для корпораций или товарные запасы в интернет-магазине. Каждая запись занимает не более нескольких КБ, а их количество ограничено.

Колоночные и строчные СУБД

Большинство СУБД хранит некоторый *набор записей*, состоящий из *столбцов* и *строк таблиц*. *Поле* находится на пересечении столбца и строки и содержит одно значение некоторого типа. Поля, относящиеся к одному столбцу, обычно имеют один и тот же тип данных. Например, если мы определим таблицу, содержащую записи пользователей, все имена будут иметь один и тот же тип и находиться в одном и том же столбце. Набор значений, логически относимых к одной и той же записи (обычно идентифицируемой ключом), образует строку.

Один из способов классификации баз данных сводится к их разделению в зависимости от того, как данные сохраняются на диске: по строкам или по столбцам. Данные таблиц могут разбиваться либо по горизонтали (когда вместе сохраняются значения, относящиеся к одной строке), либо по вертикали (когда вместе сохраняются значения, относящиеся к одному столбцу). Разница между этими способами показана на рис. 1.2: (*а*) значения разбиваются по столбцам и (*б*) значения разбиваются по строкам.

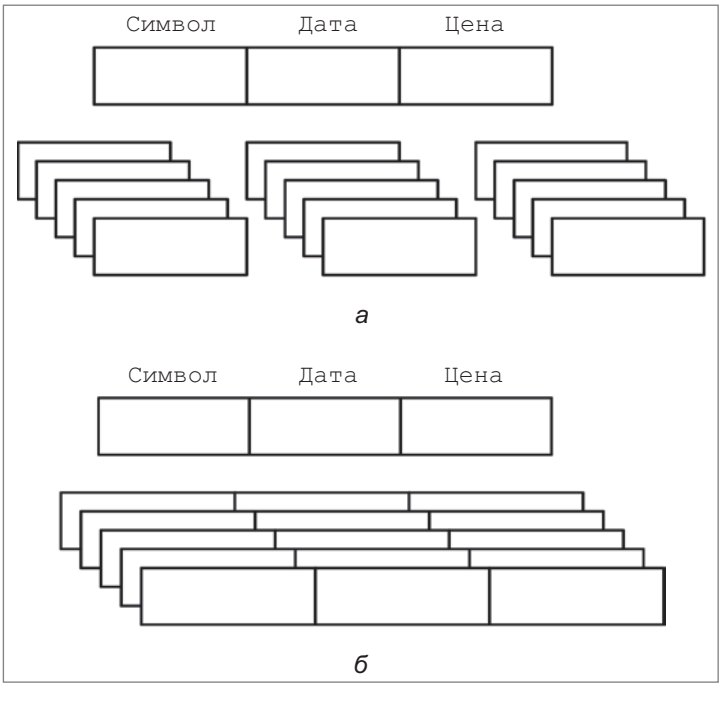

Рис. 1.2. Компоновка данных в колоночных и строчных хранилищах

Примеров строчных СУБД очень много: MySQL (https://dev.mysql.com/), PostgreSQL (https://www.postgresql.org/) и большинство традиционных реляционных баз данных. Первыми примерами колоночных хранилищ стали свободно распространяемые СУБД MonetDB (https://databass.dev/links/109) и C-Store (https://databass.dev/links/110) (на основе C-Store впоследствии была создана СУБД Vertica (https://databass.dev/links/111)).

Построчная компоновка данных

Строчные СУБД хранят данные в записях или строках. Этот способ компоновки очень напоминает табличное представление данных, при котором каждая строка имеет один и тот же набор полей. Например, строчная база данных эффективна для хранения записей пользователей, содержащих имена, даты рождения и номера телефонов.

Идентификатор	Имя	Дата рождения	Номер телефона
10	Джон	01 августа 1981	+1 111 222 333
20	Сэм	14 сентября 1988	+1 555 888 999
30	Кит	07 января 1984	+1 333 444 555

Этот подход хорошо работает в тех случаях, когда запись состоит из нескольких полей (имя, дата рождения и номер телефона) и однозначно идентифицируется ключом (в этом примере — монотонно возрастающее число). Все поля, представляющие запись одного пользователя, обычно считываются вместе. При создании записей (например, когда пользователь заполняет регистрационную форму) все поля также записываются вместе. В то же время каждое поле можно изменять по отдельности.

Поскольку строчные хранилища наиболее полезны, когда необходимо получать доступ к данным построчно, хранение целых строк в одном месте повышает степень пространственной локальности[1] [DENNING68].

Поскольку доступ к данным на персистентном носителе, таком как диск, обычно осуществляется поблочно (другими словами, минимальная единица доступа к диску — это блок), один блок будет содержать данные для всех столбцов. Такой способ отлично подходит для случаев, когда мы хотим получить доступ ко всей записи пользователя, но делает более затратными обращения к отдельным полям нескольких записей (например, запросы для получения только телефонных номеров), так как данные из других полей также будут загружаться в кэш.

Поколоночная компоновка данных

Колоночные СУБД разделяют данные *по вертикали* (т. е. по столбцам), вместо того чтобы хранить их построчно. В этом случае на диске вместе сохраняются значения отдельных столбцов (а не отдельных строк, как в предыдущем примере). Напри-

[1] Пространственная локальность — это один из принципов локальности, означающий, что если к какой-либо области памяти осуществляется доступ, то в ближайшем будущем будет осуществляться доступ к соседней области.

мер, при сохранении исторических данных о ценах на фондовом рынке мы можем сохранять вместе различные котировки цен. Сохранение значений разных столбцов в отдельных файлах или сегментах файлов позволяет эффективно выполнять запросы по столбцам, поскольку столбцы в таком случае могут быть считаны целиком, вместо того, чтобы считывать отдельные строки и отбрасывать данные тех столбцов, которые не были запрошены.

Колоночные хранилища хорошо подходят для аналитической обработки, требующей вычисления агрегатных величин, — для определения тенденций, вычислении средних значений и т. д. Обработка сложных агрегатов может быть использована в тех случаях, когда логические записи имеют несколько полей, но некоторые из них (в данном случае котировки цен) имеют различную значимость и часто используются вместе.

С логической точки зрения данные, представляющие котировки цен на фондовом рынке, также можно представить в виде таблицы:

Идентификатор	Символ	Дата	Цена
1	DOW	08 августа 2018	24314,65
2	DOW	09 августа 2018	24136,16
3	S&P	08 августа 2018	2414,45
4	S&P	09 августа 2018	2232,32

Однако физическая структура колоночной базы данных выглядит совершенно иначе. Здесь вместе сохраняются значения, относящиеся к одному и тому же столбцу:

```
Символ:   1:DOW; 2:DOW; 3:S&P; 4:S&P
Дата:     1:08 авг 2018; 2:09 авг 2018; 3:08 авг 2018; 4:09 авг 2018
Цена:     1:24 314,65; 2:24 136,16; 3:2 414,45; 4:2 232,32
```

Для восстановления кортежей данных, которые могут быть полезны для объединения, фильтрации и многорядных агрегатов, нам необходимо сохранить некоторые метаданные на уровне столбцов, чтобы определить, с какими элементами данных из других столбцов они связаны. Если вы сделаете это явно, то каждое значение будет содержать ключ, что приведет к дублированию и увеличению объема хранимых данных. Некоторые колоночные хранилища вместо этого используют неявные идентификаторы (*виртуальные идентификаторы*) и позиции значений (другими словами, их смещения) для их отображения на связанные значения [ABADI13].

В течение последних нескольких лет, вероятно, из-за растущего спроса на выполнение сложных аналитических запросов по увеличивающимся наборам данных появилось много новых колоночных форматов файлов, таких как Apache Parquet (https://databass.dev/links/112), Apache ORC (https://databass.dev/links/113), RCFile (https://databass.dev/links/114), а также много колоночных хранилищ, таких как Apache Kudu (https://databass.dev/links/115), ClickHouse (https://databass.dev/links/116) и многие другие [ROY12].

Различия и оптимизация

Различия между строчными и колоночными хранилищами заключаются не только в способе хранения данных. Выбор способа компоновки данных — это всего лишь один из шагов в серии возможных оптимизаций, возможных в колоночных хранилищах .

Считывание за один раз нескольких значений отдельного столбца значительно повышает эффективность использования кэша и вычислительную эффективность. Современные процессоры позволяют использовать векторные инструкции, что дает вам возможность обработать множество элементов данных с помощью всего одной инструкции[1] процессора [DREPPER07].

Сохранение в одном месте значений с одним типом данных (например, чисел вместе с другими числами, строк вместе с другими строками) открывает возможности для более высокой степени сжатия данных. Мы можем использовать различные алгоритмы сжатия в зависимости от типа данных и выбрать наиболее эффективный метод сжатия для каждого случая.

Чтобы решить, следует ли использовать колоночное или строчное хранилище, вам необходимо изучить *паттерны доступа*. Если считываемые данные используются в виде записей (т. е. запрашиваются все или почти все столбцы), а рабочая нагрузка включает в себя главным образом точечные запросы и операции сканирования диапазонов, то построчный подход, вероятно, даст лучшие результаты. Если требуется просматривать множество строк или вычислять агрегаты на основе подмножества столбцов, то стоит подумать об использовании поколоночного подхода.

Хранилища с широкими столбцами

Колоночные базы данных не следует путать с *хранилищами с широкими столбцами*, такими как BigTable (https://databass.dev/links/117) или HBase (https://databass.dev/links/118), где данные представлены в виде многомерной карты, столбцы сгруппированы в *семейства столбцов* (обычно хранящие данные одного типа), а внутри каждого семейства столбцов данные сохраняются построчно. Такая структура лучше всего подходит для хранения данных, извлекаемых с помощью ключа или последовательности ключей.

Классический пример — Webtable из статьи, посвященной BigTable [CHANG06]. Webtable сохраняет моментальные снимки содержимого веб-страниц, их атрибутов и связей между ними в момент, соответствующий определенной временной метке. Страницы идентифицируются по URL-адресу, записанному в обратном порядке, а все атрибуты (такие, как *содержимое страницы* и *якоря*, представляющие ссылки между страницами) идентифицируются по временным меткам, для которых были

[1] Векторные инструкции, или инструкции с одним потоком команд и несколькими потоками данных (Single Instruction Multiple Data, SIMD), составляют класс инструкций ЦП, выполняющих одну операцию над несколькими элементами данных.

сделаны эти снимки. В упрощенном виде это можно представить в виде вложенной карты, как показано на рис. 1.3.

```
"com.cnn.www": {
    contents: {
        t6: html: "<html>..."
        t5: html: "<html>..."
        t3: html: "<html>..."
    }
    anchor: {
        t9: cnnsi.com: "CNN"
        t8: my.look.ca: "CNN.com"
    }
}
"com.example.www": {
    contents: {
        t5: html: "<html>..."
    }
    anchor: {}
}
```

Рис. 1.3. Концептуальное представление структуры Webtable

Данные сохраняются в многомерной отсортированной карте с иерархическими индексами: мы можем найти данные, относящиеся к конкретной веб-странице, по ее обращенному URL-адресу и ее содержимое или якоря — по временной метке. Каждая строка индексируется своим *ключом строки*. Связанные столбцы группируются в *семейства столбцов* (в данном случае — в семейства `contents` (содержимое) и `anchor` (якорь)), которые сохраняются на диске по отдельности. Каждый столбец внутри семейства идентифицируется с помощью *ключа столбца*, включающего в себя имя семейства столбцов и квалификатор (в данном случае — `html`, `cnnsi.com`, `my.look.ca`). Семейства столбцов хранят несколько версий данных согласно меткам времени. Такая структура позволяет нам быстро находить записи более высокого уровня (веб-страницы в данном случае) и их параметры (версии содержимого и ссылки на другие страницы).

Хотя такое представление полезно для понимания сути хранилищ с широкими столбцами, их физическое расположение выглядит несколько по-другому. Схематическое представление компоновки данных в семействах столбцов показано на рис. 1.4: семейства столбцов сохраняются по отдельности, но в каждом семействе столбцов данные, относящиеся к отдельному ключу, сохраняются вместе.

Файлы данных и индексные файлы

Основной целью СУБД являются хранение данных и обеспечение быстрого доступа к ним. Но как организованы эти данные? Почему нужна именно СУБД, а не

Семейство столбцов: содержимое

Ключ строки	Временная метка	Квалификатор	Значение
com.cnn.www	t3	html	"<html>..."
com.cnn.www	t5	html	"<html>..."
com.cnn.www	t6	html	"<html>..."
com.example.www	t5	html	"<html>..."

Семейство столбцов: якорь

Ключ строки	Временная метка	Квалификатор	Значение
com.cnn.www	t8	cnnsi.com	"CNN"
com.cnn.www	t5	my.look.ca	"CNN.com"

Рис. 1.4. Физическая структура Webtable

просто некоторый набор файлов? Как организация файлов повышает эффективность работы?

СУБД используют файлы для хранения данных, но вместо того, чтобы для поиска записей полагаться на иерархию каталогов и файлов файловой системы, они формируют файлы с использованием специальных форматов, зависящих от конкретной реализации системы. Основными причинами использования специальной структуры файлов вместо неструктурированных файлов являются:

Эффективность хранения

Файлы организованы таким образом, чтобы минимизировать затраты на хранение каждой записи данных.

Эффективность доступа

Записи можно получить за наименьшее возможное число шагов.

Эффективность обновления

Обновления записей выполняются таким образом, чтобы свести к минимуму количество внесенных на диск изменений.

СУБД хранят *записи данных*, состоящие из нескольких полей, в таблицах, а каждая таблица обычно представлена отдельным файлом. Каждую запись в таблице можно найти с помощью *ключа поиска*. Для поиска записей СУБД используют *индексы*: вспомогательные структуры данных, которые позволяют эффективно находить записи данных без сканирования всей таблицы при каждой операции доступа. Индексы состоят из подмножества полей, идентифицирующих записи.

В системах баз данных обычно разделяются *файлы данных* и *индексные файлы*: файлы данных хранят записи данных, а индексные файлы хранят метаданные записей, кото-

рые используются для поиска записей в файлах данных. Индексные файлы обычно меньше файлов данных. Файлы разбиваются на *страницы*, которые обычно имеют размер одного или нескольких дисковых блоков. Страницы могут быть организованы как последовательности записей или как *слоттированные страницы* (slotted page) (см. раздел «Слоттированные страницы», с. 70).

Новые записи (вставки) и обновления существующих записей представлены парами «ключ–значение». Большинство современных систем хранения данных *не удаляют* данные со страниц явным образом. Вместо этого они используют *маркеры удаления* (также называемые *отметками полного удаления*), которые содержат метаданные удаления, такие как ключ и временная метка. Пространство, занимаемое записями, *затененными* обновлениями или маркерами удаления, освобождается во время сборки мусора, в ходе которой считываются страницы, затем в новое место записываются актуальные (т. е. незатененные) записи и удаляются затененные.

Файлы данных

Файлы данных (иногда называемые *первичными* файлами) могут быть реализованы в виде *индексированной таблицы* (index-organized table, IOT), *таблицы-кучи* (*файла-кучи*) или *хэшированной таблицы* (*хэшированного файла*).

Записи в файлах-кучах не организованы согласно какому-либо определенному порядку и в большинстве случаев размещаются в порядке записи. Таким образом, при добавлении новых страниц не требуется никаких дополнительных действий или реорганизации файлов. Файлы-кучи требуют использования дополнительных индексных структур, указывающих на места хранения записей данных, чтобы можно было производить их поиск.

В хэшированных файлах записи хранятся в сегментах, а хэш-значение ключа определяет, к какому сегменту относится запись. Записи в сегменте могут храниться в порядке добавления или сортироваться по ключам для повышения скорости поиска.

В индексированных таблицах (IOT) данные хранятся в самом индексе. Поскольку записи отсортированы по ключу, сканирование диапазона в IOT можно реализовать путем последовательного сканирования содержимого.

Хранение записей данных в индексе позволяет снизить число операций дискового поиска по крайней мере на единицу, так как после обхода индекса и нахождения искомого ключа нам не нужно обращаться к отдельному файлу, чтобы найти соответствующую запись данных.

Когда записи хранятся в отдельном файле, индексные файлы содержат *элементы данных*, однозначно идентифицирующие записи данных и содержащие достаточно информации, чтобы найти их в файле данных. Например, мы можем хранить файловые *смещения* (иногда называемые *локаторами строк*), позиции записей данных в файле данных или идентификаторы сегментов в случае хэш-файлов. В индексированных таблицах элементы данных содержат непосредственно записи данных.

Индексные файлы

Индекс — это структура, с помощью которой записи данных организуются на диске таким образом, чтобы повысить эффективность операций извлечения. Индексные файлы организованы как специализированные структуры, которые сопоставляют ключи с теми местами в файлах данных, где хранятся записи, идентифицированные этими ключами (в случае файлов-куч) или первичными ключами (в случае индексированных таблиц).

Индекс *первичного файла* (*файла данных*) называется *первичным индексом*. В большинстве случаев мы также можем принять, что первичный индекс строится на основе первичного ключа или набора ключей, идентифицированных как первичные. Все остальные индексы называются *вторичными*.

Вторичные индексы могут указывать непосредственно на запись данных или просто хранить ее первичный ключ. Указатель на запись данных может содержать смещение в файле-куче или индексированной таблице. Несколько вторичных индексов могут указывать на одну и ту же запись, что позволяет идентифицировать одну запись данных разными полями и находить ее с помощью разных индексов. Первичные индексные файлы содержат уникальную запись для каждого ключа поиска, а вторичные индексы могут содержать несколько записей для каждого ключа поиска [MOLINA08].

Если порядок записей данных соответствует порядку ключей поиска, такой индекс называется *кластеризованным*. Записи данных в случае кластеризации обычно хранятся в том же файле или в *кластеризованном файле* в соответствии с порядком ключей. Если данные хранятся в отдельном файле и их порядок не соответствует порядку ключей, индекс называется *некластеризованным* (или *некластерным*).

На рис. 1.5 показана разница между этими двумя подходами:

а) Два индекса ссылаются на записи данных непосредственно из вторичных индексных файлов.

б) Вторичный индекс обращается к уровню косвенности первичного индекса, чтобы найти записи данных.

Многие СУБД используют явный *первичный ключ* — набор столбцов, которые однозначно идентифицируют запись базы данных. В тех случаях, когда первичный ключ не задан, может создать *неявный* первичный ключ (например, подсистема хранения InnoDB базы данных MySQL добавляет новый столбец с автоинкрементом индекса и заполняет его значения автоматически).

Эта терминология используется в СУБД различных типов: реляционных СУБД (таких, как MySQL и PostgreSQL), нереляционных хранилищах на основе Dynamo (таких, как Apache Cassandra (https://databass.dev/links/119) и Riak (https://databass.dev/links/120)) и документоориентированных хранилищах (таких, как MongoDB). Понятия могут иметь специфичные для конкретной системы названия, но чаще всего есть четкое сопоставление с этой терминологией.

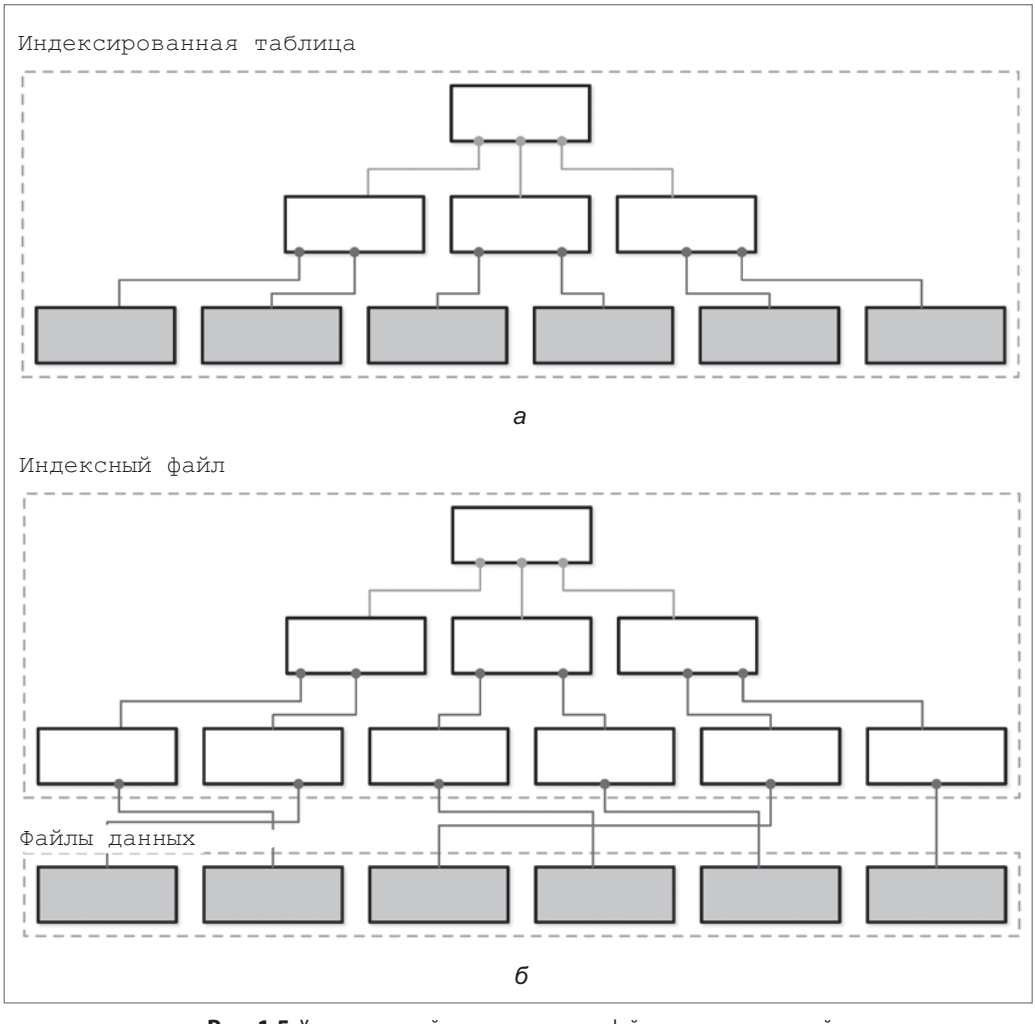

Рис. 1.5. Хранение записей данных в индексном файле и хранение смещений в пределах файла данных (сегменты индекса показаны белым цветом; сегменты, содержащие записи данных, показаны серым цветом)

Индексированные таблицы хранят информацию в порядке индекса и являются кластеризованными по определению. Первичные индексы являются кластеризованными *в большинстве случаев*. Вторичные индексы являются некластеризованными по определению, так как служат для облегчения доступа по ключам, отличным от первичных. Кластеризованные индексы могут быть как индексированными файлами, так и иметь отдельные файлы индексов и данных.

Первичный индекс как косвенный уровень

Специалисты по базам данных расходятся во мнениях относительно того, следует ли ссылаться на записи данных напрямую (посредством файлового смещения) или посредством индекса первичного ключа[1].

Оба подхода имеют свои плюсы и минусы, и лучше обсуждать их в рамках полной реализации. При непосредственном обращении к данным мы можем сократить число операций дискового поиска, но здесь кроются дополнительные затраты на обновление указателей всякий раз, когда запись обновляется или перемещается в процессе обслуживания. Использование косвенного обращения посредством первичного индекса позволяет снизить затраты на обновление указателей, но при этом возрастает стоимость чтения.

Обновление всего нескольких индексов хорошо подходит в том случае, когда рабочая нагрузка в основном состоит из операций чтения, но этот подход плохо работает для рабочих нагрузок с интенсивной записью данных с использованием нескольких индексов. Для снижения стоимости обновления указателей вместо смещения полезных данных некоторые реализации используют первичные ключи для косвенной адресации. Например, MySQL InnoDB использует первичный индекс и при выполнении запроса производит две операции поиска: по вторичному и по первичному индексам [TARIQ11]. При этом мы несем дополнительные затраты на выполнение поиска по первичному индексу, вместо того чтобы следовать смещению непосредственно из вторичного индекса.

На рис. 1.6 показано различие этих двух подходов:

а) Два индекса ссылаются на записи данных непосредственно из вторичных индексных файлов.

б) Вторичный индекс обращается к уровню косвенности первичного индекса, чтобы найти записи данных.

Также можно использовать гибридный подход и хранить смещения вместе с первичными ключами внутри файлов данных. Во время обработки запроса, вы сперва проверяете, является ли смещение (указатель) на данные все еще действительным, и только в случае если указатель более не действителен, вы будете вынуждены осуществить поиск по первичному ключу, и обновить устарелый указатель.

Буферизация, неизменяемость и упорядочение

Каждая подсистема хранения построена на основе некоторой структуры данных. Однако эти структуры не описывают суть кэширования, восстановления, управления

[1] Исходная публикация, которая вызвала эту дискуссию, была спорной и односторонней, но вы можете обратиться к презентации, сравнивающей индексы MySQL и PostgreSQL и форматы хранения, которая также ссылается на исходный источник.

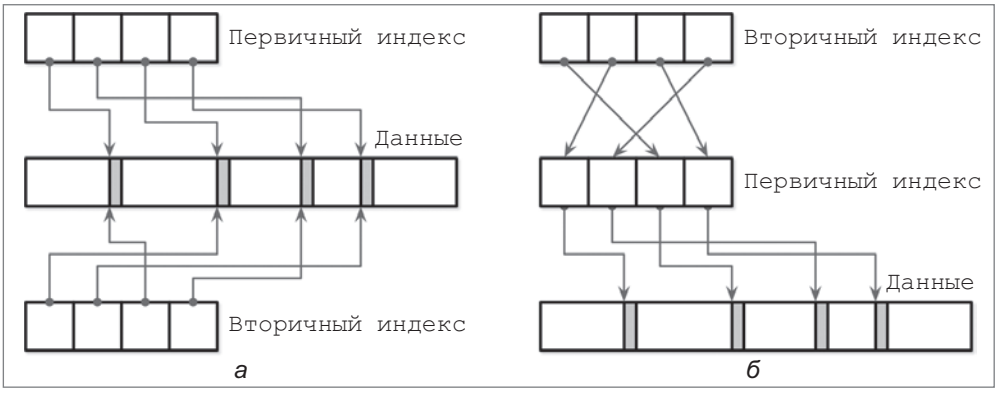

Рис. 1.6. Прямое обращение к кортежам данных (а) и использование первичного индекса в качестве косвенного уровня (б)

транзакциями и других элементов, надстраиваемых подсистемами хранения поверх этих структур.

В следующих главах мы начнем обсуждение с B-деревьев (см. раздел «Вездесущие B-деревья», с. 51) и попытаемся понять, почему существует так много вариантов B-деревьев и почему для баз данных постоянно появляются новые структуры хранения данных.

Структуры хранения имеют три общие переменные: они используют *буферизацию* (или избегают ее использование), *неизменяемые* (или изменяемые) файлы и хранят значения в *упорядоченном* (или неупорядоченном) виде. Большинство из тех различий и оптимизаций в структурах хранения, которые мы рассмотрим в этой книге, связаны с одним из этих трех понятий.

Буферизация

Определяет, будет ли структура хранения накапливать определенный объем данных в памяти перед их размещением на диске. Конечно, каждая дисковая структура должна в какой-то степени использовать буферизацию, так как минимальная единица передачи данных на диск и с диска — это *блок* и желательно записывать полные блоки. Здесь речь идет о необязательной буферизации, которая реализуется разработчиками подсистем хранения по их усмотрению. Так, одна из первых оптимизаций, обсуждаемых в этой книге, сводится к тому, чтобы добавить резидентные буферы в узлы B-дерева и тем самым снизить затраты на ввод-вывод (см. раздел «Ленивые B-деревья», с. 132). Однако это не единственный способ применения буферизации. Например, двухкомпонентные LSM-деревья (см. подраздел «Двухкомпонентное LSM-дерево», с. 151), несмотря на их сходство с B-деревьями, используют буферизацию совершенно по-другому и сочетают буферизацию с неизменяемостью.

Изменяемость (*или неизменяемость*)

Определяет, будет ли структура хранения после считывания частей файла и их обновления записывать обновленные результаты в то же место в файле. Неизменяемые структуры *доступны только для добавления*: после записи содержимое файла не изменяется. Вместо этого изменения добавляются в конец файла. Есть и другие способы реализации неизменяемости. Один из них — *копирование при записи* (см. раздел «Копирование при записи», с. 129), где измененная страница, содержащая обновленную версию записи, записывается в новое место в файле, а не в ее исходное местоположение. Одно из главных различий между LSM- и B-деревьями состоит в том, что первые являются неизменяемыми структурами, а вторые — структурами с обновлением на месте. В то же время некоторые структуры основаны на идее B-деревьев, но являются при этом неизменяемыми, как, например, Bw-деревья (см. раздел «Bw-деревья», с. 138).

Упорядочение

Определяется тем, совпадает ли порядок (сортировка) записей данных в страницах на диске с порядком ключей в данных. Другими словами, хранятся ли соседние ключи в смежных сегментах на диске. Это свойство часто определяет, можем ли мы эффективно сканировать *диапазон* записей, а не просто находить отдельные записи данных. Хранение данных в неупорядоченном виде (чаще всего в порядке включения в базу) открывает некоторые возможности для оптимизации времени записи. Например, Bitcask (см. подраздел «Bitcask», с. 171) и WiscKey (см. подраздел «WiscKey», с. 173) хранят записи данных непосредственно в файлах, доступных только для добавления.

Конечно, краткого обсуждения этих трех концепций недостаточно, чтобы показать их силу, и мы продолжим эту дискуссию на протяжении всей книги.

Итоги

В этой главе мы обсудили архитектуру СУБД и рассмотрели ее основные компоненты.

Мы обсудили резидентные и дисковые хранилища, чтобы выяснить, насколько важными являются дисковые структуры и чем они отличаются от структур, размещаемых в оперативной памяти. Мы пришли к выводу, что дисковые структуры важны для обоих типов хранилищ, но используются для разных целей.

Чтобы понять, как схемы доступа влияют на проектирование СУБД, мы обсудили колоночные и строчные СУБД, а также основные их различия. Чтобы начать дискуссию о *способах хранения данных*, мы рассмотрели файлы данных и индексные файлы.

Наконец, мы ввели три основных понятия: буферизация, неизменяемость и упорядочение. Мы будем использовать их на протяжении всей этой книги, чтобы акцентировать внимание на связанных с ними свойствах подсистем хранения.

Дополнительная литература

Если вы хотите узнать больше о концепциях, упомянутых в этой главе, вы можете обратиться к следующим источникам:

Архитектура баз данных

Hellerstein, Joseph M., Michael Stonebraker, and James Hamilton. 2007. «Architecture of a Database System». Foundations and Trends in Databases 1, no. 2 (February): 141–259. https://doi.org/10.1561/1900000002.

Колоночные СУБД

Abadi, Daniel, Peter Boncz, Stavros Harizopoulos, Stratos Idreaos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. Hanover, MA: Now Publishers Inc.

Резидентные СУБД

Faerber, Frans, Alfons Kemper, and Per-Åke Alfons. 2017. Main Memory Database Systems. Hanover, MA: Now Publishers Inc.

ГЛАВА 2

Введение в B-деревья

В предыдущей главе мы разделили структуры хранения данных на две группы: изменяемые и неизменяемые, и определили неизменяемость как одно из основных свойств, влияющих на их проектирование и реализацию. Большинство изменяемых структур хранения используют механизм обновления на месте. Во время операций вставки, удаления или обновления записи данных обновляются непосредственно на месте их расположения в целевом файле.

Подсистемы хранения часто допускают наличие в базе данных нескольких версий одной и той же записи данных; например, при использовании многоверсионного управления конкурентным доступом (см. подраздел «Многоверсионное управление конкурентным доступом» на с. 117) или при организации слоттированных страниц (см. раздел «Слоттированные страницы» на с. 70). Для простоты пока давайте предположим, что каждый ключ связан только с одной записью данных, которая имеет уникальное местоположение.

Одной из самых популярных структур хранения данных является B-дерево. Многие СУБД с открытым исходным кодом основаны на B-деревьях, и за прошедшие годы они доказали, что охватывают большинство сценариев использования.

B-деревья не являются недавним изобретением: они были введены Рудольфом Байером и Эдвардом М. Маккрейтом еще в 1971 году и с годами приобрели популярность. К 1979 году существовало уже довольно много вариантов B-деревьев. Дуглас Комер сгруппировал и систематизировал некоторые из них [COMER79].

Прежде чем мы погрузимся в изучение B-деревьев, давайте сначала поговорим о том, почему мы должны рассматривать альтернативы традиционным деревьям поиска, таким как, например, двоичные деревья поиска, 2-3-деревья и АВЛ-деревья [KNUTH98]. Для этого давайте вспомним, что такое двоичные деревья поиска.

Двоичные деревья поиска

Двоичное дерево поиска — это упорядоченная структура данных в оперативной памяти, используемая для эффективного поиска значений по ключу. Двоичные деревья состоят из нескольких узлов. Каждый узел дерева представлен ключом, значением, связанным с этим ключом, и двумя указателями на потомков (отсюда и название — двоичное). Двоичные деревья начинаются с одного узла, называемого *корневым*. При этом у дерева может быть только один корень. На рис. 2.1 показан пример двоичного дерева поиска.

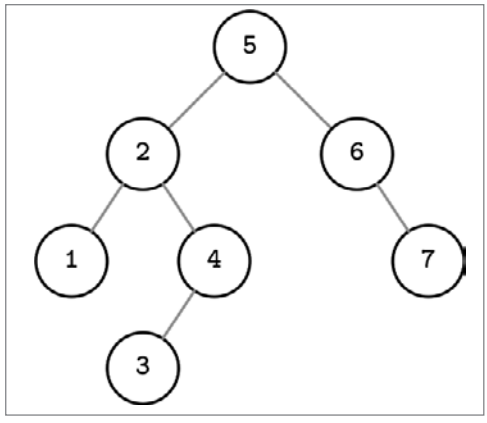

Рис. 2.1. Двоичное дерево поиска

Каждый узел разделяет пространство поиска на левое и правое поддеревья, как показано на рис. 2.2; при этом ключ узла *больше* любого ключа, хранящегося в его левом поддереве, и *меньше* любого ключа, хранящегося в его правом поддереве [SEDGEWICK11].

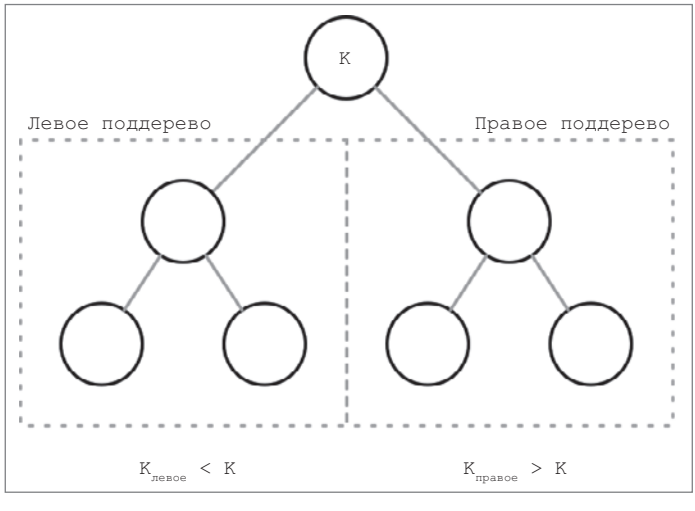

Рис. 2.2. Инварианты узлов двоичного дерева

Следуя по левым указателям от корня дерева вниз до листового уровня (где узлы уже не имеют потомков), можно найти узел, содержащий наименьший ключ в дереве и связанное с ним значение. Аналогичным образом, следуя по правым указателям, можно найти узел, содержащий самый большой ключ в дереве и связанное с ним значение. Значения можно хранить в любых узлах дерева. Поиск начинается

с корневого узла и может завершиться до достижения нижнего уровня дерева, если искомый ключ будет найден на более высоком уровне.

Балансировка деревьев

Операции вставки не следуют какой-либо определенной схеме, поэтому вставка элементов может привести к тому, что дерево станет несбалансированным (т. е. одна из его ветвей будет длиннее другой). Как показано на рис. 2.3 (*б*), в наихудшем случае мы можем в итоге получить «вырожденное» (или «патологическое») дерево, которое больше похоже на связанный список и вместо желаемой логарифмической сложности дает нам линейную, как показано на рис. 2.3 (*а*).

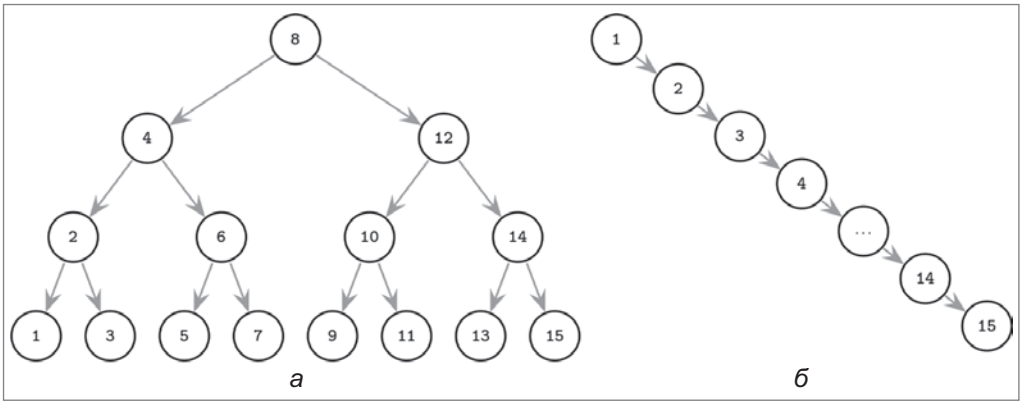

Рис. 2.3. Примеры сбалансированного (а) и несбалансированного, или патологического, (б) дерева

Этот пример может показаться несколько преувеличенным, но он иллюстрирует, почему дерево должно быть сбалансировано: даже если и маловероятно, что все элементы окажутся на одной стороне дерева, то по крайней мере некоторые из них обязательно окажутся там, что значительно замедлит поиск.

Сбалансированное дерево определяется как дерево, имеющее высоту $\log_2 N$, где N — общее число элементов в дереве, а разница в высоте между двумя поддеревьями не превышает единицы[1] [KNUTH98]. Без балансировки мы теряем преимущества производительности структуры двоичного дерева поиска и позволяем операциям вставки и удаления определять форму дерева.

В сбалансированном дереве следование указателю на левый или правый узел сокращает пространство поиска в среднем вдвое, поэтому сложность поиска логарифмическая: $O(\log_2 N)$. Если дерево не сбалансировано, в худшем случае сложность

[1] Этот параметр соблюдается АВЛ-деревьями и несколькими другими структурами данных. В более общем случае двоичные деревья поиска сохраняют разницу высот между поддеревьями в пределах небольшого постоянного значения.

возрастает до O(N), так как мы можем оказаться в ситуации, когда все элементы расположатся на одной стороне дерева.

Вместо того чтобы добавлять новые элементы в одну из ветвей дерева и делать ее длиннее, в то время как другая остается пустой (как показано на рис. 2.3 (*б*)), дерево *балансируется* после каждой операции. Балансировка выполняется путем реорганизации узлов таким образом, чтобы минимизировать высоту дерева и сохранить количество узлов на каждой стороне в конкретных пределах.

Один из способов сохранения сбалансированности дерева сводится к выполнению операции поворота после добавления или удаления узлов. Если операция вставки оставляет ветвь несбалансированной (два последовательных узла в ветви имеют только одного потомка), мы можем повернуть узлы вокруг среднего узла. В примере, показанном на рис. 2.4, во время поворота средний узел (3) выступает в качестве оси вращения и смещается на один уровень выше; при этом его родитель становится его правым потомком.

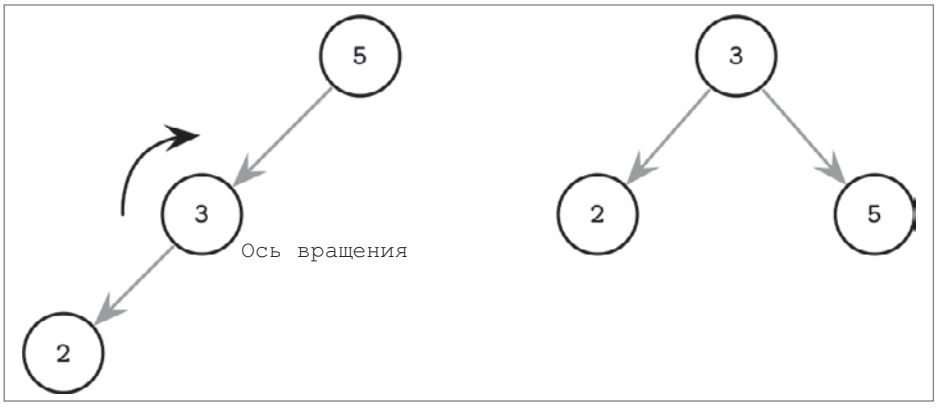

Рис. 2.4. Пример операции поворота

Деревья для дисковых хранилищ

Как уже упоминалось ранее, несбалансированные деревья в худшем случае имеют сложность O(N). Сбалансированные деревья в среднем дают нам сложность $O(\log_2 N)$. В то же время из-за низкой степени ветвления (максимально допустимого числа потомков на узел) нам приходится довольно часто выполнять балансировку, перемещать узлы и обновлять указатели. Повышенные затраты на обслуживание делают нецелесообразным использование двоичных деревьев в качестве дисковых структур данных [NIEVERGELT74].

Если бы мы захотели хранить двоичные деревья на диске, то столкнулись бы с несколькими проблемами. Одной из этих проблем является локальность: поскольку элементы добавляются в случайном порядке, нет никакой гарантии, что вновь созданный узел будет записан близко к своему родителю, а это означает, что указатели на потомков узла могут размещаться с интервалом в несколько страниц диска. Мы

можем в определенной степени улучшить ситуацию, изменив компоновку дерева и используя страничные двоичные деревья (см. врезку «Страничные двоичные деревья» на с. 51).

Другой проблемой, тесно связанной с затратами на следование указателям на потомков, является высота дерева. Поскольку степень ветвления двоичных деревьев равняется всего двум, высота — это двоичный логарифм числа элементов в дереве, и мы должны выполнить $O(\log_2 N)$ операций поиска, чтобы найти искомый элемент, и, следовательно, выполнить столько же дисковых операций. 2-3-деревья и другие деревья с низкой степенью ветвления имеют аналогичное ограничение: хотя их можно использовать в качестве резидентных структур данных, малый размер узлов делает нецелесообразным их использование для хранения на внешних устройствах [COMER79].

Простейшая реализация двоичного дерева поиска на диске потребует столько же операций дискового поиска, сколько будет операций сравнения, поскольку локальность не предусмотрена самой концепцией таких деревьев. Это заставляет нас искать структуру данных, которая обладала бы этим свойством.

Учитывая эти факторы, версия дерева, хорошо подходящая для реализации на диске, должна обладать следующими свойствами:

- *Высокая степень ветвления* для улучшения локальности соседних ключей.
- *Низкая высота* для сокращения количества операций дискового поиска во время обхода.

Степень ветвления и высота находятся в обратной зависимости: чем выше степень ветвления, тем меньше высота. При большой степени ветвления каждый узел может содержать больше потомков, в результате чего уменьшается количество узлов и, следовательно, высота.

Дисковые структуры

Ранее мы в общих чертах обсудили резидентные и дисковые хранилища (см. раздел «Резидентные и дисковые СУБД» на с. 27). Такое же различие можно провести и для конкретных структур данных: некоторые из них лучше подходят для использования на диске, а некоторые лучше работают в оперативной памяти.

Как мы уже говорили, не каждая структура данных, удовлетворяющая требованиям к занимаемому пространству и сложности, может быть эффективно использована для хранения на диске. Структуры данных, используемые в базах данных, должны быть адаптированы с учетом ограничений персистентных носителей.

Дисковые структуры данных часто используются, когда объемы информации настолько велики, что хранение всего набора данных в памяти невозможно или непрактично. В каждый момент может быть кэширована только часть данных, а остальные должны храниться на диске таким образом, чтобы к ним можно было эффективным образом обращаться.

Жесткие диски

Большинство традиционных алгоритмов были разработаны в то время, когда наиболее распространенным персистентным носителем данных были вращающиеся диски, что существенно повлияло на их свойства. Позже новые разработки в области носителей данных, такие как флэш-накопители, вдохновили новые алгоритмы и модификации существующих, использующие возможности нового оборудования. В наши дни появляются новые типы структур данных, оптимизированные для работы с энергонезависимыми хранилищами с байтовой адресацией (например, [XIA17] [KANNAN18]).

Вращающиеся диски повышают затраты на случайное считывание, поскольку они требуют вращения диска и механического движения головки для размещения головки чтения/записи в нужном месте. Однако после выполнения этой затратной части операции чтение или запись непрерывной последовательности байтов требует сравнительно небольших затрат.

Наименьшей единицей передаваемой информации у вращающегося накопителя является сектор, поэтому при выполнении любой операции нужно прочитать или записать как минимум один целый сектор. Размеры секторов обычно варьируют от 512 байт до 4 КБ.

Позиционирование головки — самая затратная часть операции, выполняемой на жестком диске. Это одна из причин, по которой мы часто слышим о положительных эффектах *последовательного ввода-вывода*, т. е. чтения и записи на диске последовательных сегментов памяти.

Твердотельные накопители

Твердотельные накопители (SSD) не имеют движущихся частей: нет диска, который вращается, или головки, которую необходимо позиционировать для чтения. Типичный твердотельный накопитель состоит из *ячеек памяти*, соединенных в *строки* (обычно от 32 до 64 ячеек на строку), строки объединяются в *массивы*, массивы — в *страницы*, а страницы — в *блоки* [LARRIVEE15].

В зависимости от используемой технологии ячейка может содержать один или несколько битов данных. Размер страниц может быть разным в зависимости от устройства, но находится в диапазоне от 2 до 16 Кб. Блоки, как правило, содержат от 64 до 512 страниц. Блоки организованы в *пластины*, и, наконец, отдельные пластины составляют *кристалл*. Твердотельные накопители могут иметь один или несколько кристаллов. Эта структура показана на рис. 2.5.

Минимальная единица, которую можно записать (запрограммировать) или прочитать, — это страница. Однако мы можем вносить изменения только в пустые ячейки памяти (т. е. в те, которые были стерты до записи). Минимальный объект для стирания — это не страница, а блок, содержащий несколько страниц, поэтому его часто называют *блоком стирания*. Страницы в пустом блоке должны записываться последовательно.

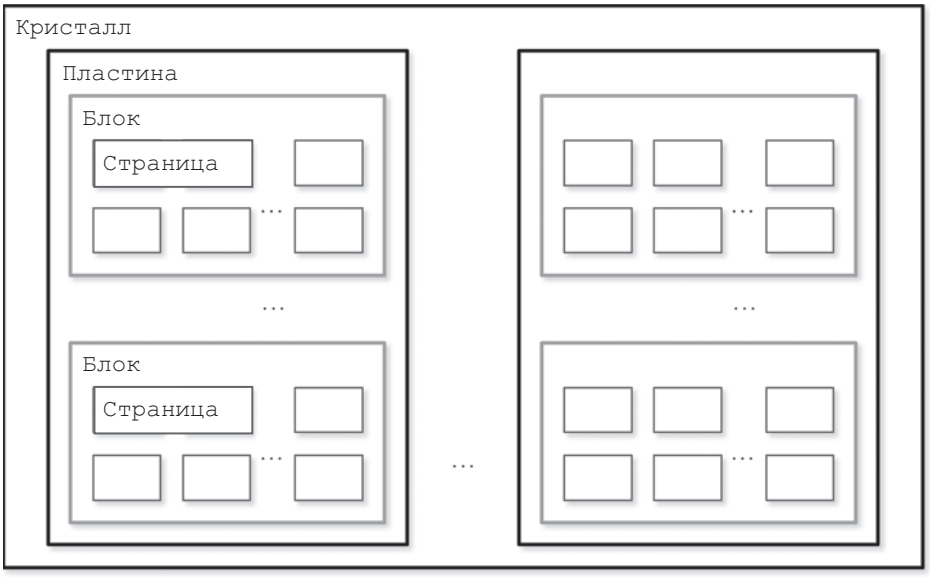

Рис. 2.5. Схема организации SSD

Часть контроллера флэш-памяти, отвечающая за сопоставление идентификаторов страниц с их физическим местоположением, отслеживание пустых, записанных и отброшенных страниц, называется слоем трансляции флэш-памяти (Flash Translation Layer, FTL) (для получения дополнительной информации см. подраздел «Слой преобразования флэш-памяти (FTL)» на с. 176). Данный слой также отвечает за операцию сборки мусора, во время которой он находит блоки, которые можно безопасно стереть. Некоторые блоки все еще могут содержать актуальные страницы. В этом случае FTL перемещает актуальные страницы из этих блоков в новые места и изменяет идентификаторы страниц, чтобы они указывали на их местоположение. После этого он стирает неиспользуемые блоки, делая их доступными для записи.

Поскольку в обоих типах устройств (жестких дисках и твердотельных накопителях) мы адресуем области памяти, а не отдельные байты (т. е. производим доступ к данным поблочно), в большинстве операционных систем имеется абстракция *блочного устройства* [CESATI05]. Она скрывает внутреннюю структуру диска и буферизует операции ввода-вывода, поэтому, когда мы считываем из блочного устройства *одно слово*, полностью считывается *весь содержащий его блок*. Это ограничение нельзя игнорировать и необходимо всегда учитывать при работе с дисковыми структурами данных.

В отличие от жестких дисков, в твердотельных накопителях нет особой разницы между произвольным и последовательным вводом-выводом, поскольку разница в задержках между произвольным и последовательным считыванием не так велика. При этом все же остаются некоторые другие различия, связанные с предварительной выборкой, чтением смежных страниц и внутренним параллелизмом [GOOSSAERT14].

Несмотря на то что сборка мусора обычно является фоновой операцией, ее последствия могут отрицательно сказаться на производительности записи, особенно в случае записи в произвольном порядке и без выравнивания записей по размеру и смещению блока на диске.

Запись только полных блоков и объединение последовательных операций записи в один и тот же блок может помочь сократить количество требуемых операций ввода-вывода. В следующих главах мы обсудим буферизацию и неизменяемость как способы достижения этих целей.

Дисковые структуры

Помимо затрат на сам доступ к диску главным ограничением и условием проектирования для построения эффективных дисковых структур является то, что наименьшей единицей при работе с диском является блок. Чтобы следовать указателю на определенное место в блоке, мы должны извлечь весь блок. Имея такое ограничение, мы можем изменить компоновку структуры данных, чтобы воспользоваться этим как преимуществом.

Мы уже несколько раз упоминали указатели в этой главе, но это слово имеет несколько иную семантику в случае дисковых структур. На диске мы обычно управляем компоновкой данных вручную (за исключением тех случаев, когда используются, к примеру, файлы, отображенные на память (https://databass.dev/links/64)). Это все еще похоже на обычные операции с указателями, но, в отличие от операций в оперативной памяти, мы вынуждены вычислять адреса, на которые направляют указатели, и реализовывать разыменование (т. е. чтение данных, на которые ссылается указатель).

В большинстве случаев смещения (координаты значений на диске) просчитываются заранее, а указатели записываются на диск перед самими данными, на которые указывают. Альтернатива этому подходу — кеширование смещений в памяти перед их сохранением на диск. Создание длинных цепочек зависимостей в дисковых структурах значительно повышает сложность кода и структуры, поэтому рекомендуется сводить к минимуму количество указателей и то, насколько далеко данные находятся от указателя. Например, всегда предпочтительно использовать смещения для адресации указателей, ссылающихся на данные внутри той же страницы, где находится сам указатель. В то же время для ссылки на данные, находящиеся на другой странице, зачастую достаточно всего лишь сослаться на саму страницу.

Таким образом, дисковые структуры разрабатываются с учетом специфики целевого носителя и обычно оптимизируются для меньшего числа обращений к диску. Для этого мы можем улучшить локальность, оптимизировать внутреннее представление структуры и сократить количество указателей, выходящих за границы страницы.

В разделе «Двоичные деревья поиска» на с. 43 мы пришли к выводу, что необходимыми свойствами для оптимальной дисковой структуры являются *высокая степень ветвления* и *малая высота*. Мы также только что обсудили дополнительные затраты, связанные с пространством и указателями, и затраты на обслуживание из-за

повторного сопоставления этих указателей в результате балансировки. B-деревья объединяют все эти идеи: повышают степень ветвления узлов и уменьшают высоту дерева, число указателей узлов и частоту операций балансировки.

СТРАНИЧНЫЕ ДВОИЧНЫЕ ДЕРЕВЬЯ

Построение двоичного дерева путем группировки узлов в страницы, как показано на рис. 2.6, улучшает локальность. Чтобы найти следующий узел, нужно только следовать указателю на уже извлеченной странице. Тем не менее остаются некоторые затраты, связанные с узлами и указателями между ними. Расположение структуры на диске и ее дальнейшее обслуживание — непростая задача, особенно если ключи и значения не сортируются заранее и добавляются в случайном порядке. Балансировка требует реорганизации страницы, что, в свою очередь, приводит к обновлению указателей.

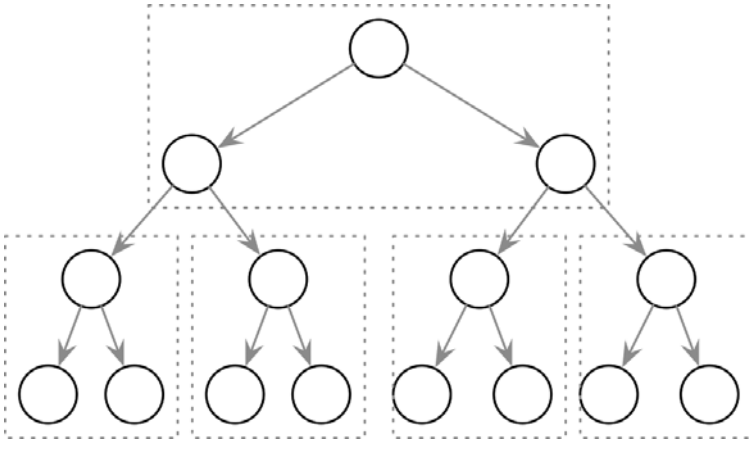

Рис. 2.6. Страничные двоичные деревья

Вездесущие B-деревья

Мы храбрее пчел и... выше деревьев... [1]

Винни-Пух

B-деревья представляют собой что-то вроде огромной комнаты с каталогом в библиотеке: сначала вы должны выбрать правильный шкаф, затем правильную полку в этом шкафу, затем правильный ящик на полке, а затем просмотреть карточки в ящике, чтобы найти искомую. Аналогичным образом B-дерево строит иерархию, которая помогает быстро перемещаться по нему и находить искомые элементы.

[1] В оригинале игра слов «bee» и «tree», обыгрывается B-tree: We are braver than a bee, and a... longer than a tree... — *Примеч. ред.*

Как мы выяснили в разделе «Двоичные деревья поиска» на с. 43, B-деревья строятся на основе сбалансированных деревьев поиска и отличаются тем, что имеют более высокую степень ветвления (предусматривают больше потомков узлов) и меньшую высоту.

В большинстве источников узлы двоичного дерева изображаются в виде кругов. Поскольку каждый узел отвечает только за один ключ и разделяет диапазон на две части, этот уровень детализации достаточен и интуитивно понятен. В то же время узлы B-дерева часто изображаются в виде прямоугольников; при этом блоки указателей также показываются явно, чтобы подчеркнуть связь между потомками и ключами-разделителями. Сходства и различия между узлами двоичного дерева, 2-3-дерева и B-дерева наглядно показаны на рис. 2.7.

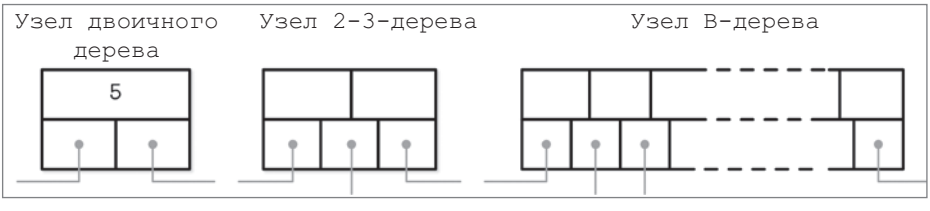

Рис. 2.7. Сравнение узлов двоичного дерева, 2-3-дерева и B-дерева

Конечно, двоичные деревья можно изобразить таким же образом. Обе структуры имеют схожую семантику следования указателям, и различия начинают проявляться в том, как производится балансировка. Это видно на рис. 2.8, который показывает, в чем состоит сходство между двоичными деревьями поиска и B-деревьями: в обоих случаях ключи разделяют дерево на поддеревья и используются для продвижения по дереву и поиска нужных ключей. Вы можете сравнить его с рис. 2.1.

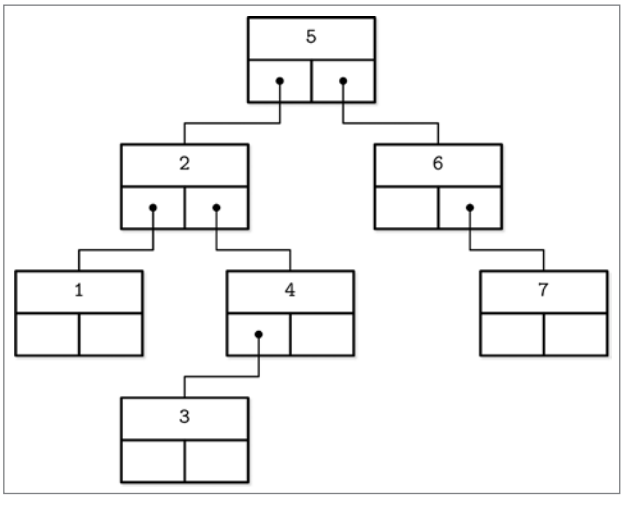

Рис. 2.8. Альтернативное представление двоичного дерева

B-деревья являются *упорядоченными*: ключи внутри узлов B-дерева хранятся в упорядоченном виде. Поэтому чтобы найти искомый ключ, мы можем использовать алгоритм поиска по сортированным данным: двоичный поиск. Это также означает, что поиск в B-деревьях имеет логарифмическую сложность. Например, поиск нужного ключа среди 4 миллиардов (4×10^9) элементов потребует около 32 сравнений (для получения дополнительной информации см. подраздел «Сложность поиска в B-дереве» на с. 55). Если бы нам пришлось производить операцию дискового поиска для всех этих операций сравнения, это значительно замедлило бы поиск, но поскольку узлы B-дерева хранят десятки или даже сотни элементов, нам требуется выполнять лишь одну операцию дискового поиска при каждом переходе между уровнями. Мы обсудим алгоритм поиска более подробно позже в этой главе.

Используя B-деревья, мы можем эффективно выполнять как *точечные запросы*, так и *запросы диапазона*. Точечные запросы, которые в большинстве языков запросов выражаются предикатом равенства (`=`), находят один элемент. А запросы диапазона, выражаемые предикатами сравнения (<, >, ≤, ≥), используются для запроса нескольких элементов данных, расположенных по порядку.

Иерархия B-деревьев

B-деревья состоят из нескольких узлов. Каждый узел содержит до `N` ключей и `N + 1` указателей на потомков. Эти узлы логически сгруппированы в три группы:

Корневой узел

Вершина дерева без родительских узлов.

Листовые узлы

Узлы нижнего уровня, у которых нет дочерних узлов.

Внутренние узлы

Все остальные узлы, соединяющие корень с листьями. Обычно существует более одного уровня внутренних узлов.

Иерархия показана на рис. 2.9.

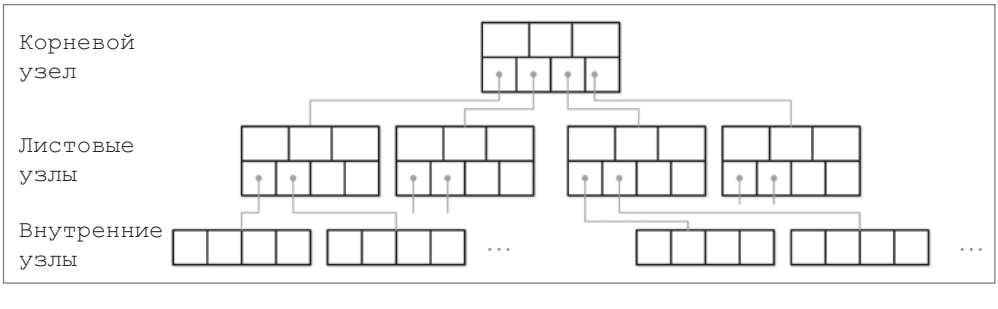

Рис. 2.9. Иерархия B-дерева

Поскольку B-деревья являются методом *постраничной* организации (т. е. они используются для организации и навигации по страницам фиксированного размера), термины *узел* и *страница* часто используются как синонимы.

Зависимость между вместимостью узла и количеством ключей, которые он фактически содержит, называется *заполненностью* (occupancy).

B-деревья характеризуются *степенью ветвления*: количеством ключей, хранящихся в каждом узле. Более высокая степень ветвления помогает снизить затраты на изменения в структуре, необходимые для поддержания дерева в сбалансированном состоянии, а также сократить число операций поиска за счет хранения ключей и указателей на потомков в одном блоке или в нескольких последовательных блоках. Операции балансировки (а именно *разделение* и *слияние*) запускаются, когда узлы заполнены или почти пусты.

B⁺-ДЕРЕВЬЯ

Термин «B-дерево» подразумевает целое семейство структур данных, которые разделяют все или большинство упомянутых свойств. Более точное название для описанной структуры данных — B⁺-дерево. В источнике [KNUTH98] деревья с высокой степенью ветвления называются «многоканальными деревьями».

B-деревья позволяют хранить значения на любом уровне: в корневых, внутренних и листовых узлах. B⁺-деревья хранят значения только в листовых узлах. Внутренние узлы хранят только *ключи-разделители*, используемые для направления алгоритма поиска к соответствующему значению, хранящемуся на листовом уровне.

Поскольку значения в B⁺-деревьях хранятся только на листовом уровне, все операции (вставка, обновление, удаление и извлечение записей данных) затрагивают только листовые узлы, а более высокие уровни затрагиваются только во время разделения и слияния.

B⁺-деревья получили широкое распространение, и мы называем их B-деревьям подобно другим источникам. Например, в источнике [GRAEFE11] B⁺-деревья называются стандартной схемой, а в MySQL InnoDB собственная реализация B⁺-дерева называется B-деревом.

Ключи-разделители

Ключи, хранящиеся в узлах B-дерева, называют *элементами индекса*, *ключами-разделителями* или *ячейками-разделителями*. Они разделяют дерево на *поддеревья* (также называемые *ветвями* или *поддиапазонами*), содержащие соответствующие диапазоны ключей. Ключи хранятся в отсортированном порядке, с тем чтобы было возможно выполнение двоичного поиска. Поддерево находят путем нахождения ключа и следования по соответствующим указателям с верхнего уровня на нижний.

Первый указатель в узле указывает на поддерево, ключи в котором *меньше* первого ключа, а последний указатель в узле указывает на поддерево, ключи в котором *больше* последнего ключа *или равны* ему. Каждый из других указывает на поддерево, ключи в котором находятся в диапазоне между двумя ключами: $K_{i-1} \leq K_s < K_i$,

где K — набор ключей, а K_s — ключ, принадлежащий поддереву. Эти инварианты показаны на рис. 2.10.

Некоторые варианты B-деревьев также предусматривают указатели на узлы с одним родителем (одноуровневые узлы), чаще всего на листовом уровне, чтобы упростить выполнение операции сканирования диапазона. Такие указатели позволяют избежать возвращения к родителю при необходимости найти следующий одноуровневый узел. Некоторые реализации имеют указатели, направленные в обоих направлениях, что в результате дает на листовом уровне двусвязный список и делает возможным обход (итерацию) в обратном порядке .

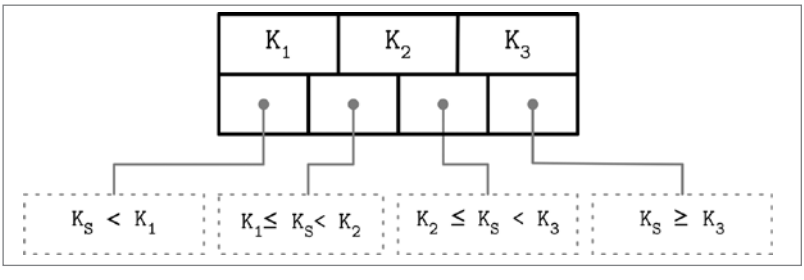

Рис. 2.10. Ключи-разделители разделяют дерево на поддеревья

Главным отличием B-деревьев является то, что они строятся не сверху вниз (как двоичные деревья поиска), а, наоборот снизу вверх. По мере роста числа листовых узлов увеличиваются количество внутренних узлов и высота дерева.

Поскольку B-деревья резервируют дополнительное пространство внутри узлов для будущих вставок и обновлений, коэффициент использования хранилища на базе дерева может составлять лишь 50%, но обычно заметно выше. Более высокая заполненность не оказывает отрицательного влияния на производительность B-дерева.

Сложность поиска в B-дереве

Сложность поиска по B-дереву можно оценивать двумя способами: либо по количеству загрузок блоков, либо по количеству сравнений, выполняемых во время поиска.

Если исходить из количества загрузок, то основание логарифма равняется N (количество ключей на узел). На каждом новом уровне узлов в K раз больше и следование по указателю на потомка уменьшает область поиска в N раз. Во время поиска искомого ключа адресуется не более $\log_K M$ страниц (где M — общее число элементов в B-дереве). Число указателей на потомков, по которым нужно следовать при переходе от корня к листу, также равно числу уровней, или, иначе говоря, высоте дерева *h*.

Если исходить из количества сравнений, то основание логарифма равно 2, так как поиск ключа внутри каждого узла осуществляется путем двоичного поиска. Каждое сравнение делит область поиска пополам, поэтому сложность составляет $\log_2 M$.

Зная разницу между числом запросов и числом сравнений, мы можем получить четкое представление о том, как выполняются операции поиска и как выглядит сложность поиска с обеих точек зрения.

В учебниках и статьях[1] сложность поиска по B-дереву обычно оценивается как `log M`. В этой оценке сложности, как правило, не указывается основание логарифма, поскольку изменение основания ведет лишь к введению некоторого постоянного коэффициента (https://databass.dev/links/65), а умножение на постоянный коэффициент не изменяет сложность. Например, при ненулевом постоянном коэффициенте *c* `O(|c| × n) == O(n)` [KNUTH97].

Алгоритм поиска в B-дереве

Теперь, когда мы рассмотрели структуру и внутреннюю организацию B-деревьев, мы можем описать алгоритмы поиска, вставки и удаления. Чтобы найти элемент в B-дереве, мы должны выполнить один обход от корня до листа. Цель этого поиска — найти искомый ключ или предшествующий ему ключ. Поиск точного совпадения используется при выполнении точечных запросов, обновлений и удалений, а поиск предшествующего ключа — при сканировании диапазонов и выполнении вставки.

Начав с корня, алгоритм выполняет двоичный поиск, сравнивая искомый ключ с ключами, хранящимися в корневом узле, пока не найдет первый ключ-разделитель, превышающий искомое значение. Это позволяет найти искомое поддерево. Как уже говорилось ранее, ключи индекса разбивают дерево на поддеревья с границами между двумя соседними ключами. Найдя поддерево, мы переходим по его указателю и продолжаем тот же процесс поиска (находим разделительный ключ, следуем по указателю), пока не достигнем целевого листового узла, где мы либо находим искомый ключ, либо заключаем, что его нет, найдя предшествующий ему ключ.

На каждом уровне мы получаем более детальное представление дерева: мы начинаем на самом грубом уровне (корень дерева) и спускаемся на следующий уровень, где ключи представляют более точные, детализированные диапазоны, пока наконец не достигаем листьев, которые содержат записи данных.

В ходе точечного запроса поиск завершается, когда найден или не найден искомый ключ. В ходе сканирования диапазона итерирование начинается с ближайшей найденной пары «ключ–значение» и продолжается путем следования по указателям одноуровневых узлов, пока не будет достигнут конец диапазона или исчерпан предикат диапазона.

Подсчет ключей

В литературе описываются различные способы подсчета ключей и смещений потомков. В источнике [BAYER72] предлагается использовать зависящее от устройства натуральное число k, соответствующее оптимальному размеру страницы. Страницы

[1] Например, [KNUTH98].

в этом случае могут содержать от k до 2k ключей, но также могут быть частично заполнены и содержать не менее k + 1 и не более 2k + 1 указателей на дочерние узлы. Корневая страница может содержать от 1 до 2k ключей. Далее вводится число l и говорится, что любая нелистовая страница может иметь l + 1 ключей.

В некоторых других источниках, например [GRAEFE11], говорится, что узлы могут содержать до N ключей-разделителей и N + 1 указателей с аналогичной семантикой и инвариантами.

Оба подхода приводят нас к одному и тому же результату, и различия между ними служат лишь для более удобной подачи материла соответствующих книг. Для ясности в этой книге мы будем считать, что число ключей (или, в случае листовых узлов, пар «ключ–значение») равно N.

Разделение узлов B-дерева

Чтобы вставить значение в B-дерево, мы сначала должны найти целевой лист и точку вставки. Для этого мы используем алгоритм, описанный в предыдущем разделе. После того как лист найден, к нему добавляются ключ и значение. Обновления в B-деревьях осуществляют путем нахождения целевого листового узла с помощью алгоритма поиска и связывания нового значения с существующим ключом.

Если в целевом узле недостаточно свободного места, мы говорим, что узел *переполнен* [NICHOLS66] и должен быть разделен на две части, чтобы вместить новые данные. Если говорить точнее, то узел разделяется, если выполняются следующие условия:

- Для листовых узлов: если узел может вместить до N пар «ключ–значение», а вставка еще одной пары приведет к превышению его максимальной емкости N.
- Для нелистовых узлов: если узел может вместить до N + 1 указателей и вставка еще одного указателя приведет к превышению его максимальной емкости N + 1.

Разделение выполняется путем выделения нового узла, переноса в него половины элементов из разделяемого узла и добавления его первого ключа и указателя на родительский узел. В этом случае мы говорим, что уровень ключа *продвигается по структуре*. Индекс, в котором выполняется разделение, называется *точкой разделения* (или *средней точкой*). Все элементы после точки разделения (включая точку разделения в случае разделения нелистового узла) переносятся во вновь созданный одноуровневый узел, а остальные элементы остаются в разделяемом узле.

Если родительский узел полностью заполнен и не имеет свободного места для ключа, уровень которого необходимо продвинуть вверх по структуре, и указателя на вновь созданный узел, его также необходимо разделить. Эта операция может повторяться рекурсивно до самого корня.

Если при этом исчерпывается емкость дерева (т. е. разделение доходит до самого корня), мы должны разделить корневой узел. При разделении корневого узла выделяется новый корень, содержащий ключ точки разделения. Старый корень (теперь содержащий только половину записей) понижается на следующий уровень вместе

с его вновь созданным одноуровневым узлом, при этом высота дерева увеличивается на единицу. Высота дерева изменяется, когда корневой узел разделяется и выделяется новый корень или когда два узла объединяются, чтобы сформировать новый корень. На уровнях листовых и внутренних узлов дерево растет только горизонтально.

На рис. 2.11 показана вставка нового элемента 11 в полностью заполненный листовой узел. Мы проводим линию посередине заполненного узла, оставляем половину элементов в этом узле, а остальные элементы перемещаем в новый. Значение точки разделения помещается в родительский узел в качестве ключа-разделителя.

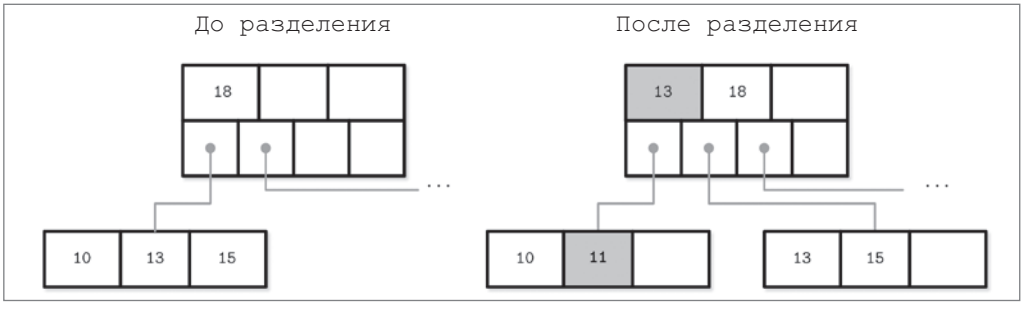

Рис. 2.11. Разделение листового узла при вставке элемента 11. Новый элемент и повышаемый ключ выделены серым цветом

На рис. 2.12 показан процесс разделения полностью заполненного *нелистового* (т. е. корневого или внутреннего) узла при вставке нового элемента 11. Чтобы выполнить разделение, мы сначала создаем новый узел и перемещаем в него элементы, начинающиеся с индекса `N/2 + 1`. Ключ точки разделения повышается до уровня родительского элемента.

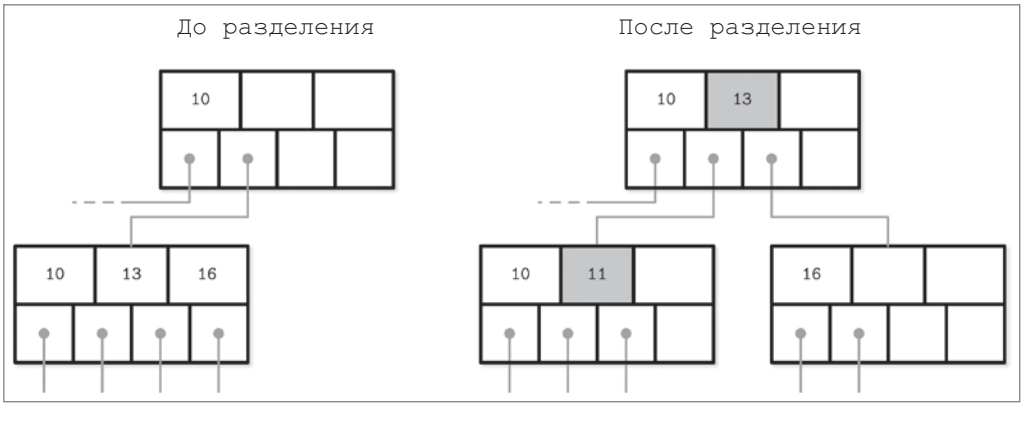

Рис. 2.12. Разделение нелистового узла при вставке элемента 11. Новый элемент и повышаемый ключ выделены серым цветом

Так как разделение нелистовых узлов всегда является следствием разделений, распространяющихся с нижележащих уровней, результатом этой операции является дополнительный указатель, который нужно поместить в узел-родитель разделяемого узла (на вновь созданный узел на следующем уровне). Если у родителя недостаточно места, его тоже нужно разделить.

Как видим, не имеет значения, какой узел разделяется — листовой или нелистовой (т. е. содержит ли разделяемый узел ключи и значения или только ключи). В случае разделения листового узла ключи перемещаются вместе с соответствующими значениями.

После разделения у нас есть два узла, и нужно выбрать узел для вставки. Для этого мы можем использовать инварианты ключа-разделителя. Если вставляемый ключ меньше, чем добавленный узел-родитель, мы завершаем операцию вставкой в разделяемый узел. В противном случае мы производим вставку во вновь созданный узел.

Подводя итог, можно сказать, что разбиение узлов выполняется в четыре этапа:

1. Выделите новый узел.
2. Скопируйте половину элементов из разделяемого узла в новый узел.
3. Поместите новый элемент в соответствующий узел.
4. В родителе разделяемого узла добавьте ключ-разделитель и указатель на новый узел.

Слияние узлов B-дерева

Удаление начинается с определения местоположения целевого листа. После нахождения этого листа производится удаление ключа и связанного с ним значения.

Если соседние узлы имеют слишком мало значений (т. е. их заполненность падает ниже порогового значения), производится слияние одноуровневых узлов. В такой ситуации, мы говорим что узлы недозаполнены. Источник [BAYER72] описывает два сценария недозаполненности: если два соседних узла имеют общего родителя и их содержимое помещается в один узел, их содержимое должно быть слито (соединено); если их содержимое не помещается в один узел, ключи перераспределяются между ними для восстановления баланса (см. раздел «Перебалансировка» на с. 87). Другими словами, два узла сливаются, если выполняются следующие условия:

- Для листовых узлов: если узел может содержать до `N` пар «ключ–значение», а общее число пар «ключ–значение» в двух соседних узлах меньше или равно N.
- Для нелистовых узлов: если узел может содержать до `N + 1` указателей, а общее число указателей в двух соседних узлах меньше или равно `N + 1`.

На рис. 2.13 показан процесс слияния при удалении элемента 16. Для этого мы перемещаем элементы из одного одноуровневого узла в другой. Обычно элементы перемещаются из правого одноуровневого узла в левый, но можно сделать и наоборот при условии, что мы не изменим порядок ключей.

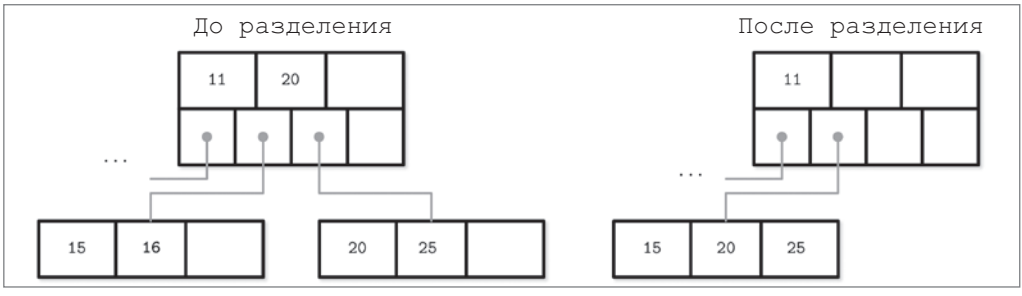

Рис. 2.13. Слияние листовых узлов

На рис. 2.14 показаны два одноуровневых узла, которые необходимо слить при удалении элемента 10. Если мы объединим их элементы, они поместятся в один узел, так что у нас будет один узел вместо двух. В случае слияния нелистовых узлов мы должны изъять соответствующий ключ-разделитель из родительского узла (т. е. понизить его). Количество указателей уменьшается на единицу, так как данное слияние является результатом распространения удаления указателя с нижнего уровня, вызванного удалением страницы. Как и разделение, слияние может распространяться вплоть до корневого уровня.

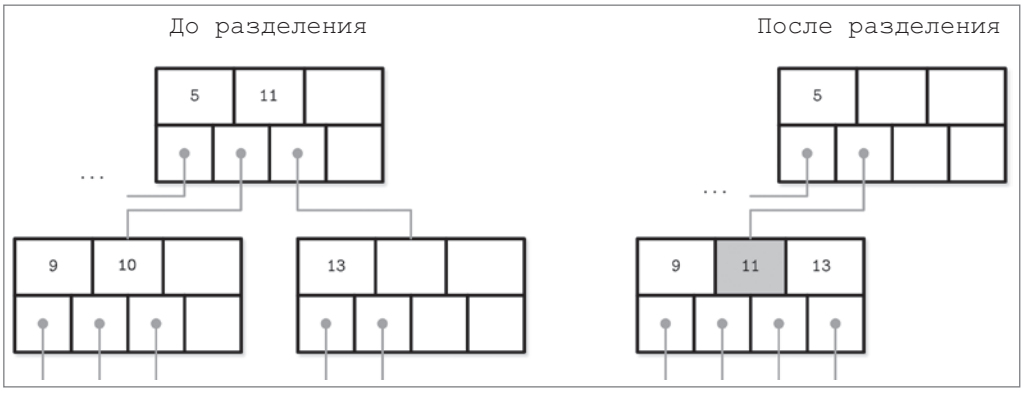

Рис. 2.14. Слияние нелистовых узлов

Подводя итог, можно сказать, что слияние узлов выполняется в три этапа (предполагается, что элемент уже удален):

1. Скопируйте все элементы из правого узла в левый.
2. Удалите указатель на правый узел из родительского узла (или понизьте его в случае слияния нелистовых узлов).
3. Удалите правый узел.

Одним из методов, часто используемых в B-деревьях для уменьшения числа разделений и слияний, является перебалансировка, которую мы обсудим в соответствующем разделе на с. 87.

Итоги

В этой главе мы начали с актуальности создания специализированных структур для дисковых хранилищ. Хотя двоичные деревья поиска могут обладать нужными характеристиками сложности, они все равно не подходят для диска из-за низкой степени ветвления и большого количества требуемых перемещений и обновлений указателей, вызываемых балансировкой. B-деревья решают обе проблемы благодаря увеличению количества элементов, хранящихся в каждом узле (повышению степени ветвления) и уменьшению частоты операций балансировки.

После этого мы обсудили внутреннюю структуру B-дерева и получили общее представление об алгоритмах поиска, вставки и удаления. Операции разделения и слияния позволяют реорганизовывать дерево таким образом, чтобы оно оставалось сбалансированным после добавления и удаления элементов. Мы стараемся минимизировать глубину дерева и добавляем элементы к существующим узлам, пока в них еще есть свободное пространство.

Мы можем использовать эти знания для создания B-деревьев в оперативной памяти. Чтобы создать дисковую реализацию, нам нужно подробно изучить способы расположения узлов B-дерева на диске и составить дисковую компоновку, используя форматы кодирования данных.

Дополнительная литература

Если вы хотите узнать больше о концепциях, упомянутых в этой главе, вы можете обратиться к следующим источникам:

Двоичные деревья поиска

Sedgewick, Robert and Kevin Wayne. 2011. Algorithms (4th Ed.). Boston: Pearson.[1]

Knuth, Donald E. 1997. The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms. Boston: Addison-Wesley Longman.[2]

Алгоритмы разделения и слияния для B-деревьев

Elmasri, Ramez and Shamkant Navathe. 2011. Fundamentals of Database Systems (6th Ed.). Boston: Pearson.

Silberschatz, Abraham, Henry F. Korth, and S. Sudarshan. 2010. Database Systems Concepts (6th Ed.). New York: McGraw-Hill.

[1] *Седжвик Р., Уэйн К.* Алгоритмы на Java. 4-е изд. М.: Вильямс, 2013. — *Примеч. ред.*

[2] *Кнут Д. Э.* Искусство программирования. Т. 2. Получисленные алгоритмы (3-е изд.). М.: Вильямс, 2013. — *Примеч. ред.*

ГЛАВА 3

Форматы файлов

Теперь, когда мы изучили основы B-деревьев, можно посмотреть, как именно B-деревья и другие структуры реализуются на диске. Доступ к диску отличается от доступа к оперативной памяти: с точки зрения разработчика приложений, доступ к памяти почти всегда прозрачен. Благодаря виртуальной памяти [BHATTACHARJEE17] нам не нужно рассчитывать смещения для указателей вручную. Доступ к дискам осуществляется с помощью системных вызовов (см. https://databass.dev/links/54). Как правило, нам необходимо указать смещение (адрес) внутри целевого файла, а затем перевести дисковое представление в форму, подходящую для оперативной памяти.

Это означает, что для достижения эффективности при разработке дисковых структур необходимо учитывать различия между этими формами. Для этого мы должны придумать формат файла, который было бы легко формировать, изменять и интерпретировать. В этой главе мы обсудим те общие принципы и методы, которые могут помочь нам в проектировании не только B-деревьев, но и всех других видов дисковых структур.

Существует много способов реализации B-деревьев, и здесь мы обсудим несколько удобных методов. Детали могут варьировать в зависимости от реализации, но общие принципы остаются неизменными. Вы должны понимать такие базовые механизмы B-деревьев, как разделение и слияние, однако этого еще недостаточно для создания полноценной реализации. Чтобы конечный результат был пригоден для практического использования, он должен сочетать в себе множество различных вещей.

Семантика управления указателями в дисковых структурах несколько отличается от семантики резидентных структур. Дисковые B-деревья можно рассматривать как некоторый механизм управления страницами: алгоритмы должны формировать *страницы* и обеспечивать навигацию по ним. Соответственно необходимо рассчитывать и размещать страницы и указатели на них.

Поскольку большая часть сложности в случае B-деревьев связана с изменяемостью, мы подробно поговорим о компоновке, разделении и перемещении страниц, а также других понятиях, относящихся к изменяемым структурам данных. Позже, когда мы будем говорить о LSM-деревьях (см. раздел «LSM-деревья» на с. 148), мы сосредоточимся на сортировке и обслуживании, так как именно с этими операциями связана большая часть сложности в случае LSM-деревьев.

Актуальность

Создание формата файла имеет много общего с созданием структуры данных на языках с неуправляемой моделью памяти. Мы выделяем блок данных и можем его разделять любым удобным нам способом, используя примитивы и структуры фиксированного размера. Если необходимо сослаться на область памяти большего размера или структуру с переменным размером, мы используем указатели.

Языки с неуправляемой моделью памяти позволяют нам в любой момент выделить дополнительную память (в разумных пределах) без необходимости беспокоиться, есть ли в наличии непрерывный сегмент памяти, фрагментирован ли он или о том, что произойдет после его освобождения. На диске мы сами должны заботиться о сборке мусора и фрагментации.

Компоновка данных не столь важна при их размещении в памяти, как при их размещении на диске. Чтобы дисковая структура данных была эффективной, мы должны расположить данные на диске таким образом, чтобы к ним можно было быстро обращаться, а также учитывать особенности персистентного носителя данных, разработать двоичные форматы данных и найти способы эффективной сериализации и десериализации данных.

Каждый, кто когда-либо использовал такой низкоуровневый язык, как C, без дополнительных библиотек, знает об этих ограничениях. Структуры имеют заранее определенный размер и выделяются и освобождаются явным образом. Выделение и дальнейшее управление памятью вручную представляет еще больше сложностей, поскольку вы можете работать только с сегментами памяти определенного размера и должны отслеживать, какие сегменты уже освобождены, а какие еще используются.

При хранении данных в оперативной памяти большинство проблем с форматом данных отсутствует, легче поддается решению или решается с помощью сторонних библиотек. Например, намного проще реализуется обработка полей переменной длины и данных большого размера, так как мы можем использовать выделение памяти и указатели и нет необходимости компоновать эти данные каким-либо особым образом. В некоторых случаях разработчики и сейчас разрабатывают специализированные форматы данных для оперативной памяти, чтобы использовать преимущества строк кэша процессора, предварительной выборки и других аппаратных особенностей, хотя это обычно делается лишь для целей оптимизации [FOWLER11].

Несмотря на то что часть функций берут на себя операционная и файловая системы, реализация дисковых структур требует большего внимания к деталям и скрывает в себе больше подводных камней.

Двоичное кодирование

Для эффективного хранения данных на диске их необходимо закодировать в компактный и удобный для сериализации и десериализации формат. Когда речь заходит

о двоичных форматах, вы довольно часто слышите слово «макет» (layout). Поскольку при работе с бинарными данными на диске можно использовать только примитивы чтения и записи `read` и `write`, но не примитивы выделения и освобождения памяти `malloc` и `free`, обращаться к данным приходится по-другому, что требует соответствующей подготовки данных.

Здесь мы обсудим основные принципы эффективной компоновки макета страниц. Эти принципы справедливы для любого двоичного формата: их можно с успехом использовать и при создании файловых форматов, форматов сериализации и протоколов обмена данными.

Чтобы мы могли организовать записи в страницы, необходимо сначала разобраться с тем, как производится представление ключей и записей в двоичном формате, объединение нескольких значений в более сложные структуры и реализация типов без фиксированной длины (другими словами, типов переменного размера) и массивов.

Примитивные типы

Ключи и значения обладают некоторым *типом*, таким как `integer` (целое число), `date` (дата) или `string` (строка), и могут быть представлены (сериализованы и десериализованы) в необработанном двоичном виде.

Большинство числовых типов данных представляется в виде значений фиксированного размера. При работе с многобайтовыми числовыми значениями важно использовать один и тот же *порядок байтов* как при кодировании, так и при декодировании. Существуют две разновидности порядка байтов.

Big-endian — порядок от старшего к младшему

Последовательность начинается со старшего байта (MSB), за которым следуют байты в порядке убывания значимости. Другими словами, старший байт имеет наименьший адрес.

Little-endian — порядок от младшего к старшему

Последовательность начинается с младшего байта (LSB), за которым следуют байты в порядке возрастания значимости.

Разницу между этими способами демонстрирует рис. 3.1. Здесь показаны оба способа представления шестнадцатеричного 32-разрядного целого числа `0xAABBCCDD`, где `AA` — старший байт.

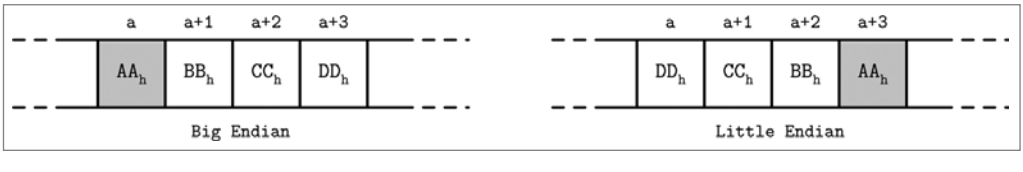

Рис. 3.1. Прямой и обратный порядок байтов. Старший байт выделен серым цветом. Адреса, обозначенные буквой a, увеличиваются слева направо

Например, чтобы воссоздать 64-разрядное целое число с соответствующим порядком байтов, в СУБД RocksDB предусмотрены специфичные для платформы определения, которые помогают идентифицировать порядок байтов целевой платформы (https://databass.dev/links/55)[1]. Если порядок байтов целевой платформы не совпадает с порядком байтов рассматриваемого значения (что определяется путем сравнения значения переменной `kLittleEndian` с порядком байтов рассматриваемого значения с помощью функции `EncodeFixed64WithEndian` (https://databass.dev/links/56)), то порядок байтов инвертируется с помощью функции `EndianTransform` (https://databass.dev/links/57), которая считывает байты значения в обратном порядке и добавляет их к результату.

Записи в базах данных включают в себя такие примитивы, как числа, строки и логические значения, а также их комбинации. Однако при передаче данных по Сети или сохранении их на диске мы можем использовать только байтовые последовательности. Это означает, что для отправки или сохранения записи необходимо производить *сериализацию* (преобразование записи в интерпретируемую последовательность байтов), а чтобы запись можно было использовать после ее получения или считывания, необходимо производить *десериализацию* (обратное преобразование последовательности байтов в исходную запись).

Работая с двоичными форматами данных, мы всегда начинаем компоновку с примитивов, которые служат строительными блоками для более сложных структур. Различные числовые типы могут различаться по размеру: `byte` содержит 8 бит, `short` — 2 байта (16 бит), `int` — 4 байта (32 бита), а `long` — 8 байт (64 бита).

Числа с плавающей точкой (такие, как `float` и `double`) представлены знаком числа, мантиссой и порядком (показателем степени). Широко распространенный способ представления чисел с плавающей точкой описан в стандарте IEEE для двоичной арифметики с плавающей точкой (IEEE 754) (https://ieeexplore.ieee.org/document/30711). 32-разрядный тип `float` содержит число с одинарной точностью. Например, двоичное представление числа с плавающей точкой 0,15652 выглядит так, как показано на рис. 3.2. Первые 23 бита представляют мантиссу, следующие 8 бит — порядок (показатель степени) и еще один бит — знак числа (и в случае положительного, и в случае отрицательного числа).

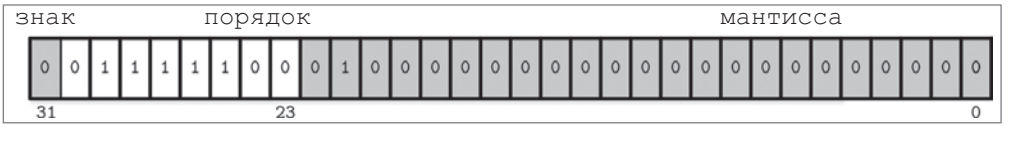

Рис. 3.2. Двоичное представление числа с плавающей точкой одинарной точности

Поскольку расчет чисел с плавающей точкой производится с использованием дробей, представленный таким образом результат является приближенным. Подробное об-

[1] С помощью переменной `kLittleEndian` указывается, поддерживает ли платформа (macOS, Solaris, Aix, один из вариантов BSD или Windows) прямой порядок байтов.

суждение алгоритма преобразования выходит за рамки этой книги, поскольку здесь мы рассматриваем лишь основные принципы представления данных.

Тип `double` служит для представления чисел с плавающей запятой двойной точности [SAVARD05]. Стандартные библиотеки большинства языков программирования содержат средства для кодирования и декодирования чисел с плавающей запятой в их двоичное представление и из него.

Строки и данные переменного размера

Все примитивные числовые типы имеют фиксированный размер. Составление более сложных типов во многом напоминает работу с типом `struct`[1] в языке C. Вы можете объединять примитивные типы в структуры и использовать массивы фиксированного размера или указатели на другие области памяти.

Строки и другие типы данных переменного размера (например, массивы данных фиксированного размера) могут быть сериализованы в виде числа, представляющего длину массива или строки, за которым следует соответствующее количество байтов — фактических данных. В случае строк такой способ представления часто называют строкой UCSD (Калифорнийский университет в Сан-Диего) или строкой Pascal (https://databass.dev/links/59) в честь популярной реализации языка программирования Паскаль. Используя псевдокод, это можно выразить следующим образом:

```
String
{
    size    uint_16
    data    byte[size]
}
```

Альтернативой строкам Pascal являются *строки с нулевым окончанием*: здесь строка считывается побайтово до тех пор, пока не будет достигнут символ конца строки. Подход строк Pascal имеет ряд преимуществ: вы можете определять длину строки за постоянное время, вместо того чтобы перебирать содержимое строки, а специфичную для языка строку можно сформировать путем извлечения соответствующего количества байтов из памяти и передачи массива байтов конструктору строки.

Побитно упакованные данные: логические значения, перечисления и флаги

Логические значения могут быть представлены либо с помощью одного байта, либо путем представления значений `true` (истина) и `false` (ложь) в виде `1` и `0`. Поскольку логическая переменная имеет только два значения, использование целого байта для ее представления является расточительным, поэтому разработчики часто упаковывают

[1] Стоит отметить, что компиляторы могут добавлять к структурам заполняющие биты и байты, причем это также зависит от архитектуры. Такое добавление может сделать отличным от предполагаемого реальное смещение и расположение байтов. Подробнее об упаковке структур можно прочитать по ссылке: https://databass.dev/links/58.

логические значения в группы по восемь значений, где каждое значение занимает только один бит. Биты со значением `1` принято называть «установленными», а биты со значением `0` — «сброшенными» или «пустыми».

Перечисляемые типы (https://databass.dev/links/60), или, как их еще называют, *перечисления*, могут быть представлены в виде целых чисел и часто используются в двоичных форматах и протоколах обмена данными. Перечисления используются для представления множеств низкой мощности с часто повторяющимися элементами. Например, с помощью перечисления можно представить тип узла B-дерева:

```
enum NodeType {
   ROOT,     // 0x00h, корень
   INTERNAL, // 0x01h, внутренний
   LEAF      // 0x02h, лист
};
```

Еще одной схожей концепцией являются *флаги*, которые представляют собой своего рода комбинацию упакованных логических значений и перечислений. Флаги могут представлять невзаимоисключающие именованные логические параметры. Например, с помощью флагов можно указать, содержит ли страница ячейки со значениями, обладают ли эти значения фиксированным или переменным размером и связаны ли с этим узлом страницы переполнения. Поскольку каждый бит представляет собой значение флага, мы можем использовать в качестве маски только числа, равные степени двойки (поскольку эти числа всегда имеют в двоичном коде только один установленный бит, например: 2^3 `== 8 == 1000b`, 2^4 `== 16 == 0001 0000b` и т. д.):

```
int IS_LEAF_MASK         = 0x01h; //бит № 1
int VARIABLE_SIZE_VALUES = 0x02h; //бит № 2
int HAS_OVERFLOW_PAGES   = 0x04h; //бит № 3
```

Как и упакованные логические значения, значения флагов можно считывать и записывать в упакованном числе с помощью *битовых масок* и побитовых операторов. Например, чтобы установить бит, отвечающий за один из флагов, мы можем использовать побитовое ИЛИ (`|`) и битовую маску. Вместо битовой маски также можно использовать *побитовый сдвиг* (`<<`) и битовый индекс. Чтобы сбросить бит, можно использовать побитовое И (`&`) и побитовый оператор отрицания (`~`). Чтобы проверить, установлен ли бит `n`, можно сравнить результат побитового И с `0`:

```
// Установка бита
flags |= HAS_OVERFLOW_PAGES;
flags |= (1 << 2);

// Сброс бита
flags &= ~HAS_OVERFLOW_PAGES;
flags &= ~(1 << 2);

// Проверяем, установлен ли бит
is_set = (flags & HAS_OVERFLOW_PAGES) != 0;
is_set = (flags & (1 << 2)) != 0;
```

Основные принципы

Обычно разработка формата файла начинается с решения о том, как будет осуществляться адресация: будет ли файл разделен на страницы одинакового размера, представленные одним блоком или несколькими смежными блоками. Большинство структур хранения с обновлением на месте используют страницы постоянного размера, так как это значительно упрощает доступ для чтения и записи. В структурах хранения, доступных только для добавления, данные тоже часто записываются постранично: записи добавляются одна за другой, и после заполнения страницы в памяти она выгружается на диск.

Файл обычно начинается с *заголовка* фиксированного размера и может заканчиваться *трейлером* фиксированного размера, содержащим вспомогательную информацию, к которой необходимо обеспечить быстрый доступ или которая необходима для декодирования остальной части файла. Остальная часть файла разбита на страницы. Схема такой организации файлов показана на рис. 3.3.

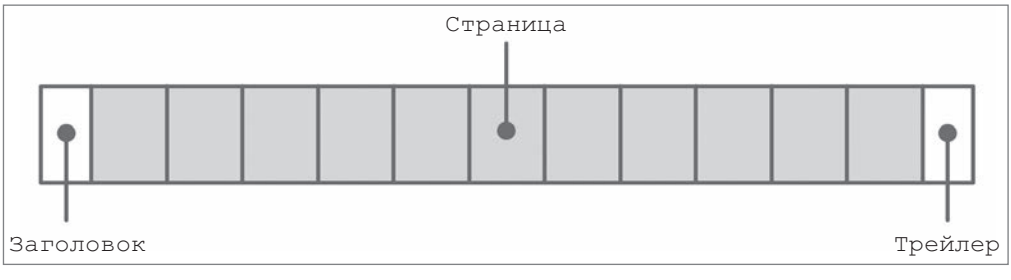

Рис. 3.3. Организация файлов

Во многих хранилищах, схема данных является фиксированной. В таких случаях схема определяет количество, порядок и тип полей, которые может содержать таблица. Использование фиксированной схемы позволяет уменьшить объем хранящихся на диске данных: вместо многократной записи имен полей мы можем использовать указатели на их позиции.

Если бы нам нужно было разработать формат для каталога компании, хранящего имена, даты рождения, налоговые номера и пол каждого сотрудника, мы могли бы использовать несколько подходов. Мы могли бы сохранять поля фиксированного размера (например, поля для даты рождения и налогового номера) в начале структуры, а следом за ними размещать поля переменного размера:

```
Поля фиксированного размера:
| (4 байта)      employee_id          |
| (4 байта)      tax_number           |
| (3 байта)      date                 |
```

```
| (1 байт)       gender               |
| (2 байта)      first_name_length    |
| (2 байта)      last_name_length     |

Поля переменного размера:
| (first_name_length байт) first_name |
| (last_name_length байт) last_name   |
```

Теперь для доступа к полю `first_name` мы можем срезать `first_name_length` байт, следующих за областью фиксированного размера. Для доступа к полю `last_name` мы можем найти его начальную позицию, узнав размеры предшествующих полей переменной величины. Чтобы избежать вычислений, включающих несколько полей, мы можем сохранять и смещение, и длину в области фиксированного размера. В этом случае мы сможем рассчитать местоположения каждого поля переменного размера по отдельности.

Более сложные структуры обычно создаются с использованием определенной иерархии: поля образуются из примитивов, ячейки — из полей, страницы — из ячеек, разделы — из страниц, области — из разделов и т. д. Здесь нет строгих правил, которым вы должны следовать, и все зависит от того, для каких данных вы создаете формат.

Файлы базы данных часто состоят из нескольких частей, при этом в осуществлении навигации помогает таблица подстановки, указывающая на начальные смещения этих частей, записанные либо в заголовке или трейлере того же файла, либо в отдельном файле.

Структура страницы

СУБД хранят записи в файлах данных и индексных файлах. Эти файлы разбиты на секции фиксированного размера, называемые *страницами*, размер которых часто составляет несколько блоков файловой системы. Размер страниц обычно находится в диапазоне от 4 до 16 килобайт.

Давайте рассмотрим в качестве примера узел дискового B-дерева. С точки зрения структуры в B-деревьях мы различаем *листовые узлы*, которые содержат пары ключ/запись данных, и *нелистовые узлы*, которые содержат ключи и указатели на другие узлы. Каждый узел B-дерева занимает одну страницу или несколько связанных друг с другом страниц, поэтому в контексте B-деревьев термины *узел* и *страница* (и даже *блок*) часто используются как синонимы.

В статье с исходным описанием B-деревьев [BAYER72] описывается простейший способ организации страниц для записей данных фиксированного размера, при котором каждая страница представляет собой просто последовательное соединение из трех

элементов, как показано на рис. 3.4. Здесь буквой k обозначены ключи, буквой v — соответствующие значения и буквой p — указатели на дочерние страницы.

Рис. 3.4. Способ организации страниц для записей фиксированного размера

Этот подход легко поддается реализации, однако у него есть и недостатки:

- При добавлении ключа не справа, а в каком-либо другом месте потребуется перемещать элементы.
- Данный способ хорошо подходит только для данных фиксированного размера и не может обеспечить эффективное управление и доступ к данным в случае записей переменного размера.

Слоттированные страницы

При сохранении записей переменного размера основной проблемой является управление свободным пространством, т. е. высвобождение пространства, занятого удаляемыми записями. Если мы попытаемся поместить запись размера n в пространство, ранее занятое записью размера m, то если m не равно n и мы не найдем еще одну запись с размером m – n, это пространство останется неиспользованным. Аналогично сегмент размера m нельзя использовать для сохранения записи размера k, если k больше m, и потому такая вставка производится без высвобождения неиспользуемого пространства.

Чтобы упростить управление пространством в случае записей переменного размера, мы можем разделить страницу на сегменты фиксированного размера. Однако и такой подход в конечном итоге приведет к потере пространства. Например, при использовании сегментов размером 64 байта, если размер записи не кратен 64, мы теряем 64 – (остаток от деления n на 64) байтов, где n — размер вставляемой записи. Другими словами, если размер записи не кратен 64, один из блоков будет заполнен только частично.

Высвобождение пространства можно произвести путем простого перезаписывания страницы с перемещением записей, но при этом нужно будет сохранить смещения записей, так как их могут использовать указатели, ведущие на страницу из-за ее пределов. Это тоже желательно делать с минимальными потерями пространства.

Подводя итог, можно сказать, что нам нужен формат страницы, который позволяет:

- Сохранять записи переменного размера с минимальными накладными расходами.
- Высвобождать место, занятое удаляемыми записями.
- Ссылаться на записи страницы независимо от их точного расположения.

Для эффективного сохранения записей переменного размера, таких как строки, большие двоичные объекты и т. д., можно использовать такой способ организации, как *слоттированная страница* (т. е. страница со слотами, slotted page) [SILBERSCHATZ10] или *каталог слотов* [RAMAKRISHNAN03]. Этот подход используется во многих базах данных, например в PostgreSQL (https://databass.dev/links/61).

При этом страница организуется в виде набора *слотов* или *ячеек* с отведением для указателей и ячеек двух независимых областей памяти, расположенных в разных частях страницы. Это означает, что для сохранения порядка достаточно производить реорганизацию указателей, обращающихся к ячейкам, а удалить запись можно путем обнуления либо удаления ее указателя.

Слоттированная страница снабжается заголовком фиксированного размера с важной информацией о странице и ячейках (см. раздел «Заголовок страницы» на с. 79). Ячейки могут различаться по размеру и содержать произвольные данные: ключи, указатели, записи данных и т. д. На рис. 3.5 показан способ организации слоттированных страниц, при котором каждая страница имеет служебную область (заголовок), ячейки и указатели на них.

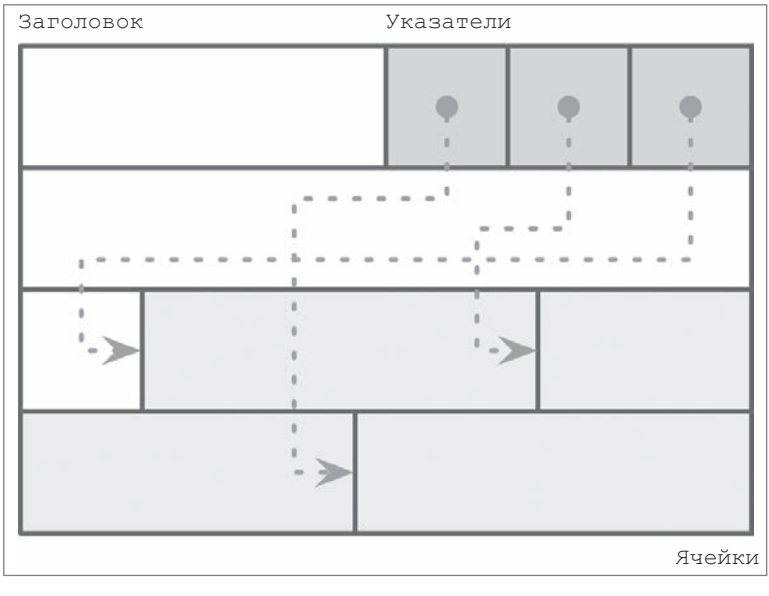

Рис. 3.5. Слоттированная страница

Давайте посмотрим, как такой подход устраняет проблемы, о которых мы говорили в начале этого раздела:

- Минимальные накладные расходы: единственные накладные расходы при использовании слоттированных страниц — это массив указателей, содержащий смещения до точного места размещения записей.

- Высвобождение пространства: пространство можно восстановить путем дефрагментации и перезаписи страницы.
- Динамическая компоновка: за пределами страницы обращение к слотам производится только посредством идентификаторов, поэтому данные о точном местоположении скрыты внутри страницы.

Структура ячеек

Используя флаги, перечисления и примитивные типы, мы можем начать проектировать структуру ячеек, а затем объединить ячейки в страницы и составить дерево из страниц. На уровне ячеек мы различаем ячейки, содержащие ключи, и ячейки, содержащие пары «ключ–значение». Ячейка с ключами содержит ключ-разделитель и указатель на страницу, расположенную *между* двумя соседними указателями. Ячейки с парами «ключ–значение» содержат ключи и связанные с ними данные.

Подразумевается, что все ячейки внутри страницы являются однородными (например, все ячейки содержат только ключи или только пары «ключ–значение»; либо аналогичным образом все ячейки содержат только данные фиксированного размера или только данные переменного размера, но не смесь и того и другого). Это означает, что метаданные, описывающие ячейки, достаточно сохранять только на уровне страницы, а не дублировать их в каждой ячейке.

Для формирования ячейки с ключом нам нужно знать:

- Тип ячейки (который можно определить в метаданных самой страницы).
- Размер ключа.
- Идентификатор дочерней страницы, на которую указывает эта ячейка.
- Байты ключа.

Структура ячейки с ключом переменного размера может выглядеть примерно так (у ячейки с данными фиксированного размера размер ключа будет опущен):

```
0                4                8
+----------------+---------------+-------------+
| [int] key_size | [int] page_id | [bytes] key |
+----------------+---------------+-------------+
```

Мы сгруппировали вместе поля данных фиксированного размера, разместив за ними `key_size` байтов. Конечно, группировать поля таким образом не обязательно, но такой подход упрощает вычисление смещений за счет того, что ко всем полям фиксированного размера можно будет обращаться с помощью статических, заранее вычисленных смещений, а вычислять придется только смещения для данных переменного размера.

Ячейки с парами «ключ–значение» отличаются тем, что содержат записи данных вместо идентификаторов дочерних страниц. В остальном их структура аналогична:

- Тип ячейки (можно вывести логическим путем из метаданных страницы).
- Размер ключа.
- Размер значения.
- Байты ключа.
- Байты записи данных.

```
0              1               5 ...
+--------------+---------------+
| [byte] flags | [int] key_size |
+--------------+---------------+

5                  9               .. + key_size
+------------------+---------------+----------------------+
| [int] value_size |  [bytes] key  | [bytes] data_record  |
+------------------+---------------+----------------------+
```

Чтобы сослаться на ячейку, нам достаточно указать ее *смещение* (offset), а для адресации страницы мы обычно используем *идентификаторы страниц*. Поскольку страницы имеют фиксированный размер и управляются кэшем страниц (см. раздел «Организация буферизации данных» на с. 98), нам достаточно сохранять только идентификатор страницы, который впоследствии преобразуется в фактическое смещение в файле с помощью таблицы подстановки. *Смещения ячеек* представляют собой локальное смещение в пределах страницы, отсчитываемое от ее начала; это позволяет использовать меньшее количество элементов и тем самым сделать представление более компактным.

ДАННЫЕ ПЕРЕМЕННОГО РАЗМЕРА

Ключ и значение в ячейке необязательно должны иметь фиксированный размер. Размеры ключей и значений могут быть произвольными, и от нас не требуется предопределять эти размеры заранее. Их местоположение можно вычислять из заголовка ячейки фиксированного размера, используя смещения.

При работе с данными переменного размера, для того чтобы узнать смещение, на котором начинается значение ключа, нам нужно пропустить байты, которые занимает заголовок, и считать `key_size` байт. Аналогичным образом, чтобы рассчитать смещение, на котором находится значение, нужно пропустить заголовок и `key_size` байт и прочитать `value_size` байт.

Существуют и другие способы работы с такими данными. Например, можно сохранять общий размер данных и вычислять размер значения путем вычитания. Главное — располагать достаточной информацией для того, чтобы можно было разделить ячейку на отдельные части и затем восстановить закодированные данные.

Объединение ячеек в слоттированные страницы

Чтобы организовать ячейки в страницы, мы можем использовать метод слоттированных страниц, который мы обсуждали в разделе «Структура страницы» на с. 69. В этом

случае ячейки добавляются в правую часть страницы (ближе к ее концу) с сохранением смещений/указателей ячеек в левой части страницы, как показано на рис. 3.6.

И значения при этом не обязательно вставлять по порядку. При вставке новые ячейки будут добавлены в свободное место, без реорганизации ячеек, а их логический порядок будет поддерживаться с помощью пересортировки их смещений. Такая структура позволяет добавлять ячейки на страницу с минимальными усилиями, так как ячейки не нужно перемещать во время операций вставки, обновления или удаления.

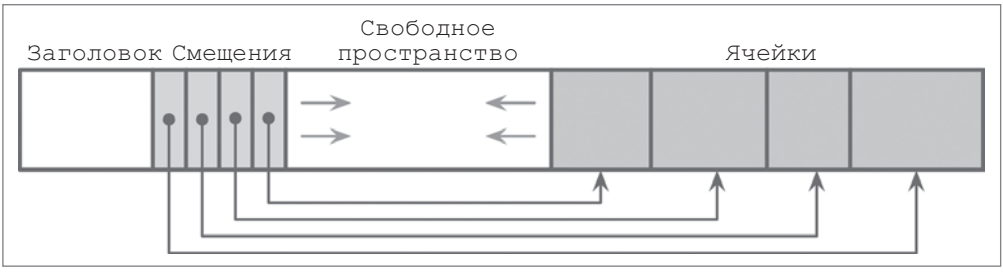

Рис. 3.6. Смещение и направление роста ячеек

Давайте рассмотрим пример страницы, содержащей имена. На страницу добавляются два имени; при этом сначала вставляется имя «Том», а затем имя «Лесли». Как видно из рис. 3.7, их *логический порядок* (в данном случае алфавитный) *не совпадает* с их *порядком вставки* (порядком их добавления на страницу). При этом ячейки расположены в порядке вставки, а смещения пересортировываются таким образом, чтобы можно было использовать двоичный поиск.

Рис. 3.7. Записи, добавленные в произвольном порядке: «Том», «Лесли»

Допустим, что нам нужно добавить на эту страницу еще одно имя: «Рон». Новые данные добавляются у верхней границы свободного пространства страницы, но смещения ячеек должны сохранять лексикографический порядок ключей: «Лесли», «Рон», «Том». Для этого мы должны пересортировать смещения ячеек: мы смещаем вправо указатели, расположенные после *точки вставки*, чтобы освободить место для нового указателя на ячейку с именем «Рон», как показано на рис. 3.8.

Рис. 3.8. Добавление еще одной записи: «Рон»

Управление данными переменного размера

Для удаления элемента на странице не требуется удалять саму ячейку и смещать другие ячейки для заполнения освободившегося пространства. Вместо этого можно пометить ячейку как удаленную и обновить размещенный в памяти *список доступности*, указав объем освободившейся памяти и предоставив указатель на освобожденную область. В списке доступности хранятся смещения освобожденных сегментов и их размеры. Вставляя новую ячейку, мы сначала проверяем список доступности, чтобы определить, есть ли сегмент, в который она может поместиться. Пример фрагментированной страницы с доступными сегментами приведен на рис. 3.9.

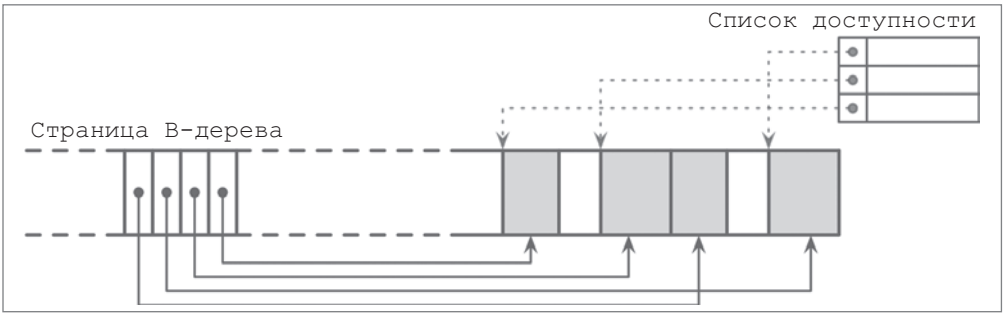

Рис. 3.9. Фрагментированная страница и список доступности. Занятые страницы выделены серым цветом. Пунктирные линии представляют собой указатели на незанятые области памяти из списка доступности

В СУБД SQLite незанятые сегменты называются *свободными блоками* (freeblock) (https://databass.dev/links/62), и указатель на первый свободный блок сохраняется в заголовке страницы. Кроме того, внутри страницы сохраняется общее количество доступных байтов, чтобы можно было быстро проверить, поместится ли новый элемент на странице после дефрагментации.

При поиске подходящей области памяти используются следующие стратегии:

Первый подходящий

Может привести к увеличению накладных расходов, так как пространство, оставшееся после размещения данных в первом подходящем сегменте, часто

оказывается слишком малым для размещения какой-либо другой ячейки и, таким образом, фактически тратится впустую.

Наиболее подходящий

Для более эффективного использования пространства можно попытаться найти сегмент, при использовании которого останется наименьший остаток.

Если не получается найти достаточное количество последовательных байтов для размещения новой ячейки, но при этом имеется достаточное количество фрагментированных байтов, существующие ячейки считываются и перезаписываются — таким образом производится дефрагментация страницы и освобождается место для новых записей. Если свободного места не хватает даже после дефрагментации, мы должны создать страницу переполнения (см. подраздел «Страницы переполнения» на с. 83).

Чтобы повысить степень локальности (особенно когда используются ключи небольшого размера), в некоторых реализациях ключи и значения сохраняются отдельно друг от друга на листовом уровне. Сохранение ключей в одном месте повышает степень локальности, тем самым ускоряя поиск. После нахождения искомого ключа его значение можно найти в ячейке со значением с соответствующим индексом. В случае ключей переменного размера при этом требуется вычислять и сохранять дополнительный указатель на ячейку со значением.

Подведем итоги. Чтобы упростить структуру B-дерева, мы предполагаем, что каждый узел занимает одну страницу. Страница состоит из заголовка фиксированного размера, блока указателей на ячейки и ячеек. Ячейки содержат ключи и указатели на страницы, представляющие дочерние узлы или соответствующие записи данных. В B-деревьях используется простая иерархия указателей: идентификаторы страниц для поиска дочерних узлов в файле дерева и смещения ячеек для поиска ячеек внутри страницы.

Управление версиями

СУБД постоянно развиваются, и разработчики работают над добавлением функций, исправлением ошибок и проблем с производительностью. В результате этого двоичный формат файла может измениться. В большинстве случаев любая версия подсистемы хранения должна поддерживать несколько форматов сериализации (например, текущий и один или несколько устаревших форматов для обратной совместимости). При этом у нас должна быть возможность узнать, с какой версией файла мы имеем дело.

Такую возможность можно обеспечить несколькими способами. Например, СУБД Apache Cassandra использует префиксы версий в именах файлов. Это позволяет определить версию файла, даже не открывая его. Начиная с версии 4.0, имя файла данных имеет префикс `na`, например: `na-1-big-Data.db`. Более ранние файлы имеют другие префиксы: так, файлы версии 3.0 имеют префикс `ma`.

Кроме того, сведения о версии могут сохраняться в отдельном файле. Например, СУБД PostgreSQL (https://databass.dev/links/63) сохраняет номер версии в файле PG_VERSION.

Сведения о версии также могут сохраняться непосредственно в заголовке индексного файла. В этом случае часть заголовка (или весь заголовок) должна кодироваться в формате, не меняющемся от версии к версии. Выяснив, для какой версии закодирован файл, мы можем создать рассчитанный на нее итератор для интерпретации содержимого. Некоторые форматы файлов идентифицируют версию с помощью системных кодов, которые мы рассмотрим более подробно в подразделе «Системные коды» на с. 79.

Вычисление контрольной суммы

Файлы на диске могут быть повреждены или искажены в результате программных ошибок и аппаратных сбоев. Чтобы сразу же выявлять эти проблемы, исключая распространение поврежденных данных в другие подсистемы или узлы, мы можем использовать контрольные суммы и циклический избыточный контроль (cyclic redundancy check, CRC).

Некоторые источники не делают различий между криптографическими и некриптографическими хэш-функциями, CRC и контрольными суммами. Все эти методы объединяет то, что они сводят значительный объем данных к некоторому небольшому числу, однако они различаются в части сценариев использования, назначения и предоставляемых гарантий.

Контрольные суммы дают самые слабые гарантии и не позволяют выявлять одновременное повреждение нескольких битов. Они обычно вычисляются путем использования исключающего ИЛИ в сочетании с проверкой четности или суммированием [KOOPMAN15].

Циклический избыточный контроль (CRC) часто помогает выявлять пакеты ошибок (например, случаи повреждения нескольких последовательных битов), и его реализации обычно используют таблицы подстановки и полиномиальное деление [STONE98]. Обнаружение многобитных ошибок играет крайне важную роль, поскольку именно здесь проявляется значительная часть сбоев, происходящих в коммуникационных сетях и устройствах хранения.

Некриптографические хэш-функции и CRC не стоит использовать для проверки данных на предмет постороннего вмешательства. Для этого всегда следует использовать сильные криптографические хэш-функции, предназначенные для обеспечения безопасности. CRC предназначен главным образом для проверки данных на отсутствие непреднамеренных и случайных изменений. Эти алгоритмы не предназначены для защиты от атак и преднамеренного изменения данных.

Перед записью данных на диск мы вычисляем их контрольную сумму и записываем ее вместе с данными. При чтении данных мы снова вычисляем контрольную сумму и сравниваем ее с записанной. Если контрольные суммы не совпадают, значит, произошло повреждение и мы не должны использовать считанные данные.

Поскольку вычисление контрольной суммы для всего файла часто не имеет практического смысла, ведь очень маловероятно, что при обращении к файлу нам придется считывать содержимое файла полностью , контрольные суммы обычно вычисляются на страницах и помещаются в их заголовки. Благодаря этому повышается надежность контрольных сумм (поскольку они вычисляются на основе небольшого подмножества данных); кроме того, вам не приходится отбрасывать весь файл, поскольку выявляемое повреждение затрагивает только одну страницу.

Итоги

В этой главе мы поговорили о способе организации двоичных данных: о том, как производится сериализация примитивных типов данных, как они объединяются в ячейки, как ячейки объединяются в слоттированные страницы и как производится навигация по этим структурам.

Мы узнали, как следует поступать с типами данных переменного размера, такими как строки, последовательности байтов и массивы, и как создавать специальные ячейки, содержащие размер содержащихся в них значений.

Мы обсудили формат слоттированной страницы, который позволяет ссылаться на отдельные ячейки из-за пределов страницы, используя идентификатор ячейки, сохранять записи в порядке вставки и порядок ключей путем сортировки смещений ячеек.

Эти принципы можно использовать как для создания двоичных форматов для дисковых структур, так и для создания сетевых протоколов.

Дополнительная литература

Если вы хотите узнать больше о концепциях, упомянутых в этой главе, вы можете обратиться к следующим источникам:

Способы организации файлов

Folk, Michael J., Greg Riccardi, and Bill Zoellick. 1997. File Structures: An Object-Oriented Approach with C++ (3rd Ed.). Boston: Addison-Wesley Longman.

Giampaolo, Dominic. 1998. Practical File System Design with the Be File System (1st Ed.). San Francisco: Morgan Kaufmann.

Vitter, Jeffrey Scott. 2008. “Algorithms and data structures for external memory.” Foundations and Trends in Theoretical Computer Science 2, no. 4 (January): 305–474. https://doi.org/10.1561/0400000014.

ГЛАВА 4

Реализация B-деревьев

В предыдущей главе мы говорили об общих принципах построения двоичных форматов и узнали, как создавать ячейки, строить иерархии и соединять их со страницами с помощью указателей. Эти концепции применимы как к структурам хранения с обновлением на месте, так и к структурам, доступным только для добавления. В этой главе мы обсудим некоторые понятия, характерные для B-деревьев.

Разделы этой главы разбиты на три логические группы. Сначала мы поговорим о способах организации: о том, как создавать отношения между ключами и указателями и реализовывать заголовки и ссылки между страницами.

Затем мы поговорим о процессах, происходящих по мере спуска от корня к листу, а именно о том, как производить двоичный поиск, составлять навигационные цепочки и отслеживать родительские узлы на тот случай, если позднее будет нужно разделить или объединить узлы.

Наконец, мы поговорим о методах оптимизации (перебалансировке, добавлении только справа и массовой загрузке), процессах обслуживания и сборке мусора.

Заголовок страницы

Заголовок страницы содержит информацию о странице, которая может быть полезной при выполнении навигации, обслуживания и оптимизации. Обычно он содержит флаги, описывающие содержимое и макет страницы, количество ячеек на странице, нижнее и верхнее смещения, обозначающие пустое пространство (используется для добавления смещений ячеек и данных), и другие полезные метаданные.

Например, СУБД PostgreSQL (https://databass.dev/links/12) сохраняет в заголовке размер страницы и версию макета. В MySQL InnoDB (https://databass.dev/links/13) заголовок страницы содержит информацию о количестве записей в куче, уровне и некоторых других, специфических для реализации параметрах. В SQLite (https://databass.dev/links/14) заголовок страницы содержит количество ячеек и крайний правый указатель.

Магические числа

Помимо прочего, в заголовке файла или страницы часто размещается некоторое магическое число. Обычно это блок, который состоит из нескольких байт, содержащий постоянное число, которое может служить сигналом о том, что блок представляет собой страницу, либо определять вид или версию страницы.

Магические числа часто используются для валидации и проверки корректности [GIAMPAOLO98]. Вероятность того, что последовательность байтов с произвольно выбранным смещением будет точно совпадать с магическими числами, крайне мала. Если такое совпадение имеется, то используемое смещение, скорее всего, является корректным. Например, чтобы убедиться, что страница загружена и выровнена правильно, во время записи можно поместить в заголовок магическое число `50414745` (шестнадцатеричное представление слова PAGE (СТРАНИЦА)). Тогда при чтении страницы можно будет проверять ее корректность, сравнивая четыре байта из считанного заголовка с ожидаемой последовательностью байтов.

Ссылки на одноуровневые элементы

Некоторые реализации сохраняют прямые и обратные ссылки, указывающие на левую и правую страницы того же уровня. Эти ссылки позволяют находить соседние узлы, не поднимаясь обратно к родительскому узлу. Этот подход несколько усложняет операции разделения и слияния, так как вы должны дополнительно обновлять смещения одноуровневых элементов. Например, когда разделяется не крайний правый узел, обратный указатель правого одноуровневого узла (ранее указывавший на узел, который теперь разделен) необходимо перенаправить на вновь созданный узел.

На рис. 4.1 видно, что при отсутствии ссылок между одноуровневыми узлами для нахождения одноуровневого узла необходимо обратиться к родительскому узлу. Эта операция может привести к самому корню, так как непосредственный родитель может помочь в адресации только своих собственных потомков. Если же мы сохраним ссылки на одноуровневые элементы непосредственно в заголовке, то, чтобы найти предыдущий или следующий узел того же уровня, достаточно будет просто перейти по такой ссылке.

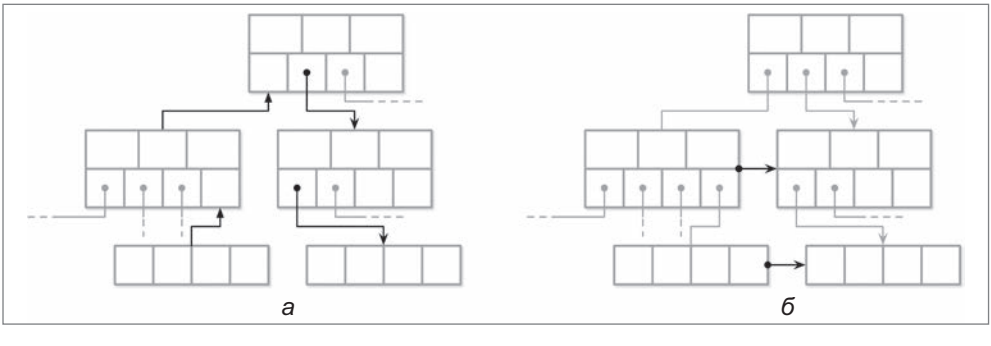

Рис. 4.1. Нахождение одноуровневого элемента путем перехода по ссылкам родителя (*а*) и перехода по ссылкам на одноуровневые узлы (*б*)

Одним из недостатков сохранения ссылок на одноуровневые узлы является то, что их необходимо обновлять при разделении и слиянии. Поскольку такое обновление производится в одноуровневом узле, а не в том узле, который подвергается разделе-

нию или слиянию, это может потребовать дополнительной блокировки. Как ссылки на одноуровневые узлы могут использоваться в параллельной реализации B-дерева, будет рассказано в подразделе «B^{link}-деревья» на с. 126.

Крайние правые указатели

Ключи-разделители B-дерева имеют строгие инварианты: они используются для разбиения дерева на поддеревья и навигации по ним, поэтому указателей на дочерние страницы всегда на единицу больше, чем ключей. Именно поэтому мы добавляли дополнительную единицу в подразделе «Подсчет ключей» на с. 56.

В подразделе «Ключи-разделители» на с. 54 мы уже рассмотрели инварианты ключей-разделителей. Во многих реализациях узлы выглядят примерно так, как показано на рис. 4.2: каждый ключ-разделитель имеет указатель на потомка, в то время как последний указатель сохраняется отдельно, так как он не связан с каким-либо ключом. Вы можете сравнить эту схему с рис. 2.10.

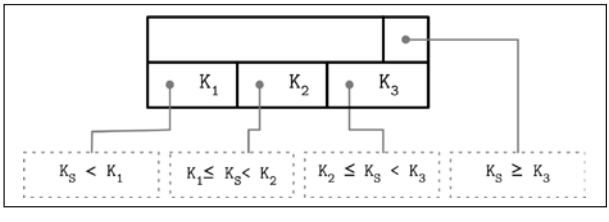

Рис. 4.2. Крайний правый указатель

Этот дополнительный указатель может быть сохранен в заголовке, как, например, это реализовано в СУБД SQLite (https://databass.dev/links/16).

В случае разделения крайнего правого дочернего элемента с добавлением новой ячейки к родительскому элементу вам потребуется переназначить указатель на крайнего правого потомка. Как показано на рис. 4.3, после такой операции разделения ячейка, добавленная к родительской ячейке (выделенная на рисунке серым цветом), содержит повышенный ключ и указывает на разделенный узел. Новым крайним правым указателем назначается указатель на новый узел. Аналогичный подход описан и реализован в SQLite[1].

Высокие ключи узла

Мы можем использовать несколько иной подход, сохраняя крайний правый указатель в ячейке вместе с *«высоким ключом»* узла. «Высокий ключ» — это максимально возможный ключ, который может присутствовать в поддереве под текущим узлом. Этот подход используется в PostgreSQL и называется «B^{link}-деревья» (о том, как этот подход сказывается на параллелизме, будет рассказано в подразделе «B^{link}-деревья» на с. 126).

[1] Этот алгоритм можно найти в функции `balance_deeper` в репозитории проекта.

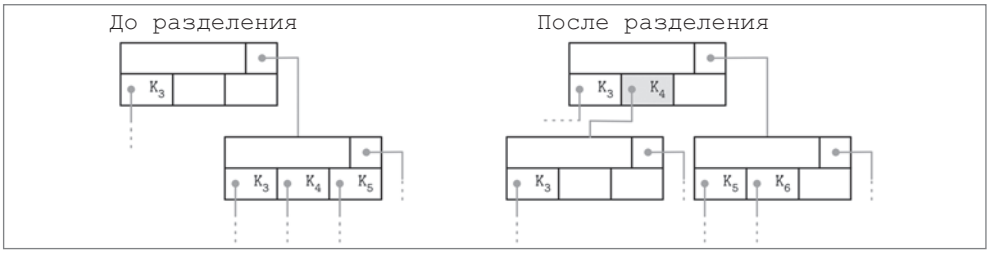

Рис. 4.3. Обновление крайнего правого указателя при разделении узла. Повышенный ключ выделен серым цветом

B-деревья содержат N ключей (обозначаемых как K_i) и N + 1 указателей (обозначаемых как P_i). В каждом поддереве ключи ограничены выражением $K_{i-1} \le K_s < K_i$. Ключ $K_0 = -\infty$ является неявным и не представлен в узле.

В случае B^{link}-деревьев каждый узел содержит дополнительный ключ K_{N+1}. Он задает верхнюю границу для ключей, которые могут храниться в поддереве, на которое указывает указатель P_N, и, следовательно, является верхней границей значений, которые могут храниться в текущем поддереве. Оба подхода показаны на рис. 4.4: на схеме (*а*) изображен узел без высокого ключа; на схеме (*б*) — узел с высоким ключом.

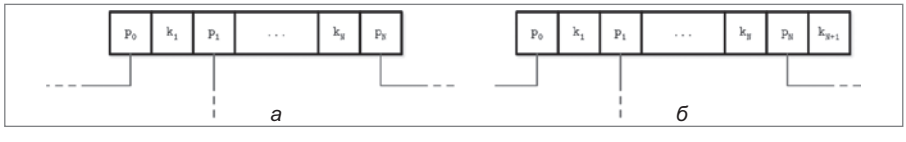

Рис. 4.4. B-деревья без высокого ключа (*а*) и с высоким ключом (*б*)

В данном случае указатели можно хранить попарно, а каждая ячейка может иметь соответствующий указатель, что упрощает работу с крайним правым указателем, поскольку требуется учитывать не так много граничных случаев.

На рис. 4.5 схематически показаны структура страниц при использовании обоих подходов и различия в способе разбиения пространства поиска: в первом случае оно доходит до $+\infty$, а во втором — до верхней границы K_3.

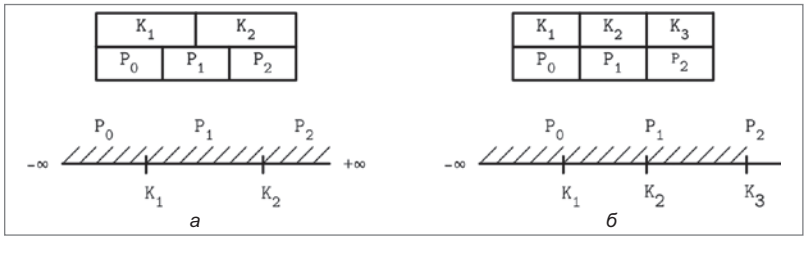

Рис. 4.5. Использование $+\infty$ в качестве виртуального ключа (*а*) и сохранение высокого ключа (*б*)

Страницы переполнения

Размер узла и степень ветвления дерева фиксированы и не меняются динамически. При этом также сложно подобрать их универсально оптимальным образом: если дерево будет содержать значения переменного размера и они будут достаточно большими, то на странице поместится только часть этих значений. Если эти значения будут очень малыми, это приведет к потере зарезервированного пространства.

Алгоритм B-дерева определяет, что каждый узел хранит определенное количество элементов. Поскольку размер значений может варьировать, мы можем столкнуться с ситуацией, когда, согласно алгоритму B-дерева, узел *еще не будет заполнен*, но на странице фиксированного размера, содержащей этот узел, *уже не будет свободного места*. Изменение размера страницы требует копирования уже записанных данных в новую область и часто нецелесообразно, однако нам все же нужно каким-то образом увеличить размер страницы или расширить ее.

Чтобы реализовать узлы переменного размера без копирования данных в новую непрерывную область, мы можем составлять узлы из нескольких привязанных к нему страниц. Например, допустим, что размер страницы по умолчанию равен 4 Кб и после вставки нескольких значений размер ее данных превысил 4 Кб. Вместо того чтобы использовать произвольные размеры, мы будем увеличивать размер узла с шагом в 4 Кб и соответственно выделим страницу расширения с размером 4 Кб и свяжем ее с исходной страницей. Такие привязанные расширения страниц называются *страницами переполнения*. Для ясности в рамках этого раздела мы будем называть исходную страницу *первичной страницей*.

Большинство реализаций B-деревьев позволяют сохранять лишь некоторое фиксированное количество байтов полезных данных непосредственно в узле B-дерева с *переливанием* (spilling) остальных данных на страницу переполнения. Это количество вычисляется путем деления размера узла на степень ветвления. Используя такой подход, мы никогда не столкнемся с ситуацией, когда на странице уже не будет свободного места, так как у нее всегда будет как минимум `max_payload_size` байтов. Более подробные сведения об использовании страниц переполнения в СУБД SQLite см. в репозитории исходного кода СУБД SQLite (https://databass.dev/links/16) и документации системы MySQL InnoDB (https://databass.dev/links/17).

Если размер вставляемых полезных данных превышает `max_payload_size`, то проверяется наличие привязанных к узлу страниц переполнения. Если страница переполнения уже существует и имеет достаточно свободного места, то оставшиеся байты записи выгружаются туда. В противном случае выделяется новая страница переполнения.

На рис. 4.6 показана первичная страница с записями, содержащими указатели на записи на странице переполнения, где располагаются непоместившиеся полезные данные.

При использовании страниц переполнения требуется вести некоторую дополнительную «бухгалтерию», так как они могут фрагментироваться, как и первичные стра-

ницы, и у нас должна быть возможность высвобождать это пространство для записи новых данных или удалять страницу переполнения, когда она уже будет не нужна.

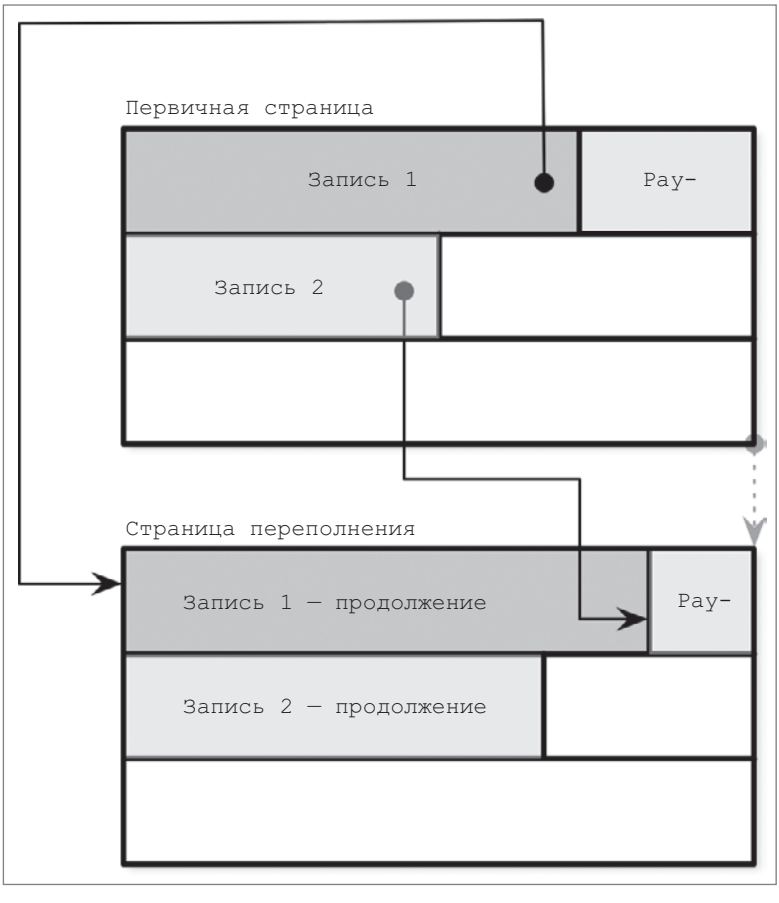

Рис. 4.6. Страницы переполнения

При выделении первой страницы переполнения ее идентификатор сохраняется в заголовке первичной страницы. Если одной страницы переполнения недостаточно, несколько страниц переполнения связываются вместе путем сохранения идентификатора каждой следующей страницы переполнения в заголовке предыдущей. В таком случае иногда приходится проходить через несколько страниц, чтобы найти «перелившуюся» часть полезных данных.

Поскольку множество ключей обычно имеет высокую мощность, их часто целесообразно сохранять в виде отдельных частей, так как большинство операций сравнения можно производить с помощью усеченной части ключа, размещенной на первичной странице.

Во время обработки запроса нам нужно восстановить полную запись и соединить данные из первичной страницы и страницы переполнения, которые относятся к этой записи. Однако это не имеет большого значения, так как такая операция производится нечасто. Если размер всех записей данных превышает размер первичной страницы, то стоит подумать об использовании специализированного хранилища, предназначенного для больших двоичных объектов.

Двоичный поиск

При обсуждении алгоритма поиска в B-дереве (см. подраздел «Алгоритм поиска в B-дереве» на с. 56) упоминалось, что поиск искомого ключа в узле производится с помощью алгоритма *двоичного поиска*. Двоичный поиск работает *только* для отсортированных данных. Если ключи не упорядочены, двоичный поиск к ним не применим. Вот почему важно сохранять порядок ключей и поддерживать инвариант сортировки.

Алгоритм двоичного поиска получает массив отсортированных элементов и искомый ключ и возвращает число. Если возвращаемое число является положительным, это означает, что искомый ключ был найден, а число указывает его позицию во входном массиве. Отрицательное значение указывает, что искомый ключ отсутствует в входном массиве, и дает нам *точку вставки*.

Точка вставки — это индекс первого элемента, *который больше* заданного ключа. Абсолютное значение этого числа представляет собой индекс, по которому можно вставить искомый ключ с сохранением порядка. Вставку можно выполнить путем смещения элементов на одну позицию, начиная с точки вставки, с тем чтобы освободить место для вставляемого элемента [SEDGEWICK11].

Зачастую поиск на высоких уровнях не дает точного совпадения, и тогда следует определить направление поиска, а затем найти первое значение, превышающее искомое, и перейти по соответствующей ссылке на дочерний элемент в поддереве.

Двоичный поиск с косвенными указателями

Ячейки на странице B-дерева зачастую сохраняются в порядке вставки, а логический порядок элементов сохраняется только для смещений ячеек. Мы начинаем двоичный поиск по ячейкам страницы со смещения ячейки, находящейся ровно посередине, находим ячейку по ее указателю, сравниваем значение в этой ячейке с искомым ключом, чтобы определить направление дальнейшего поиска (влево или вправо), и продолжить этот процесс рекурсивно, пока не будет найден искомый элемент или точка вставки, как показано на рис. 4.7.

Распространение операций разделения и слияния

Как мы уже говорили в предыдущих главах, операции разделения и слияния в B-дереве могут распространяться на более высокие уровни. Для этого у нас должна

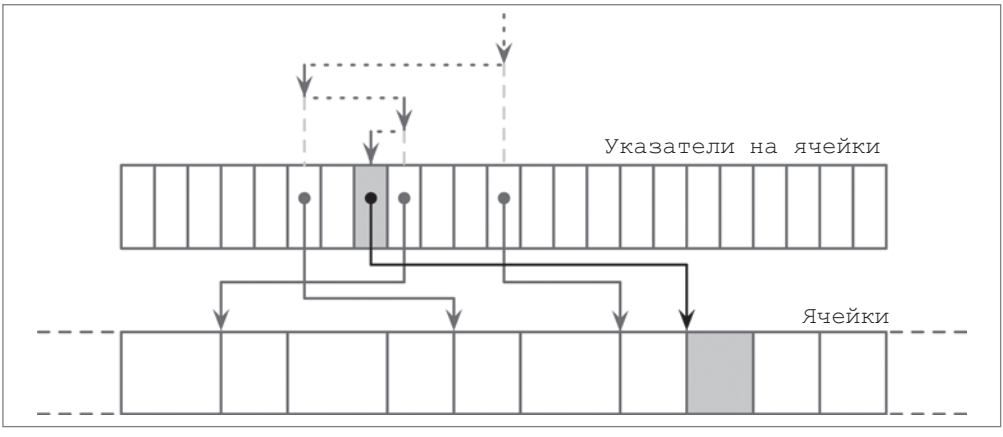

Рис. 4.7. Двоичный поиск с косвенными указателями. Искомый элемент выделен серым цветом. Пунктирными стрелками показан двоичный поиск по указателям ячеек. Сплошными линиями показаны операции доступа, следующие по указателям ячеек, необходимые для сравнения значения ячейки с искомым ключом

быть возможность пройти по цепочке обратно к корневому узлу от разделяемого листа или от объединяемой пары листьев.

Узлы B-дерева могут включать в себя указатели на родительские узлы. Страницы более низких уровней всегда загружаются в оперативную память, когда на них ссылаются с более высокого уровня, поэтому нет необходимости сохранять эту информацию на диске.

Подобно указателям на одноуровневые элементы (см. подраздел «Ссылки на одноуровневые элементы» на с. 80), указатели на родительские элементы должны обновляться при каждом изменении родительского элемента. Это делается во всех тех случаях, когда ключ-разделитель с идентификатором страницы передается от одного узла к другому: при разделении, слиянии или перебалансировке родительского узла.

Некоторые реализации (например, WiredTiger, https://databass.dev/links/20) используют указатели на родительские элементы для обхода листового уровня, чтобы исключить вероятность взаимной блокировки, которая может происходить при использовании указателей на одноуровневые элементы (см. [MILLER78] [LEHMAN81]). Вместо указателей на одноуровневые элементы для обхода листовых узлов в этом алгоритме используются указатели на родительские элементы (примерно так, как показано на рис. 4.1).

Для нахождения одноуровневого узла, мы можем сперва найти указатель на него в родительских узлах, а затем спуститься обратно рекурсивно на нижний уровень. Всякий раз, когда мы достигаем конца родительского узла после обхода всех одноуровневых элементов, связанных с этим родительским узлом, поиск рекурсивно продолжается вверх до достижения корня, а далее — вниз до листового уровня.

Навигационная цепочка

Вместо того чтобы сохранять и поддерживать указатели на родительские узлы, можно отслеживать, через какие узлы производился переход к целевому листовому узлу, и переходить по этой цепочке родительских узлов в обратном порядке в случае каскадного разделения, вызванного вставкой, или слияния, вызванного удалением.

При выполнении операций, способных привести к структурным изменениям в B-дереве (операций вставки или удаления), мы сначала проходим дерево от корня до листа, чтобы найти целевой узел и точку вставки. Поскольку часто нельзя сказать заранее, приведет ли операция к разделению или слиянию (по крайней мере, до тех пор, пока не будет найден целевой листовой узел), мы должны составлять *навигационную цепочку* (breadcrumbs).

Навигационная цепочка содержит ссылки на узлы, пройденные по мере спуска от корня, и используется для прохождения через эти узлы в обратном направлении при распространении операций разделения или слияния. Наиболее подходящей структурой данных для таких цепочек является стек. Например, в СУБД PostgreSQL навигационные цепочки сохраняются в стеке, который в рамках системы называют BTStack[1].

В случае разделения или слияния узла с помощью навигационной цепочки можно найти точки вставки для ключей, передаваемых в родительский узел, и при необходимости вернуться вверх по дереву, чтобы распространить структурные изменения на узлы более высокого уровня. Эта структура данных хранится в памяти.

На рис. 4.8 показан пример перехода от корня к листу с составлением навигационной цепочки, содержащей указатели на посещаемые узлы и индексы ячеек. В случае разделения целевого листового узла извлекается верхний элемент стека, для того чтобы найти непосредственного родителя узла. Если в родительском узле достаточно свободного места, то к нему добавляется новая ячейка, индекс для которой берется из навигационной цепочки (при условии, что этот индекс остается правильным). В противном случае родительский узел также разделяется. Этот процесс продолжается рекурсивно либо до полного исчерпания стека с достижением корневого узла, либо до достижения уровня, на котором уже не требуется разделение.

Перебалансировка

Некоторые реализации B-дерева стремятся как можно дольше откладывать операции разделения и слияния, чтобы снизить затраты на их выполнение за счет *перебалансировки* элементов внутри уровня или перемещения элементов из более занятых узлов в менее занятые. Это позволяет повысить степень заполненности узлов и часто приводит к уменьшению количества уровней в дереве при потенциально более высоких затратах на проведение перебалансировки.

[1] Подробнее об этом можно прочитать в репозитории проекта: https://databass.dev/links/21.

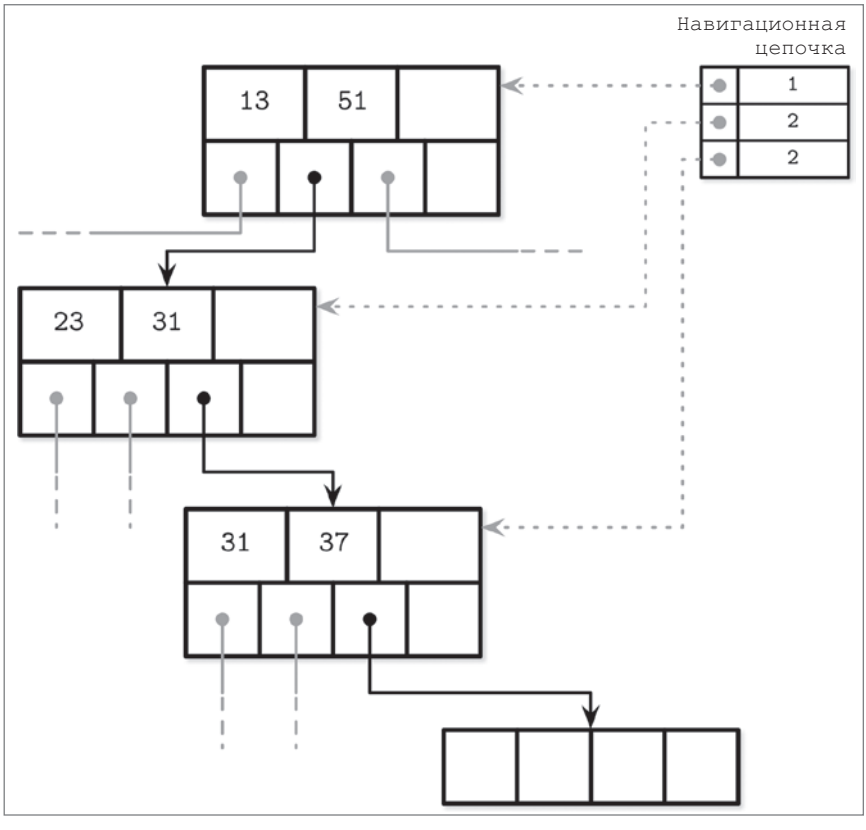

Рис. 4.8. Навигационная цепочка, составленная в процессе поиска, содержащая пройденные узлы и индексы ячеек. Пунктирными линиями показаны логические связи с посещаемыми узлами. Числа в таблице навигационной цепочки представляют собой индексы пройденных указателей на потомков

Балансировку загрузки можно выполнять при выполнении операций вставки и удаления [GRAEFE11]. Чтобы оптимизировать использование пространства, вместо разделения узла при переполнении можно перемещать несколько элементов в один из одноуровневых узлов, освобождая место для вставки. Аналогичным образом при выполнении удаления вместо слияния одноуровневых узлов можно перемещать несколько элементов из соседних узлов таким образом, чтобы узел был хотя бы наполовину заполнен.

*B**-деревья продолжают распределять данные между соседними узлами до тех пор, пока не будут заполнены оба одноуровневых узла [KNUTH98]. Затем вместо разделения одного узла на два узла, заполненных наполовину, алгоритм разбивает два узла на три узла, заполненных на две трети. При реализации СУБД SQLite применили именно этот вариант. Этот подход повышает среднюю степень заполненности, откладывая операции разделения, но требует дополнительных механизмов отслеживания и балансировки. Более высокая степень заполненности также повышает эффектив-

ность поиска за счет сокращения высоты дерева и уменьшения количества страниц, посещаемых по мере перехода к искомому листу.

На рис. 4.9 показан процесс распределения элементов между соседними узлами в случае, когда левый одноуровневый узел содержит больше элементов, чем правый. Элементы из более заполненного узла перемещаются в менее заполненный узел. Поскольку балансировка изменяет инвариант минимума и максимума одноуровневых узлов, для его поддержания мы должны обновить ключи и указатели в родительском узле.

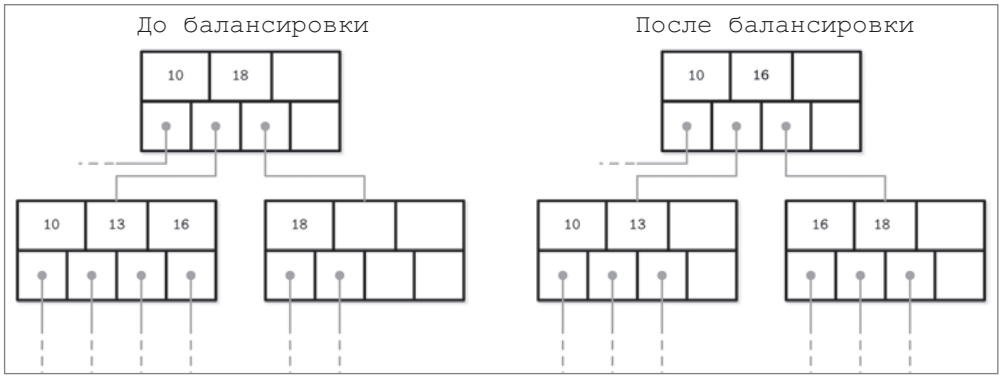

Рис. 4.9. Балансировка B-деревьев: распределение элементов между более заполненным и менее заполненным узлами

Балансировка нагрузки — это полезный метод, используемый во многих реализациях баз данных. Например, в SQLite реализован примерно такой же алгоритм балансировки одноуровневых элементов (https://databass.dev/links/23), какой был описан в этом разделе. Балансировка может привести к некоторому усложнению кода, но так как ее сценарии использования изолированы, ее можно реализовать как оптимизацию на более позднем этапе.

Добавление только справа

Многие СУБД используют в качестве ключей первичного индекса автоматически инкрементируемые и монотонно возрастающие значения. Такой подход позволяет производить оптимизацию, так как вставка всегда производится в конце определения индекса (в самом крайнем листе), и потому операции разделения, как правило, происходят в крайнем правом узле каждого уровня. Кроме того, так как ключи монотонно возрастают, при небольшой доле операций добавления по сравнению с операциями обновления и удаления нелистовые страницы будут менее фрагментированны, чем в случае использования неупорядоченных ключей.

В СУБД PostgreSQL такой подход называется *быстрым путем* (fastpath, https://databass.dev/links/24). Когда вставляемый ключ больше, чем первый ключ на крайней

правой странице и на крайней правой странице достаточно места для размещения вставляемой записи, эта запись вставляется в соответствующее место в кэшируемом крайнем правом листе и весь путь чтения может быть пропущен.

В СУБД SQLite есть аналогичная концепция под названием *быстрый баланс* (quick-balance, https://databass.dev/links/25). Когда запись вставляется в крайнем правом конце, а целевой узел заполнен (т. е. он становится самой большой записью в дереве после вставки), вместо перебалансировки или разделения узла выделяется новый крайний правый узел и добавляется его указатель на родительский элемент (подробнее о реализации балансировки в SQLite написано в разделе «Перебалансировка» на с. 87). Хотя вновь созданная страница остается почти пустой (а не пустой наполовину, как в случае разделения узла), вероятность того, что узел будет заполнен в ближайшее время, очень высока.

Массовая загрузка

Если у нас есть предварительно отсортированные данные и нам нужно произвести их массовую загрузку или перестроить дерево (например, с целью дефрагментации), мы можем развить идею добавления только справа еще дальше. Поскольку данные, необходимые для создания дерева, уже отсортированы, при выполнении массовой загрузки нам нужно лишь добавить элементы в крайней правой области дерева.

В таком случае мы можем полностью избежать разделения и слияния и составить дерево снизу вверх, записывая его уровень за уровнем или записывая узлы более высокого уровня при получении достаточного количества указателей на уже записанные узлы более низкого уровня.

Один из подходов к реализации массовой загрузки состоит в том, чтобы записывать предварительно отсортированные данные на листовом уровне постранично (вместо того, чтобы добавлять отдельные элементы). После записи листовой страницы ее первый ключ распространяется на родительский элемент и используется обычный алгоритм построения более высоких уровней B-дерева [RAMAKRISHNAN03]. Поскольку значения добавляются в отсортированном порядке, все разделения в таком случае происходят в крайнем правом узле.

Поскольку B-деревья всегда строятся начиная с нижнего (листового) уровня, весь листовой уровень может быть записан до того, как будут составлены узлы более высокого уровня. Это позволяет получить все указатели на потомков к моменту построения более высоких уровней. Основным преимуществом такого подхода является то, что нам не нужно выполнять какие-либо разделения или слияния на диске и, кроме того, во время построения в памяти нужно хранить лишь минимальную часть дерева (т. е. всех родителей листового узла, заполняемого в данный момент).

Таким же образом могут создаваться и неизменяемые B-деревья, но, в отличие от изменяемых B-деревьев, они не требуют дополнительного пространства для последующих модификаций, поскольку все операции над деревом носят окончательный характер. Все страницы можно заполнить полностью, что повышает степень заполненности и улучшает производительность.

Сжатие

Хранение необработанных, несжатых данных может привести к значительным накладным расходам, поэтому для экономии места многие базы данных предлагают различные способы сжатия данных. Здесь очевидным образом приходится находить компромисс между скоростью доступа и степенью сжатия: увеличение степени сжатия позволяет получать больше данных за один цикл доступа, но может потребовать больше циклов ОЗУ и процессора для сжатия и распаковки.

Сжатие может выполняться с использованием различной степени гранулярности данных. Хотя сжатие целых файлов может обеспечить более высокую степень сжатия, такой подход имеет ограниченное применение, поскольку при этом приходится повторно сжимать весь файл при каждом обновлении, и для больших наборов данных обычно лучше подходит сжатие с более высокой степенью гранулярности. Сжатие всего индексного файла не только нецелесообразно, но и трудно поддается реализации: для обращения к конкретной странице необходимо обратиться (чтобы найти сжатый сегмент) ко всему файлу (или его сегменту, содержащему метаданные сжатия), после чего распаковать его и сделать доступным.

Альтернативный подход состоит в том, чтобы сжимать данные постранично. Этот вариант хорошо нам подходит, так как алгоритмы, которые мы до сих пор обсуждали, используют страницы фиксированного размера. Страницы можно сжимать и распаковывать независимо друг от друга, что позволяет сочетать сжатие с загрузкой страниц и их выгрузкой на диск. Однако сжатая страница в таком случае может занимать только часть дискового блока, и поскольку передача обычно выполняется полными дисковыми блоками, в память часто требуется загружать лишние байты [RAY95]. На рис. 4.10 в случае (*а*) сжатая страница занимает меньше места, чем дисковый блок. При загрузке в память такой страницы также загружаются лишние байты, которые относятся к другой странице. Если страница занимает несколько дисковых блоков (см. случай (*б*)), нам приходится считывать дополнительный дисковый блок.

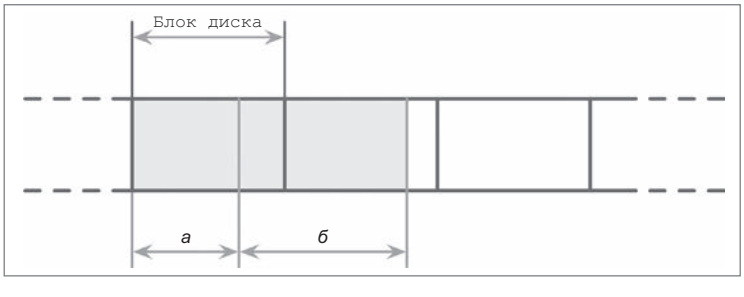

Рис. 4.10. Сжатие и заполнение блока

Еще один подход сводится к тому, чтобы сжимать только данные либо построчно (сжимая отдельные записи данных), либо поколоночно (сжимая отдельные столбцы). В таком случае управление страницами и сжатие не будут связаны друг с другом.

Большинство баз данных с открытым исходным кодом, рассмотренных при написании этой книги, позволяет подключать методы сжатия, используя доступные библиотеки, такие как Snappy (https://databass.dev/links/26), zLib (https://databass.dev/links/27), lz4 (https://databass.dev/links/28) и многие другие.

Поскольку алгоритмы сжатия дают различные результаты в зависимости от того, с каким набором данных вы работаете и какие показатели являются для вас приоритетными (степень сжатия, производительность или издержки памяти), в этой книге мы не будем проводить сравнение и вдаваться в детали различных реализаций. Существует множество обзоров, в которых оценивается использование различных алгоритмов сжатия при различном размере блоков (например, Squash Compression Benchmark, https://databass.dev/links/29). Основное внимание при этом обычно обращается на следующие четыре показателя: издержки памяти, производительность при сжатии, производительность при распаковке и степень сжатия. При выборе библиотеки сжатия важно учитывать эти показатели.

Очистка и обслуживание

До сих пор мы в основном говорили о пользовательских операциях с B-деревьями. Однако одновременно с выполнением запросов осуществляется и ряд других процессов, направленных на поддержание целостности хранилища, высвобождение пространства, снижение издержек и сохранение последовательности страниц. Выполнение этих операций в фоновом режиме позволяет нам сэкономить некоторое время и обойтись без очистки при выполнении операций вставки, обновления и удаления.

Описанная методика слоттированных страниц (см. раздел «Слоттированные страницы» на с. 70) требует некоторого обслуживания страниц для поддержания их в надлежащем состоянии. Например, многократное выполнение операций разделения и слияния во внутренних узлах или операций вставки, обновления и удаления на листовом уровне может привести к тому, что из-за фрагментации страница при наличии достаточного объема *логического* пространства не будет иметь достаточного объема *непрерывного* пространства. На рис. 4.11 показан пример такой ситуации: страница еще имеет некоторое доступное логическое пространство, но при этом фрагментирована и содержит две удаленные (мусорные) записи и некоторый объем свободного пространства между заголовком (указателями на ячейки) и ячейками.

Обход B-деревьев начинается с корневого уровня. Записи данных, которые можно получить, следуя по указателям вниз от корневого узла, являются *актуальными* (адресуемыми). Неадресуемые записи данных считаются *мусорными*: на эти записи нигде нет ссылок, и их нельзя считать или проанализировать, поэтому их содержимое можно считать заполненным нулями.

Это различие можно увидеть на рис. 4.11: ячейки, к которым еще ведут указатели, адресуемы, в отличие от удаленных или перезаписанных. Мусорные области часто не заполняются нулями из соображений производительности, так как в конечном итоге эти области все равно перезаписываются новыми данными.

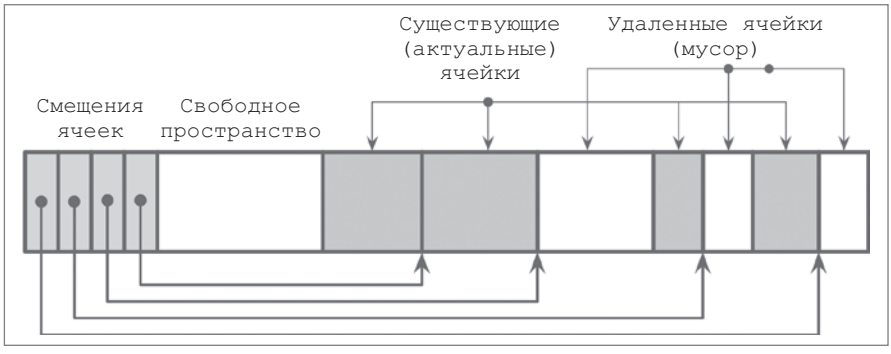

Рис. 4.11. Пример фрагментированной страницы

Фрагментация, вызываемая операциями обновления и удаления

Давайте посмотрим, при каких обстоятельствах на страницах появляются неадресуемые данные, из-за которых страницам требуется уплотнение. На листовом уровне в ходе операции удаления удаляются только смещения ячеек из заголовка, а сами ячейки не затрагиваются. После этого ячейка перестает быть *адресуемой* и ее содержимое не выводится в результатах запроса без необходимости в заполнении ее нулями или перемещении соседних ячеек.

При разделении страницы отсекаются только смещения, а поскольку остальная часть страницы неадресуема, ячейки, смещения которых были отсечены, становятся недоступными и подвергаются перезаписи при поступлении новых данных или собираются сборщиком мусора при запуске процесса очистки.

Некоторые базы данных полагаются на сборку мусора и оставляют удаленные и обновленные ячейки нетронутыми для обеспечения многоверсионного управления конкурентным доступом (см. подраздел «Многоверсионное управление конкурентным доступом» на с. 117). Ячейки остаются доступными для параллельно выполняемых транзакций до завершения обновления и могут быть собраны сборщиком мусора, когда к ним уже не будут обращаться никакие другие потоки. Некоторые базы данных поддерживают структуры, отслеживающие фантомные записи, собираемые сборщиком мусора после завершения всех транзакций, для которых эти записи были видимыми [WEIKUM01].

Поскольку в ходе операции удаления удаляются только смещения ячеек, без перемещения оставшихся ячеек в освободившееся пространство с физическим удалением удаленных ячеек, освобожденные байты могут оказаться разбросанными по странице. В таком случае мы говорим, что страница *фрагментирована* и нуждается в дефрагментации.

Чтобы произвести запись, мы должны иметь непрерывный блок свободных байтов, в котором может поместиться записываемая ячейка. Чтобы собрать освобожденные фрагменты вместе и исправить эту ситуацию, необходимо перезаписать страницу.

Операции вставки оставляют записи расположенными в порядке их вставки. Это не играет большой роли, однако расположение записей в порядке естественной сортировки часто упрощает реализацию упреждающей выборки кэша при последовательном считывании данных.

Операции обновления обычно производятся на листовом уровне: внутренние ключи страницы используются для управления навигацией и только определяют границы поддерева. Кроме того, операции обновления выполняются для каждого ключа по отдельности и, как правило, не приводят к структурным изменениям в дереве, за исключением случая создания страниц переполнения. Однако на листовом уровне операции обновления не изменяют порядок ячеек и стремятся исключить необходимость в перезаписи страницы. Это в конечном итоге может привести к сохранению нескольких версий ячейки, только одна из которых будет адресуемой.

Дефрагментация страниц

Процесс, обеспечивающий высвобождение пространства и перезапись страниц, называют *уплотнением, очисткой* или просто *обслуживанием*. Перезапись страниц может выполняться синхронно с записью, если на странице недостаточно свободного физического пространства (во избежание создания лишних страниц переполнения), но под уплотнением обычно понимается отдельный асинхронный процесс обхода страниц с выполнением сборки мусора и перезаписью их содержимого.

Этот процесс высвобождает пространство, занятое неактуальными ячейками, и перезаписывает ячейки в их логическом порядке. В ходе перезаписи страницы также могут быть перемещены в другую позицию внутри файла. Неиспользуемые страницы в памяти становятся доступными и возвращаются в кэш страниц. Идентификаторы вновь доступных страниц на диске добавляются в *список свободных страниц* (иногда называемый freelist[1]). Эту информацию необходимо хранить персистентно, чтобы не допустить потери или утечки свободного пространства в случае сбоя или перезапуска узлов.

Итоги

В этой главе мы обсудили некоторые специфические концепции дисковой реализации B-дерева:

Заголовок страницы

Какая информация обычно хранится в заголовке.

Крайние правые указатели

Мы выяснили, что они не сопряжены с ключами-разделителями, и узнали, как с ними следует обращаться.

[1] Например, СУБД SQLite поддерживает список неиспользуемых страниц следующим образом: *стволовые* страницы хранятся в связанном списке и содержат адреса освобожденных страниц.

Высокие ключи

Указывают, какой максимально допустимый ключ может быть сохранен в узле.

Страницы переполнения

Позволяют сохранить записи слишком большого и переменного размера при использовании страниц фиксированного размера.

После этого мы рассмотрели некоторые аспекты обхода деревьев от корня к листу:

- Как производить двоичный поиск с использованием косвенных указателей.
- Как отслеживать иерархии деревьев с помощью указателей на родительские элементы или навигационных цепочек.

Наконец, мы познакомились с некоторыми методами оптимизации и обслуживания.

Перебалансировка

Перемещает элементы между соседними узлами для уменьшения числа операций разделения и слияния.

Добавление только справа

Добавляет новую крайнюю правую ячейку (вместо того, чтобы разбивать ячейку) из расчета на то, что она быстро заполнится.

Массовая загрузка

Метод эффективного построения B-деревьев с нуля из отсортированных данных.

Сборка мусора

Процесс, который переписывает страницы, сортирует ячейки в порядке ключей и высвобождает пространство, занятое неадресуемыми ячейками.

Эти концепции призваны заполнить разрыв между базовым алгоритмом B-дерева и реально используемыми реализациями и помочь вам лучше разобраться в том, как работают системы хранения данных на основе B-деревьев.

Дополнительная литература

Если вы хотите узнать больше о концепциях, упомянутых в этой главе, вы можете обратиться к следующим источникам:

Дисковые B-деревья

Graefe, Goetz. 2011. “Modern B-Tree Techniques.” Foundations and Trends in Databases 3, no. 4 (April): 203–402. https://doi.org/10.1561/1900000028.

Healey, Christopher G. 2016. Disk-Based Algorithms for Big Data (1st Ed.). Boca Raton: CRC Press.

ГЛАВА 5

Обработка транзакций и восстановление

В этой книге мы начали изучать концепции СУБД «снизу вверх» и уже рассмотрели структуры хранения данных. Пора перейти к изучению компонентов более высокого уровня, отвечающих за управление буфером, управление блокировками и восстановление. Понимание их внутреннего устройства является необходимым условием для понимания транзакций баз данных.

Транзакция — это неделимая логическая единица работы в СУБД, позволяющая представить несколько операций в виде единого шага. Операции, выполняемые транзакциями, включают в себя чтение и запись значений базы данных. Транзакция базы данных должна обладать свойствами *атомарности*, *согласованности*, *изолированности* и *долговечности* (atomicity, consistency, isolation, durability, ACID) [HAERDER83].

Атомарность

Стадии транзакции *неотделимы* друг от друга, т. е. либо все стадии, связанные с транзакцией, выполняются успешно, либо не выполняется ни одна из них. Иными словами, транзакции не должны выполняться частично. Каждая транзакция может быть либо *зафиксирована* (committed) (в случае чего становятся видимыми все изменения, вносимые операциями записи, выполняемыми в рамках транзакции), либо *прервана* (в случае чего производится откат всех побочных эффектов транзакции, которые еще не стали видимыми). Фиксация — это заключительная операция. После прерывания может быть предпринята повторная попытка выполнения транзакции.

Согласованность

Согласованность — свойство, зависящее от приложения; транзакция должна переводить базу данных из одного допустимого состояния в другое допустимое состояние, сохраняя все инварианты базы данных (включая различные ограничения, ссылочную целостность и т. д.). Согласованность является наиболее слабо определенным свойством, возможно, потому, что это единственное свойство, которое контролируется пользователем, а не только самой базой данных.

Изолированность

Несколько параллельно выполняемых транзакций должны выполняться без взаимного влияния друг на друга, как если бы в это время не выполнялись никакие другие транзакции.

Изолированность определяет, *когда* должны становиться видимыми изменения в состоянии базы данных и какие изменения должны быть видимыми для параллельных транзакций. Ради повышения производительности многие базы данных используют более слабые уровни изолированности по сравнению с данным определением. В зависимости от методов и подходов, используемых для управления параллелизмом, изменения, вносимые транзакцией, могут быть видны или не видны другим параллельным транзакциям (см. подраздел «Уровни изолированности» на с. 114).

Долговечность

После фиксации транзакции все изменения в состоянии базы данных должны персистентно храниться на диске и сохраняться в случае перебоев в электроснабжении и сбоев и отказов системы.

Для реализации транзакций в СУБД в дополнение к структуре хранения, которая организует и сохраняет данные на диске, требуется совместная работа нескольких компонентов. Внутри узла *диспетчер транзакций* координирует, планирует и отслеживает транзакции и их отдельные стадии.

Диспетчер блокировок контролирует доступ к этим ресурсам и не допускает выполнения параллельных обращений, способных нарушить целостность данных. При каждом запросе блокировки диспетчер проверяет, не предоставлена ли она уже какой-либо другой транзакции в совмещаемом или исключительном режиме, и предоставляет блокировку при наличии требуемого уровня доступа. Поскольку исключительная блокировка в любой заданный момент может быть предоставлена только одной транзакции, для выполнения других запрашивающих ее транзакций необходимо дождаться освобождения блокировки либо прервать их выполнение и предпринять повторную попытку позже. После освобождения блокировки или завершения соответствующей транзакции диспетчер блокировок уведомляет одну из ожидающих транзакций, позволяя ей захватить блокировку и продолжить работу.

Кэш страниц выступает в роли посредника между персистентным носителем (диском) и остальной частью подсистемы хранения. Он аккумулирует изменения состояния в оперативной памяти и служит в качестве кэша для страниц, которые еще не были синхронизированы с персистентным носителем. Все изменения состояния базы данных сначала применяются к кэшированным страницам.

Диспетчер журналов хранит историю операций (записи журнала), примененных к кэшированным страницам, но еще не синхронизированных с персистентным носителем, чтобы изменения не были потеряны в случае сбоя. То есть журнал используется для повторного применения этих операций и восстановления кэшированного состояния во время запуска. Записи журнала также можно использовать для отмены изменений, внесенных прерванными транзакциями.

Распределенные (многосоставные) транзакции требуют дополнительной координации и удаленного выполнения. Протоколы распределенных транзакций будут рассмотрены в главе 13.

Организация буферизации данных

Большинство баз данных использует двухуровневую схему организации памяти, где одним уровнем является более медленный персистентный носитель (диск), а другим — более быстрая оперативная память (ОЗУ). Чтобы снизить количество обращений к персистентному носителю, страницы *кэшируются* в ОЗУ. При повторном запрашивании страницы уровнем хранения возвращается ее кэшированная копия.

Имеющиеся в памяти кэшированные страницы можно повторно использовать при условии, что другие процессы не будут вносить изменения в данные на диске. Такой подход иногда называют использованием *виртуального диска* [BAYER72]. При чтении виртуального диска данные считываются из физического хранилища только в том случае, если в памяти еще нет копии считываемой страницы. Более распространенное название этой концепции — *кэш страниц*, или *буферный пул*. Кэш страниц отвечает за кэширование страниц, считываемых с диска в память. В случае сбоя СУБД или некорректного завершения работы кэшированное содержимое теряется.

Поскольку термин *кэш страниц* лучше отражает назначение этой структуры, в этой книге по умолчанию используется это название. Термин *буферный пул* звучит так, как будто его основная цель — объединение и повторное использование *пустых* буферов без совместного использования их содержимого, что может быть полезной составляющей кэширования страниц и даже может использоваться в виде отдельного компонента, но не отражает целиком назначение этого механизма.

Кэширование страниц не ограничивается рамками баз данных. В операционных системах также есть понятие кэша страниц. Операционные системы используют *незанятые* сегменты памяти, чтобы прозрачно кэшировать содержимое диска для повышения производительности системных вызовов ввода-вывода.

Процесс загрузки некэшированных страниц в память называют *подкачкой страниц*. Если в кэшированную страницу вносятся изменения, она считается *грязной* до тех пор, пока эти изменения не будут *выгружены* (или *сброшены*) на диск.

Поскольку область памяти, предназначенная для хранения кэшированных страниц, обычно гораздо меньше, чем весь набор данных, кэш страниц в конечном итоге заполняется, после чего для загрузки новой страницы уже требуется *вытеснить* из кэша одну из кэшированных страниц.

На рис. 5.1 можно увидеть связь между логическим представлением страниц B-дерева, их кэшированными версиями и страницами на диске. Кэш страниц загружает страницы в свободные слоты в произвольном порядке, поэтому нет прямого соответствия между порядком следования страниц на диске и в памяти.

Основные функции кэша страниц можно свести к следующему:

- Сохраняет содержимое кэшированных страниц в памяти.
- Позволяет буферизировать изменения, вносимые в дисковые версии страниц и вносить их в кэшированные версии.

- Если запрашиваемая страница отсутствует в памяти и для нее имеется достаточно свободного места, она загружается в кэш страниц, а затем возвращается ее кэшированная версия.
- Если запрашиваемая страница уже находится в кэше, то просто возвращается ее кэшированная версия.
- Если для новой страницы недостаточно свободного места, некоторая другая страница *вытесняется* из кэша, а ее содержимое *выгружается* на диск.

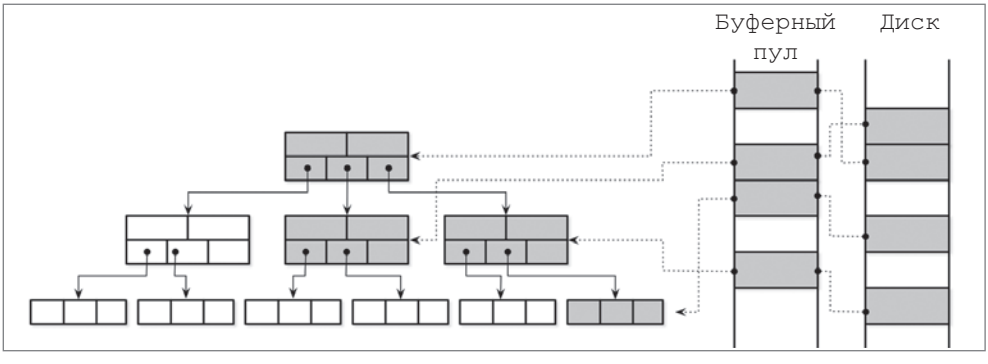

Рис. 5.1. Кэш страниц

ОБХОД СТРАНИЧНОГО КЭША ЯДРА

Многие СУБД открывают файлы с помощью флага `O_DIRECT`. Этот флаг позволяет системным вызовам ввода-вывода обходить кэш страниц ядра, обращаясь к диску напрямую и используя управление буфером конкретной базы данных. К этому порой неодобрительно относятся разработчики операционных систем.

Линус Торвальдс не одобряет (https://databass.dev/links/32) использование флага `O_DIRECT` по той причине, что такой подход не асинхронен и не предусматривает упреждающее чтение или другие средства для информирования ядра о схемах доступа. Однако до тех пор, пока операционные системы не начнут предлагать лучшие механизмы, флаг `O_DIRECT` по-прежнему будет полезен.

Мы можем получить некоторый контроль над процессом вытеснения страниц из кэша операционной системы, используя функцию `fadvise` (https://databass.dev/links/33), но такой подход позволяет лишь попросить ядро учесть наше мнение и не гарантирует, что оно так и сделает. Чтобы отказаться от системных вызовов при вводе-выводе, мы можем использовать отображение в память, но тогда мы теряем контроль над кэшированием.

Семантика кэширования

Все изменения, внесенные в буферы, сохраняются в памяти, пока они не будут записаны обратно на диск. Поскольку ни один другой процесс не может вносить

изменения в резервный файл, эта синхронизации производится в одном направлении — из памяти на диск, но не наоборот. Кэш страниц обеспечивает базе данных бóльшую степень контроля над управлением памятью и доступом к диску. Его можно рассматривать как специфичный для приложения эквивалент кэша страниц ядра: он напрямую обращается к устройству блочного ввода-вывода, реализует аналогичную функциональность и служит той же цели. Он абстрагирует доступ к диску и отделяет логические операции записи от физических.

Кэширование страниц позволяет хранить часть дерева в памяти без внесения дополнительных изменений в алгоритм и реализации объектов в памяти. Все, что нам нужно сделать, — это заменить операции доступа к диску вызовами кэша страниц.

Когда подсистема хранения обращается к странице (т. е. запрашивает ее), мы сначала проверяем, не кэшировано ли уже ее содержимое, в случае чего возвращается содержимое кэшированной страницы. Если содержимое страницы еще не кэшировано, кэш преобразует логический адрес страницы или номер страницы в свой физический адрес, загружает ее содержимое в память и возвращает кэшированную версию подсистеме хранения. При этом подсистема хранения получает ссылку на буфер с содержимым кэшированной страницы, которую необходимо вернуть обратно кэшу страницы или удалить после завершения операции. Также можно дать кэшу страниц указание не вытеснять страницу из кэша, *закрепив* ее.

После изменения страницы (например, при добавлении ячейки) она помечается как «грязная». Если на странице установлен «грязный» флаг, это говорит о том, что ее содержимое не синхронизировано с диском и должно быть выгружено для обеспечения долговечности данных.

Вытеснение из кэша

Нужно стремиться к тому, чтобы кэш всегда был заполненным: это позволит производить больше операций чтения без обращения к персистентному носителю и буферизировать больше операций записи, выполняемых в рамках одной страницы. Однако емкость кэша страниц ограничена, и рано или поздно для доставки нового содержимого потребуется вытеснять старые страницы. Если содержимое страницы синхронизировано с диском (т. е. уже выгружено или никогда не изменялось), а страница не закреплена и на нее нет ссылок, то ее можно сразу же вытеснить. «Грязные» страницы перед вытеснением из кэша необходимо *выгрузить на диск*. Страницы, на которые есть ссылки, не должны вытесняться, пока их использует какой-то другой поток.

Так как запуск выгрузки при каждом вытеснении может плохо сказаться на производительности, некоторые базы данных используют отдельный фоновый процесс, который циклически обходит те «грязные» страницы, которые могут подвергаться вытеснению, и обновляет их дисковые версии. Например, именно это делает фоновый модуль выгрузки (https://databass.dev/links/34) в СУБД PostgreSQL.

Также важно не забывать о необходимости обеспечения *долговечности*: если в базе данных произойдет сбой, будут потеряны все невыгруженные данные. Чтобы обес-

печить персистентное сохранение всех изменений, операции выгрузки координируются процессом создания *контрольных точек*. Этот процесс управляет журналом упреждающей записи и кэшем страниц и обеспечивает их согласованную работу. Из журнала упреждающей записи могут удаляться только те записи, которые относятся к операциям, выполненным на уже выгруженных кэшированных страницах. «Грязные» страницы не подлежат вытеснению из кэша до завершения этого процесса.

Это означает, что всегда приходится находить компромисс между несколькими целями:

- Откладывать выгрузку, чтобы уменьшить количество обращений к диску.
- Выполнять упреждающую выгрузку страниц для обеспечения быстрого вытеснения.
- Выбирать страницы для вытеснения и выгрузки в оптимальном порядке.
- Поддерживать размер кэша в пределах выделенной для него области памяти.
- Не допускать потерь данных до их персистентного сохранения на основном носителе.

Далее мы рассмотрим ряд методов, позволяющих улучшить первые три характеристики без выхода за пределы, указанные в последних двух пунктах.

Блокировка страниц в кэше

Выполнять дисковый ввод-вывод при выполнении каждой операции чтения или записи нецелесообразно, поскольку последовательные операции чтения могут запрашивать одну и ту же страницу, а последовательные операции записи могут модифицировать одну и ту же страницу. Поскольку B-дерево сужается кверху, узлы более высокого уровня (расположенные ближе к корню) затрагиваются в ходе большинства операций чтения. Операции разделения и слияния также в конечном счете распространяются до узлов более высокого уровня. Это означает, что всегда имеется как минимум часть дерева, кэширование которой дает значительную пользу.

Мы можем «заблокировать» те страницы, которые с большой вероятностью будут использованы в ближайшее время. Блокировка страниц в кэше называется *закреплением*. Закрепленные страницы хранятся в памяти дольше, что позволяет сократить число обращений к диску и повысить производительность [GRAEFE11].

Поскольку каждый более низкий уровень B-дерева имеет экспоненциально больше узлов по сравнению с предыдущим, в силу чего узлы более высоких уровней составляют лишь небольшую часть дерева, мы можем держать эту часть дерева в памяти постоянно, подкачивая остальные части по мере необходимости. Это означает, что при выполнении запроса не потребуется делать h обращений к диску (где h — высота дерева, как упоминалось в подразделе «Сложность поиска в B-дереве» на с. 55); достаточно будет считать с диска только некэшированные страницы нижних уровней.

Операции, выполняемые с поддеревом, могут порождать противоречащие друг другу структурные изменения — так, после нескольких операций удаления, порождающих

слияние, могут быть выполнены операции записи, порождающие разбиение, или наоборот. То же самое справедливо и для структурных изменений, распространяющихся из разных поддеревьев (которые происходят близко друг к другу по времени в разных частях дерева и распространяются вверх). Эти операции можно буферизовать, применяя изменения только в памяти, что позволит уменьшить количество операций записи на диск и снизить затраты на операции, так как вместо нескольких операций записи будет выполняться только одна.

ПРЕДВАРИТЕЛЬНАЯ ВЫБОРКА И МГНОВЕННОЕ ВЫТЕСНЕНИЕ

Кэш страниц также обеспечивает подсистеме хранения точный контроль над предварительной выборкой и вытеснением из кэша. Кэшу страниц можно дать указание загружать страницы заранее, еще до обращения к ним. Например, при обходе листовых узлов во время сканирования диапазона можно предварительно загружать следующие листы. Аналогичным образом в случае загрузки страницы процессом обслуживания ее можно вытеснять из кэша сразу после завершения этого процесса, так как она вряд ли пригодится для текущих запросов. Некоторые базы данных, например PostgreSQL (https://databass.dev/links/35), используют циклический буфер (т. е. производят замену страниц по принципу FIFO) при сканировании больших последовательных областей данных.

Замещение страниц

При достижении предельной емкости кэша для загрузки новых страниц требуется вытеснять старые страницы. Однако следует вытеснять именно те страницы, которые с наименьшей вероятностью понадобятся в ближайшее время, иначе нам придется загружать вытесняемые страницы снова и снова, хотя мы могли бы просто держать их в памяти. Для оптимизации процесса нам нужно каким-то образом оценивать вероятность последующего доступа к странице.

Страницы необходимо вытеснять в соответствии с *политикой вытеснения* (также иногда называемой *политикой замещения страниц*). Согласно этой политике, для вытеснения выбираются страницы, которые с наименьшей вероятностью понадобятся в ближайшее время. После вытеснения страницы на ее место в кэше можно загрузить новую страницу.

Чтобы реализация кэша страниц могла работать эффективным образом, она должна использовать эффективный алгоритм замещения страниц. Идеальная стратегия сводится к тому, чтобы каким-то образом предсказать точную последовательность использования страниц и вытеснить только те страницы, которые не будут использоваться максимально долго. Поскольку последовательность запросов редко соответствует какому-либо конкретному паттерну или способу распределения, предсказать ее точно практически невозможно, однако мы можем уменьшить количество вытеснений, используя правильную стратегию замещения страниц.

На первый взгляд может показаться, что уменьшить количество вытеснений можно просто путем увеличения кэша. Однако на самом деле это не так. Один из примеров,

демонстрирующих эту дилемму, называется *аномалией Белади* [BEDALY69]. Он показывает, что увеличение числа страниц может увеличить число вытеснений, если используется неоптимальный алгоритм замещения страниц. Когда страницы, которые могут потребоваться в ближайшее время, вытесняются, а затем загружаются снова, возникает конкуренция страниц за место в кэше. Поэтому нужно как следует продумать используемый алгоритм, чтобы он улучшал, а не ухудшал ситуацию.

FIFO и LRU

Простейшей стратегией замещения страниц является принцип «первым пришел — первым ушёл» (first in — first out, FIFO). Алгоритм FIFO поддерживает очередь идентификаторов страниц, расположенных в порядке их вставки, добавляя новые страницы в хвост очереди. Когда кэш страниц полностью заполняется, извлекается элемент из вершины очереди, соответствующий странице, которая была загружена раньше других. Поскольку данный алгоритм учитывает только событие загрузки в кэш и не берет в расчет последующие обращения к странице, его нецелесообразно использовать в большинстве реальных систем. Например, корневые и самые верхние страницы загружаются первыми и, согласно этому алгоритму, являются первыми кандидатами на вытеснение, хотя, согласно свойствам структуры дерева, эти страницы потребуется загрузить снова если не сразу, то в самое ближайшее время.

Естественным развитием алгоритма FIFO является алгоритм «наиболее давно использовавшийся» (least-recently used, LRU) [TANENBAUM14]. Данный алгоритм также поддерживает очередь кандидатов на вытеснение, расположенных в порядке вставки, но при этом позволяет снова поместить страницу в хвост очереди при повторном обращении к ней, как если бы это была первоначальная загрузка страницы в кэш. Однако обновление ссылок и повторное связывание узлов при каждом обращении могут оказаться затратными в параллельной среде.

Существуют и другие стратегии вытеснения из кэша на основе LRU. Например, алгоритм 2Q (LRU с двумя очередями) поддерживает две очереди и помещает страницы в первую очередь при первоначальном обращении и во вторую «горячую» очередь — при последующих обращениях, что позволяет различать недавно и часто запрашиваемые страницы [JONSON94]. Алгоритм LRU-K определяет, ссылки на какие страницы используются наиболее часто, отслеживая последние K обращений и используя эту информацию для оценки количества обращений к каждой странице [ONEIL93].

CLOCK

В некоторых ситуациях эффективность может быть важнее точности. В качестве альтернативы для алгоритма LRU часто используются различные варианты алгоритма CLOCK. Они отличаются компактностью и дружественностью к кэшу и могут использоваться в параллельных средах [SOUNDARARARJAN06]. Один из вариантов этого алгоритма (https://databass.dev/links/36), например, используется в Linux.

Алгоритм CLOCK хранит ссылки на страницы и связанные с ними биты доступа в циклическом буфере. Некоторые варианты используют счетчики (https://databass.

dev/links/37) вместо битов для учета частоты. При каждом обращении к странице ее бит доступа устанавливается равным 1. Алгоритм обходит циклический буфер, проверяя биты доступа:

- Если бит доступа равен 1 и страница не используется в данный момент, этот бит устанавливается в 0 и проверяется следующая страница.
- Если бит доступа уже равен 0, страница становится кандидатом и подлежит вытеснению.
- Если страница в данный момент используется, ее бит доступа остается неизменным. Подразумевается, что у страницы, к которой осуществляется доступ, бит доступа не может быть равным 0, поэтому она не может быть подвергнута вытеснению. Это снижает вероятность вытеснения страниц со ссылками.

На рис. 5.2 показан циклический буфер с битами доступа.

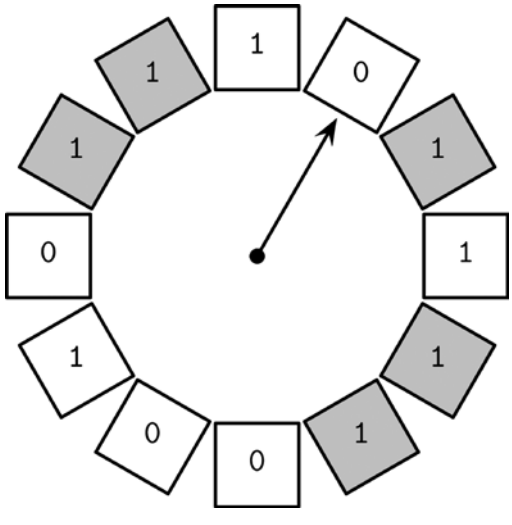

Рис. 5.2. Алгоритм CLOCK. Счетчики используемых страниц выделены серым цветом. Счетчики неиспользуемых страниц выделены белым цветом. Стрелка указывает на элемент, который будет проверяться следующим

Преимущество циклического буфера состоит в том, что и указатель стрелки часов, и содержимое можно изменять с помощью операций сравнения с обменом, без применения дополнительных механизмов блокировки. Этот алгоритм прост в понимании и реализации и часто используется как в учебниках [TANENBAUM14], так и в реальных системах.

LRU не всегда является наилучшей стратегией замещения. В некоторых СУБД целесообразнее использовать в качестве прогностического фактора *частоту использования*, а не *недавность*. В случае СУБД с большой нагрузкой недавность часто

является не очень показательной характеристикой, поскольку она отражает лишь последовательность доступа к элементам.

LFU

Чтобы улучшить ситуацию, мы можем отслеживать не события загрузки в кэш, а события обращения по ссылке к странице. Один из способов это сделать сводится к отслеживанию наименее часто используемых (least-frequently used, LFU) страниц.

Именно это и делает политика вытеснения на основе частоты TinyLFU [EINZIGER17]: вместо вытеснения страниц в зависимости от *недавности загрузки* она вытесняет их в зависимости от *частоты использования*. Она реализована в популярной Java-библиотеке под названием Caffeine (https://databass.dev/links/38).

TinyLFU использует частотную гистограмму [CORMODE11] для поддержания компактной истории доступа к кэшу, так как сохранение всей истории может быть непомерно затратным с практической точки зрения.

Элементы могут находиться в одной из трех очередей:

- *Прием*: очередь вновь добавленных элементов, реализованная с использованием политики LRU.
- *Испытание*: очередь элементов, которые, скорее всего, будут вытеснены.
- *Защита*: очередь элементов, которые должны оставаться в очереди в течение более длительного времени.

Вместо того чтобы каждый раз выбирать, какие элементы следует вытеснить, согласно этому подходу выбираются элементы, которые следует повысить для сохранения. В очередь испытания могут быть перемещены только те элементы, частота использования которых выше, чем у элемента, который будет вытеснен в результате их повышения. При последующих обращениях элементы могут быть перемещены из очереди испытания в очередь защиты. Если очередь защиты заполнена, один из ее элементов может быть снова помещен в очередь испытания. Более часто используемые элементы имеют более высокие шансы на сохранение, а менее часто используемые — более высокие шансы на вытеснение.

На рис. 5.3 показаны логические связи между очередями приема, испытания и защиты, частотным фильтром и вытеснением.

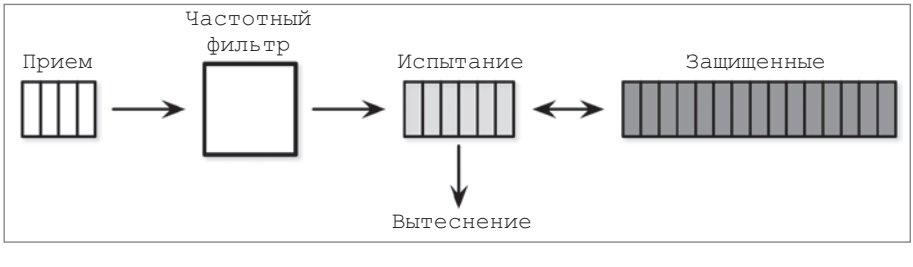

Рис. 5.3. TinyLFU: очереди приема, испытания и защиты

Существует и множество других алгоритмов, которые можно использовать для оптимального вытеснения из кэша. Поскольку выбор стратегии замещения страниц оказывает значительное влияние на время задержки и количество выполняемых операций ввода-вывода, к нему следует относиться с должным вниманием.

Восстановление

СУБД являются надстройкой поверх нескольких аппаратных и программных уровней, у которых могут быть свои проблемы со стабильностью и надежностью. При этом и сами СУБД, и лежащие в их основе программные и аппаратные компоненты могут иногда выходить из строя. Разработчики баз данных должны рассмотреть эти сценарии сбоев и проследить за тем, чтобы запись данных всегда происходила ожидаемым образом.

Журнал упреждающей записи (write-ahead log, WAL), также иногда называемый *журналом фиксации*, — это вспомогательная дисковая структура с доступом только для добавления, которая используется для восстановления после сбоев и восстановления транзакций. Кэш страниц позволяет буферизировать в памяти изменения содержимого страницы. До тех пор пока кэшированное содержимое не будет выгружено обратно на диск, единственная копия, сохраняющая историю операций на диске, располагается в журнале упреждающей записи. Журналы упреждающей записи, доступные только для добавления, используются во многих СУБД; например в PostgreSQL (https://databass.dev/links/39) и MySQL (https://databass.dev/links/40).

Основные функции журнала упреждающей записи можно свести к следующему:

- Позволяет кэшу страниц буферизировать обновления размещенных на диске страниц с обеспечением долговечности в общем контексте СУБД.
- Производит персистентное сохранение всех операций на диске до тех пор, пока кэшированные копии страниц, затронутых этими операциями, не будут синхронизированы с копиями на диске. Каждая операция, которая изменяет состояние базы данных, должна быть записана в журнале на диске до изменения содержимого соответствующих страниц.
- Позволяет в случае сбоя восстановить из журнала операций потерянные изменения, которые были произведены в памяти.

В дополнение к этим функциям журнал упреждающей записи играет важную роль в обработке транзакций. Его роль трудно переоценить, так как он гарантирует, что данные попадут на персистентный носитель и будут доступны даже после сбоя, так как незафиксированные данные воспроизводятся из журнала с полным восстановлением состояния базы данных до сбоя. В этом разделе мы часто будем упоминать алгоритм ARIES. ARIES (Algorithm for Recovery and Isolation Exploiting Semantics, алгоритм восстановления и изоляции на основе семантики) — современный алгоритм восстановления, который широко используется и часто упоминается в литературе [MOHAN92].

POSTGRESQL И FSYNC()

СУБД PostgreSQL использует контрольные точки, чтобы гарантировать обновление индексных файлов и файлов данных всей информацией вплоть до конкретной записи в файле журнала. В рамках процесса контрольных точек периодически производится выгрузка всех «грязных» (измененных) страниц. Синхронизация содержимого «грязных» страниц с диском выполняется с помощью вызова ядра `fsync()`, который должен синхронизировать «грязные» страницы с диском и сбросить «грязный» флаг на страницах ядра. Как и следовало ожидать, `fsync` возвращает ошибку, если выгрузить страницы на диск не удается.

В Linux и некоторых других операционных системах `fsync` сбрасывает «грязный» флаг даже для неудачно выгруженных страниц при ошибке ввода-вывода. Кроме того, сведения об ошибках указываются только в тех дескрипторах файлов, которые были открыты в момент сбоя, поэтому `fsync` не будет возвращать ошибки, возникшие до открытия дескриптора, для которого он был вызван [CORBET18].

Поскольку процесс создания контрольных точек не держит постоянно все файлы открытыми, может случиться так, что он пропустит уведомления об ошибках. Поскольку флаги «грязной» страницы будут очищены, процесс контрольных точек будет предполагать, что данные были успешно записаны на диск, хотя это может не соответствовать действительности.

Сочетание этих факторов может привести к потере данных или повреждению базы данных при возникновении потенциально восстанавливаемых сбоев. Такое поведение бывает трудно обнаружить, при этом восстановление может оказаться невозможным. Иногда источники и обстоятельства такого поведения могут быть весьма нетривиальными. Работая над механизмами восстановления, мы всегда должны проявлять особую осторожность, продумывать и пытаться протестировать все возможные сценарии сбоев.

Семантика журнала

Журнал упреждающей записи доступен только для добавления, и его записанное содержимое является неизменяемым, поэтому все операции записи в журнал являются последовательными. Поскольку журнал упреждающей записи — это неизменяемая структура данных, доступная только для добавления, считывающие процессы могут безопасно получать доступ ко всему содержимому журнала вплоть до порога записи, в то время как записывающий процесс продолжает добавлять данные в конец журнала.

Журнал упреждающей записи состоит из записей. Каждая запись имеет уникальный, монотонно увеличивающийся *порядковый номер журнала* (log sequence number, LSN). Обычно этот номер представляет собой внутренний счетчик или временную метку. Поскольку записи журнала не всегда занимают целый дисковый блок, их содержимое кэшируется в *буфере журнала* и выгружается на диск в ходе принудительной выгрузки. Принудительная выгрузка выполняется по мере заполнения буферов журнала и может запрашиваться диспетчером транзакций или кэшем страниц. Все записи журнала должны выгружаться на диск согласно порядковому номеру журнала LSN.

Помимо отдельных записей операций журнал упреждающей записи содержит записи, указывающие на завершение транзакции. Транзакция не считается зафиксированной, пока журнал не будет принудительно выгружен вплоть до номера записи фиксации транзакции.

Чтобы система могла продолжать корректно функционировать после сбоя, произошедшего во время отката или восстановления, некоторые системы во время отмены совершенных операций сохраняют в журнале *компенсационные записи журнала* (compensation log records, CLR).

Интерфейс между журналом упреждающей записи и основной структурой хранения обычно позволяет производить *усечение* журнала всякий раз, когда достигается контрольная точка. Ведение журнала — один из наиболее важных аспектов корректности базы данных, который довольно сложно правильно реализовать: даже малейшее рассогласование между процессом усечения журнала и процессом перемещения данных в основную структуру хранения может привести к потере данных.

Контрольные точки показывают, что все записи журнала вплоть до определенной отметки сохранены персистентным образом и их уже не нужно журналировать, что значительно сокращает объем работы, выполняемой при запуске базы данных. Процесс, который принудительно выгружает все «грязные» страницы на диск, обычно называется *контрольной точкой синхронизации*, поскольку он полностью синхронизирует основную структуру хранения.

Выгрузка всего содержимого на диск довольно непрактична и требует приостановки всех выполняемых операций до завершения обработки контрольной точки, поэтому в большинстве СУБД реализуются *нечеткие контрольные точки* (fuzzy checkpoints). В этом случае указатель `last_checkpoint`, хранящийся в заголовке журнала, содержит информацию о последней успешно обработанной контрольной точке. Нечеткая контрольная точка начинается со специальной записи журнала `begin_checkpoint`, указывающей ее начало, и заканчивается записью журнала `end_checkpoint`, содержащей информацию о «грязных» страницах и содержимом таблицы транзакций. До выгрузки всех страниц, определенных этой записью, контрольная точка считается *незавершенной*. Страницы выгружаются асинхронно, а сразу после выгрузки запись `last_checkpoint` обновляется порядковым номером LSN записи `begin_checkpoint`, с которого начинается процесс восстановления в случае сбоя [MOHAN92].

Использование операций и журнала данных

Некоторые СУБД, например System R [CHAMBERLIN81], используют *теневую подкачку*: метод «копирования при записи», обеспечивающий долговечность данных и атомарность транзакций. Новое содержимое помещается на новую неопубликованную *теневую* страницу и делается видимым путем переключения указателя со старой страницы на страницу, содержащую обновленное содержимое.

Любое изменение состояния можно представить образами «до» и «после» или соответствующими операциями повтора и отмены. Применение операции повтора к образу «до» создает образ «после». Сходным путем применение операции отмены к образу «после» создает образ «до».

Мы можем использовать физический журнал (хранящий полное состояние страницы или ее побайтовые изменения) или логический журнал (хранящий операции, кото-

рые необходимо выполнить, начиная с текущего состояния) для перевода записей или страниц из одного состояния в другое как назад, так и вперед во времени. Важно отслеживать точное состояние страниц, к которым могут быть применены записи физического и логического журналов.

При использовании физического журнала с образами «до» и «после» необходимо целиком журналировать затронутые операцией страницы, а при использовании логического журнала необходимо определить перечень применяемых к странице операций, таких как `вставка записи данных X для ключа Y`, вместе с соответствующими операциями отмены, такими как `удаление значения, связанного с ключом Y`.

На практике многие СУБД сочетают эти два подхода, используя логический журнал для отмены (в целях обеспечения параллелизма и повышения производительности) и физический журнал для повтора (в целях снижения времени восстановления) [MOHAN92].

Политики кражи и принуждения

Для определения подходящего момента для выгрузки на диск произведенных в памяти изменений СУБД используют политики «кражи»/«без кражи» (steal/no steal) и «принуждения»/«без принуждения» (force/no force). Эти политики распространяются *главным образом* на кэш страниц, но их лучше обсуждать в контексте восстановления, поскольку они в значительной мере определяют, какие методы восстановления можно будет использовать в сочетании с ними.

Метод восстановления, который позволяет выгрузить измененную транзакцией страницу еще до фиксации транзакции, называется политикой «кражи». Политика «без кражи» не позволяет выгружать на диск содержимое незафиксированных транзакций. Под «воровством» «грязной» страницы при этом понимается выгрузка ее содержимого из памяти на диск с загрузкой на ее место другой страницы с диска.

Политика «принуждения» требует, чтобы все измененные транзакциями страницы были выгружены на диск до фиксации транзакций. С другой стороны, политика «без принуждения» позволяет транзакции фиксироваться, даже если некоторые из измененных ею страниц еще не были выгружены на диск. Под «принуждением» «грязной» страницы при этом понимается ее выгрузка на диск до фиксации.

Вы должны четко понимать, как работают политики воровства и принуждения, поскольку они влияют на отмену и повтор транзакций. Операция отмены откатывает обновления на принудительно выгруженных страницах для зафиксированных транзакций, а повтор применяет изменения, выполненные зафиксированными транзакциями на диске.

Использование политики «без кражи» (no-steal) позволяет реализовать восстановление, используя только записи повтора: старая копия содержится на странице на диске, а модификация хранится в журнале [WEIKUM01]. При использовании политики «без принуждения» мы потенциально можем буферизировать обновления

страниц путем их *откладывания*. Поскольку в течение этого времени содержимое страницы должно храниться в памяти, может потребоваться кэш страниц большего размера.

При использовании политики «принуждения» (force) восстановление после сбоя не требует дополнительных действий для восстановления результатов зафиксированных транзакций, так как измененные этими транзакциями страницы уже будут выгружены. Основным недостатком использования этого подхода является то, что время выполнения транзакций увеличивается из-за операций ввода-вывода.

В общем случае, пока транзакция не зафиксирована, мы должны иметь достаточно информации, чтобы можно было отменить ее результаты. Если некоторая часть затронутых транзакцией страниц выгружается на диск, мы должны хранить необходимую для отмены информацию в журнале до фиксации, чтобы можно было откатить транзакцию. В противном случае в журнале до фиксации необходимо хранить записи повтора. В обоих случаях транзакция не может быть зафиксирована до тех пор, пока в файл журнала не будут сделаны записи отмены или повтора.

ARIES

ARIES — это алгоритм восстановления, использующий политику «воровства/без принуждения» (steal/no-force). Он использует физический повтор для повышения производительности во время восстановления (поскольку так изменения можно применить быстрее) и логическую отмену для улучшения параллелизма во время нормальной работы (так как логические операции отмены могут применяться к страницам независимо). В алгоритме используется журнал упреждающей записи для применения истории во время восстановления, чтобы полностью восстановить состояние базы данных перед отменой незафиксированных транзакций, и создаются компенсационные записи журнала во время отмены [MOHAN92].

В ходе перезапуска СУБД после сбоя восстановление происходит в три этапа:

1. На этапе *анализа* выявляются «грязные» страницы в кэше страниц и транзакции, выполнявшиеся в момент появления сбоя. Информация о «грязных» страницах используется для определения начальной точки для этапа повтора. Список незавершенных транзакций используется на этапе отмены для их отката.
2. Этап *повтора* вновь применяет историю вплоть до момента сбоя и восстанавливает прежнее состояние базы данных. Этот этап выполняется для незавершенных транзакций, а также для транзакций, которые были зафиксированы без выгрузки их содержимого на персистентный носитель.
3. Этап *отмены* откатывает все незавершенные транзакции и восстанавливает последнее согласованное состояние базы данных. Все операции откатываются в обратном хронологическом порядке. На случай повторного сбоя базы данных во время восстановления все операции, отменяющие транзакции, также регистрируются в журнале во избежание их повторного выполнения.

Алгоритм ARIES использует порядковые номера журнала LSN для идентификации записей журнала, отслеживает страницы, измененные при выполнении транзакций, в таблице «грязных» страниц и использует физический повтор, логическую отмену и нечеткие контрольные точки. Несмотря на то что статья с описанием этого алгоритма была опубликована в еще 1992 году, большинство представленных в ней концепций, подходов и парадигм актуальны в сфере обработки транзакций и восстановления данных и по сей день.

Управление параллелизмом

При обсуждении архитектуры СУБД в разделе «Архитектура СУБД» на с. 25 мы упомянули, что диспетчер транзакций и диспетчер блокировок совместно обеспечивают управление параллелизмом. Управление параллелизмом — это набор методов, обеспечивающих взаимодействие параллельно выполняемых транзакций. Эти методы можно разделить на следующие основные категории:

Оптимистичное управление параллелизмом (Optimistic concurrency control, OCC)

Позволяет транзакциям выполнять параллельные операции чтения и записи и определяет, является ли результат комбинированного выполнения сериализуемым. То есть транзакции не блокируют друг друга, сохраняют истории своих операций и проверяют эти истории на возможные конфликты перед фиксацией. Если выполнение приводит к конфликту, одна из конфликтующих транзакций прерывается.

Управление параллелизмом с несколькими версиями (multiversion concurrency control, MVCC)

Гарантирует согласованное представление базы данных в какой-то момент в прошлом, идентифицируемый временно́й меткой, допуская создание нескольких версий записи с разными временны́ми метками. Этот подход может быть реализован с помощью методов проверки, позволяющих выиграть только одной из обновляющихся или фиксирующихся транзакций, а также с помощью методов без блокировки, таких как сортировка по временны́м меткам, или методов на основе блокировки, таких как двухфазная блокировка.

Пессимистичное (консервативное) управление параллелизмом (pessimistic concurrency control, PCC)

Существуют и блокирующие, и неблокирующие консервативные методы, которые различаются по способам управления общими ресурсами и предоставления доступа к ним. Методы на основе блокировки требуют, чтобы транзакции поддерживали блокировки записей базы данных, чтобы другие транзакции не могли изменить заблокированные записи и получить доступ к изменяемым записям до тех пор, пока транзакция не снимет свои блокировки. Неблокирующие методы поддерживают списки операций чтения и записи и ограничивают выполнение в зависимости от планирования незавершенных транзакций. Пессимистичное

планирование работы может привести к взаимной блокировке, когда несколько транзакций ждут друг от друга снятия блокировки для продолжения своей работы.

В этой главе мы сосредоточимся на методах управления локальным параллелизмом внутри узлов. В главе 13 вы можете найти информацию о распределенных транзакциях и других подходах, таких как детерминированное управление параллелизмом (см. раздел «Распределенные транзакции с использованием протокола Calvin» на с. 284).

Перед тем как перейти к более подробному обсуждению управления параллелизмом, мы должны определить набор решаемых нами задач и установить, насколько операции транзакций могут перекрывать друг друга и к каким последствиям это может приводить.

Сериализуемость

Транзакции состоят из операций чтения и записи, выполняемых в отношении состояния базы данных, и бизнес-логики (преобразований, применяемых к считываемому содержимому). Под *планом* (plan) при этом понимается список операций, необходимых для выполнения набора транзакций с точки зрения СУБД (т. е. только тех, которые взаимодействуют с состоянием базы данных, как, например, операции чтения, записи, фиксации или прерывания), поскольку все остальные операции считаются свободными от побочных эффектов (т. е. не оказывают влияния на состояние базы данных) [MOLINA08].

План является *полным* (complete), если включает в себя все операции каждой транзакции, выполняемой согласно этому плану. *Корректные* (correct) планы логически эквивалентны исходному списку операций, но в целях оптимизации их составные части могут выполняться параллельно или в ином порядке, если это не нарушает ACID-свойства и корректность результатов отдельных транзакций [WEIKUM01].

План считается *последовательным* (sequential), когда транзакции в нем выполняются абсолютно независимо друг от друга и без какого-либо перемежения: каждая предыдущая транзакция выполняется полностью до начала следующей. Последовательным выполнением легко управлять, в отличие от всех возможных вариантов перемежений между несколькими многоступенчатыми транзакциями. Однако если мы будем всегда выполнять транзакции поочередно, это серьезно ограничит пропускную способность системы и снизит производительность.

Нам нужно найти способ, позволяющий выполнять операции транзакций параллельно с сохранением корректности и простоты последовательного плана. Этого можно добиться с помощью *сериализуемых* планов. План является *сериализуемым* (serializable), если он эквивалентен некоторому полному последовательному плану для того же набора транзакций. То есть такой план дает тот же результат, что и поочередное выполнение набора транзакций в определенной последовательности. На рис. 5.4 показаны три параллельные транзакции и возможные последовательности их выполнения (всего имеется 3! = 6 вариантов последовательности выполнения).

Рис. 5.4. Одновременные транзакции и возможные последовательности их выполнения

Изолированность транзакций

Транзакционные СУБД позволяют использовать различные уровни изолированности. *Уровень изолированности* (isolation level) определяет, как и когда составные части транзакции могут и должны стать видимыми для других транзакций. То есть уровни изолированности определяют степень изолированности транзакций от других одновременно выполняемых транзакций, а также то, с какими аномалиями можно столкнуться во время выполнения.

Обеспечение изолированности сопряжено с определенными издержками: чтобы предотвратить выход неполных или временны́х операций записи за пределы транзакций, требуется осуществлять дополнительную координацию и синхронизацию, что негативно сказывается на производительности.

Аномалии чтения и записи

Стандарт языка запросов SQL [MELTON06] упоминает и описывает *аномалии чтения*, которые могут возникать во время выполнения параллельных транзакций: «грязные», неповторяемые и «фантомные» (phantom reads) операции чтения.

«Грязное» чтение (dirty read) — это ситуация, при которой транзакция может читать незафиксированные изменения из других транзакций. Например, транзакция T_1 обновляет запись пользователя новым значением поля адреса, а транзакция T_2 считывает обновленный адрес до фиксации T_1. Транзакция T_1 прерывается и откатывает свои результаты выполнения. Однако T_2 уже успела прочитать это значение и, таким образом, получила доступ к значению, которое не было зафиксировано.

Неповторяемое чтение (nonrepeable read) (иногда называемое нечетким чтением) — это ситуация, при которой транзакция запрашивает одну и ту же строку дважды и получает разные результаты. Например, это может произойти, если транзакция T_1 считает некоторую строку, а затем транзакция T_2 изменит ее и зафиксирует это изменение. Если T_1 запросит ту же строку снова до завершения своего выполнения, результат будет не таким, как в первый раз.

Если в ходе транзакции производится чтение диапазона (т. е. считывается не одна запись данных, а целый диапазон записей), мы можем увидеть *фантомные записи* (phantom records). *Фантомное чтение* (phantom read) — это ситуация, когда транзакция дважды запрашивает один и тот же набор строк и получает разные результаты. Это похоже на неповторяемое чтение, но относится к запросам диапазонов.

Существуют также *аномалии записи* (write anomalies) с аналогичной семантикой: потерянное обновление, «грязная» запись и искажение записи.

Потерянное обновление (lost update) — это ситуация, когда две транзакции T_1 и T_2 пытаются обновить значение `V`. T_1 и T_2 считывают значение `V`. T_1 обновляет `V` и фиксируется, а T_2 обновляет `V` после этого и тоже фиксируется. Если обеим транзакциям разрешено совершить фиксацию, то, поскольку они не знают о существовании друг друга, результаты транзакции T_1 будут перезаписаны результатами транзакции T_2 и, таким образом, обновление транзакции T_1 будет потеряно.

«Грязная» запись (dirty write) — это ситуация, при которой одна из транзакций принимает незафиксированное значение (т. е. производит «грязное» чтение), после чего изменяет и сохраняет его. То есть в данном случае результаты транзакции основаны на значениях, которые не были зафиксированы.

Искажение записи (write skew) происходит, когда каждая отдельная транзакция соблюдает требуемые инварианты, но комбинация их действий не удовлетворяет этим инвариантам. Например, транзакции T_1 и T_2 изменяют значения двух счетов A_1 и A_2. Вначале A_1 содержит `100$`, а A_2 — `150$`. Значение счета может быть отрицательным, если сумма двух счетов неотрицательна: A_1 + A_2 >= 0. Транзакция T_1 пытается снять `200$` со счета A_1, а транзакция T_2 — со счета A_2. Так как в момент начала этих транзакций A_1 + A_2 = `250$`, то в сумме доступно `250$`. Обе транзакции предполагают, что они сохраняют инвариант и могут быть зафиксированы. Однако после фиксации счет A_1 будет содержать `100$`, а счет A_2 — `−50$`, что явно нарушает требование о том, чтобы сумма счетов всегда была положительной [FEKETE04].

Уровни изолированности

Самым низким (или, другими словами, самым слабым) уровнем изолированности является *чтение незафиксированных данных* (read uncommitted). При таком уровне изолированности транзакционная система позволяет одной транзакции видеть незафиксированные изменения других параллельных транзакций. То есть в данном случае допускается выполнение «грязных» операций чтения.

Мы можем исключить возможность возникновения некоторых аномалий. Например, мы можем добиться, чтобы любая операция чтения, выполняемая конкретной транзакцией, могла считывать только уже зафиксированные изменения. Однако это не гарантирует, что транзакция увидит то же самое значение, если снова попытается прочитать ту же запись данных на более позднем этапе. Если в промежутке между двумя операциями чтения будет зафиксирована модификация, два запроса от одной транзакции дадут разные результаты. То есть, исключив возможность выполнения

«грязных» операций чтения, мы не исключаем возможность выполнения фантомных и неповторяемых операций чтения. Такой уровень изолированности называют *чтением зафиксированных данных* (read committed). Если мы также исключим возможность выполнения неповторяемых операций чтения, мы получим уровень изолированности, называемый повторяемым чтением.

Самым сильным уровнем изолированности является сериализуемость. Как уже говорилось в подразделе «Сериализуемость» на с. 112, этот уровень гарантирует, что результаты транзакций будут появляться в некотором порядке, как если бы транзакции выполнялись *последовательно* (т. е. без перекрытия во времени). Запрет на параллельное выполнение существенно снижает производительность базы данных. Вы можете менять последовательность выполнения транзакций при условии удовлетворения их внутренних инвариантов, а также выполнять их параллельно при условии, что их результаты будут появляться в некотором последовательном порядке.

На рис. 5.5 показаны уровни изолированности и допускаемые ими аномалии.

	Грязное	Неповторяемое	Фантомное
Чтение незафиксированных данных	Допускается	Допускается	Допускается
Чтение фиксированных данных	–	Допускается	Допускается
Повторяемое чтение	–	–	–
Сериализуемость	–	–	–

Рис. 5.5. Уровни изолированности и допускаемые аномалии

Транзакции без зависимостей могут выполняться в любом порядке, поскольку их результаты полностью независимы. В отличие от свойства линеаризуемости (которое будет рассмотрено в контексте распределенных систем в подразделе «Линеаризуемость» на с. 241), сериализуемость — это свойство множества операций, выполняемых в произвольном порядке. Оно не подразумевает и не пытается задать какой-либо конкретный порядок выполнения транзакций. Изолированность в терминах ACID-свойств означает сериализуемость [BAILIS14a]. К сожалению, реализация сериализуемости требует определенной координации. То есть вы должны координировать параллельно выполняемые транзакции для сохранения инвариантов и задавать последовательный порядок при возникновении конфликтов в ходе их выполнения [BAILIS14b].

Некоторые базы данных используют *изоляцию моментальных снимков* (snapshot isolation), при которой транзакция может видеть изменения состояния, выполненные всеми транзакциями, которые были зафиксированы к моменту ее запуска. Каждая транзакция создает моментальный снимок данных и обрабатывает запросы относительно него. Этот моментальный снимок не может меняться во время выполнения

транзакции. Транзакция фиксируется только в том случае, если значения, которые она модифицировала, не менялись во время ее выполнения. В противном случае она прерывается и откатывается назад.

Когда две транзакции пытаются изменить одно и то же значение, только одна из них может быть зафиксирована. Это исключает возможность возникновения аномалии *потерянного обновления*. Например, допустим, что транзакции T_1 и T_2 пытаются изменить переменную V. Они считывают текущее значение переменной V из снимка, содержащего изменения всех транзакций, которые были зафиксированы до их запуска. Та транзакция, которая попытается зафиксироваться первой, будет зафиксирована, а вторая транзакция будет прервана. Вместо того чтобы перезаписывать значение, прерванная транзакция предпримет повторную попытку выполнения.

При использовании изоляции моментального снимка может произойти аномалия *искажения записи*, так как если две транзакции будут производить чтение из локального состояния, изменять независимые записи и сохранять локальные инварианты, то им обоим будет разрешена фиксация [FEKETE04]. Мы рассмотрим изоляцию моментальных снимков более подробно в контексте распределенных транзакций в разделе «Распределенные транзакции с использованием библиотеки Percolator» на с. 290.

Оптимистичное управление параллелизмом

Оптимистичное управление параллелизмом предполагает, что конфликты транзакций происходят редко, и, вместо того чтобы использовать блокировки и блокировать выполнение транзакций, мы можем проверять транзакции, чтобы исключить возможность конфликтов чтения/записи с параллельно выполняемыми транзакциями и обеспечить сериализуемость перед фиксацией результатов. Как правило, выполнение транзакции разбивается на три этапа [WEIKUM01]:

Этап чтения

Транзакция выполняет свои стадии в собственном частном контексте, не делая какие-либо изменения видимыми для других транзакций. После этого этапа известны все зависимости транзакции (*набор чтения*), а также производимые транзакцией побочные эффекты (*набор записи*).

Этап проверки

Наборы чтения и записи параллельных транзакций проверяются на наличие возможных конфликтов между их операциями, способных нарушать сериализуемость. Если некоторые данные, которые считывала транзакция, теперь устарели или она предполагает перезапись некоторых значений, записанных транзакциями, которые были зафиксированы во время ее этапа чтения, ее частный контекст очищается, а этап чтения перезапускается. Другими словами, этап проверки определяет, сохраняются ли ACID-свойства после фиксации транзакции.

Этап записи

Если этап проверки не выявил каких-либо конфликтов, транзакция может зафиксировать свой набор записи из частного контекста в общем состоянии базы данных.

Проверку можно выполнить путем проверки на наличие конфликтов с транзакциями, которые уже были зафиксированы (*обратная ориентация*), или с транзакциями, которые в настоящее время находятся на стадии проверки (*прямая ориентация*). Этапы проверки и записи различных транзакций должны выполняться атомарно. Ни одна транзакция не может быть зафиксирована во время проверки какой-либо другой транзакции. Поскольку фазы проверки и записи обычно короче фазы чтения, это приемлемый компромисс.

Управление параллелизмом с обратной ориентацией гарантирует, что для любой пары транзакций T_1 и T_2 выполняются следующие свойства:

- T_1 была зафиксирована до начала фазы чтения T_2, поэтому для T_2 разрешена фиксация.
- T_1 была зафиксирована до начала фазы записи T_2, и набор записи T_1 не пересекается с набором чтения T_2. Другими словами, T_1 не записала ни одно из тех значений, которые необходимо видеть транзакции T_2.
- Фаза чтения T_1 завершилась до фазы чтения T_2, и набор записи T_2 не пересекается с наборами чтения и записи T_1. Другими словами, транзакции работают с независимыми наборами данных, поэтому им обеим разрешена фиксация.

Этот подход эффективен, если проверка, как правило, проходит успешно и транзакции не нужно повторять, так как повторные попытки весьма негативно влияют на производительность. Конечно, оптимистичный параллелизм все равно подразумевает наличие некоторой *критической секции*, в которую транзакции могут входить лишь поочередно. Еще один способ обеспечить неисключительное владение для некоторых операций состоит в том, чтобы использовать блокировки чтения-записи (чтобы разрешить общий доступ для чтения) и обновляемые блокировки (чтобы разрешить преобразование общих блокировок в исключительные по мере необходимости).

Многоверсионное управление конкурентным доступом

Многоверсионное управление конкурентным доступом (multiversion concurrency control, MVCC) обеспечивает согласованность транзакций в СУБД, допуская наличие нескольких версий каждой записи и используя монотонно увеличивающиеся идентификаторы транзакций или временны́е метки. Это позволяет выполнять операции чтения и записи с минимальной координацией на уровне хранилища, поскольку операции чтения могут обращаться к старым значениям до тех пор, пока не будут зафиксированы новые.

Многоверсионное управление конкурентным доступом различает *зафиксированные* и *незафиксированные* версии, которые, соответственно, представляют собой версии значений зафиксированных и незафиксированных транзакций. Предполагается, что последней зафиксированной версией значения является его текущая версия. При этом диспетчер транзакций обычно следит за тем, чтобы в каждый момент было не более одного незафиксированного значения.

В зависимости от того, какой уровень изолированности реализует СУБД, для операций чтения может быть разрешен либо нет доступ к незафиксированным значениям [WEIKUM01]. Многоверсионное управление конкурентным доступом можно реализовать с помощью методов блокировки, планирования и разрешения конфликтов (например, двухфазной блокировки) или упорядочения по временным меткам. Одним из основных сценариев использования такого способа управления параллелизмом является реализация изоляции моментальных снимков [HELLERSTEIN07].

Пессимистичное управление параллелизмом

Пессимистичные схемы управления параллелизмом более консервативны, чем оптимистичные. Такие схемы выявляют конфликтующие транзакции в ходе их выполнения и блокируют или прерывают их выполнение.

Одной из простейших схем пессимистичного (без блокировки) управления параллелизмом является *упорядочение по временным меткам*, при котором каждая транзакция имеет свою метку. Допустимость выполнения операций транзакции определяется тем, были ли уже зафиксированы все транзакции с более ранней временной меткой. Для этого диспетчер транзакций должен поддерживать для каждого значения параметры `max_read_timestamp` и `max_write_timestamp`, описывающие операции чтения и записи, выполняемые параллельными транзакциями.

Когда операция чтения пытается прочитать значение, временная метка которого меньше, чем `max_write_timestamp`, соответствующая транзакция прерывается, так как уже существует более новое значение и выполнение этой операции нарушило бы порядок транзакций.

Аналогичным образом операции записи с временной меткой, меньшей, `чем max_read_timestamp`, будут конфликтовать с более поздними операциями чтения. Однако операции записи с временной меткой, меньшей, чем `max_write_timestamp`, разрешены, так как мы можем безопасно игнорировать ранее записанные значения. Эта гипотеза известна как *«правило записи Томаса»* [THOMAS79]. При выполнении операций чтения или записи обновляются соответствующие максимальные значения временных меток. Прерванные транзакции перезапускаются с новой временной меткой, так как в противном случае они гарантированно будут прерваны снова [RAMAKRISHNAN03].

Управление параллелизмом на основе блокировки

Схемы управления параллелизмом на основе блокировки представляют собой разновидность пессимистичного управления, которая использует явные блокировки объектов базы данных вместо составления планов, подобно тому как это делают такие протоколы, как упорядочение по временным меткам. Использование блокировок несет с собой и некоторые недостатки — в частности, это несет проблемы с масштабируемостью и конкуренцией за доступ к объектам [REN16].

Одним из наиболее распространенных методов на основе блокировки является *двухфазная блокировка* (2-phase locking, 2PL), которая разделяет управление блокировкой на две фазы:

- *Фаза роста* (также называемая *фазой расширения*), во время которой транзакция устанавливает все необходимые ей блокировки и ни одна блокировка не освобождается.
- *Фаза сокращения*, во время которой освобождаются все блокировки, установленные во время фазы роста.

Из этих двух определений вытекает следующее правило: транзакция не может устанавливать блокировки после того, как она освободит хотя бы одну из них. Важно отметить, что метод 2PL не запрещает транзакциям выполнять свои операции во время любой из этих фаз; хотя некоторые его разновидности (в частности, консервативный 2PL) все же накладывают такие ограничения.

Несмотря на схожие названия, двухфазная блокировка не имеет ничего общего с двухфазной фиксацией (см. раздел «Двухфазная фиксация» на с. 277). Двухфазная фиксация — это протокол, используемый для распределенных многосоставных транзакций, в то время как двухфазная блокировка — это механизм управления параллелизмом, часто используемый для реализации сериализации.

Взаимоблокировки

При использовании протоколов блокировки транзакции пытаются установить блокировки объектов базы данных, и в случае, если некоторая блокировка не может быть предоставлена немедленно, транзакции приходится ждать, пока эта блокировка освободится. При этом может возникнуть ситуация, при которой, пытаясь установить блокировки, необходимые для продолжения выполнения, каждая из двух транзакций будет ждать, пока другая освободит удерживаемые ею блокировки. Эта ситуация называется *взаимоблокировкой*.

На рис. 5.6 показан пример взаимоблокировки: транзакция T_1 удерживает блокировку L_1 и ждет, пока освободится блокировка L_2, в то время как транзакция T_2 удерживает блокировку L_2 и ждет, пока освободится блокировка L_1.

Самый простой способ решения проблемы взаимоблокировок состоит в том, чтобы установить некоторое максимальное время ожидания и прерывать долго выполняющиеся транзакции исходя из того предположения, что могли оказаться в ситуации взаимоблокировки. Другая стратегия — консервативный 2PL — требует, чтобы транзакция устанавливала все блокировки до выполнения любой из своих операций и чтобы в том случае, когда это не удается, выполнение транзакции прерывалось. Однако поскольку эти подходы существенно ограничивают параллелизм системы, в большинстве СУБД выявление или исключение взаимоблокировок обеспечивает диспетчер транзакций.

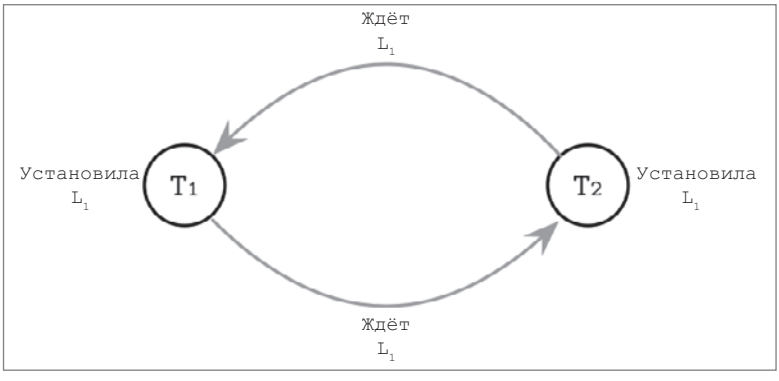

Рис. 5.6. Пример взаимоблокировки

Выявление взаимоблокировок обычно производится с помощью *графа ожидания* (waits-for graph), который отслеживает отношения между текущими транзакциями и показывает отношения ожидания между ними.

Наличие циклов на этом графике говорит о наличии взаимоблокировки: транзакция T_1 ждет транзакцию T_2, которая, в свою очередь, ждет транзакцию T_1. Поиск взаимоблокировок может выполняться периодически (с некоторым временным интервалом) или непрерывно (при каждом обновлении графа ожидания) [WEIKUM01]. При этом прерывается одна из транзакций (обычно та, которая пыталась установить блокировку последней).

Чтобы избежать взаимоблокировок и устанавливать блокировки только в тех случаях, которые не приводят к взаимоблокировке, диспетчер транзакций может использовать временные метки транзакций для определения их приоритета. При этом более ранняя временная метка обычно подразумевает более высокий приоритет и наоборот.

Если транзакция T_1 пытается установить блокировку, удерживаемую в данный момент транзакцией T_2, и транзакция T_1 имеет более высокий приоритет (она началась раньше транзакции T_2), мы можем использовать одно из следующих ограничений, чтобы избежать взаимоблокировок [RAMAKRISHNAN03]:

Ожидание-отмена (wait-die)

Транзакции T_1 разрешается блокировать и ждать блокировки. В противном случае транзакция T_1 прерывается и перезапускается. Другими словами, транзакция может быть заблокирована только транзакцией с более поздней временной меткой.

Отмена-ожидание (wound-wait)

Транзакция T_2 прерывается и перезапускается (транзакция T_1 отменяет транзакцию T_2). В противном случае (если транзакция T_2 началась раньше транзакции T_1) транзакции T_1 разрешается ждать. Другими словами, транзакция может быть заблокирована только транзакцией с более ранней временной меткой.

Обработка транзакций требует наличия планировщика для работы с взаимоблокировками. В то же время при использовании защелок (см. подраздел «Защелки» ниже) программист должен сам обеспечивать отсутствие взаимоблокировок, не полагаясь на какие-либо механизмы предотвращения взаимоблокировок.

Блокировки

Если две транзакции отправляют данные одновременно, изменяя перекрывающиеся сегменты данных, ни одна из них не должна видеть частичные результаты другой, чтобы сохранялась логическая согласованность. Точно так же два потока одной транзакции должны видеть одно и то же содержимое базы данных и иметь доступ к данным друг друга.

В сфере обработки транзакций проводится различие между механизмами сохранения логической и физической целостности данных. За обеспечение логической и физической целостности отвечают соответственно *блокировки* (locks) и *защелки* (latches). Это не слишком удачные названия, поскольку то, что здесь называют «защелками», в системном программировании обычно называют «блокировками», однако в этом разделе будет четко показано, что подразумевают эти термины.

Блокировки используются для изоляции и планирования перекрывающихся транзакций, а также для управления содержимым базы данных (за исключением внутренней структуры хранения) и устанавливаются на некоторый ключ. Блокировки могут защищать либо конкретный ключ (как существующий, так и несуществующий), либо целый диапазон ключей. Блокировки обычно хранятся и администрируются за пределами реализации дерева, представляя собой концепцию более высокого уровня, которой управляет диспетчер блокировок базы данных.

Блокировки более «тяжеловесны» по сравнению с защелками и удерживаются на протяжении всего времени выполнения транзакции.

Защелки

С другой стороны, защелки защищают *физическое* представление: содержимое листовых страниц изменяется во время операций вставки, обновления и удаления. Содержимое нелистовых страниц и структура дерева изменяются во время операций, ведущих к разделениям и слияниям, которые распространяются от листьев, подвергшихся потере значимости и переполнению. Защелки защищают физическое представление дерева (содержимое страниц и структуру дерева) во время таких операций и устанавливаются на уровне страниц. Любая страница должна быть защищена защелкой, чтобы обеспечить безопасный параллельный доступ к ней. Использовать защелки должны даже те методы управления параллелизмом, которые не используют блокировки.

Поскольку одна модификация на листовом уровне может распространяться на более высокие уровни B-дерева, защелки иногда требуется устанавливать на нескольких уровнях. Выполняемые запросы не должны видеть страницы в несогласованном состоянии, например при неполной записи или частичном разделении узлов, когда

данные находятся и в исходном, и в целевом узле или еще не распространились до родительского узла.

Те же правила справедливы и в случае обновления указателей на родителя или одноуровневый элемент. В общем случае для повышения степени параллелизма следует удерживать защелку в течение минимально возможного срока — а именно во время считывания или обновления страницы.

Конфликты между параллельными операциями можно грубо разделить на три категории:

- *Одновременное чтение*, когда несколько потоков обращаются к одной и той же странице, не изменяя ее.
- *Одновременное обновление*, когда несколько потоков пытаются внести изменения в одну и ту же страницу.
- *Чтение во время записи*, когда один поток пытается изменить содержимое страницы, а другой — получить доступ к той же странице для чтения.

Эти сценарии также распространяются на тот случай, когда операции доступа конфликтуют с обслуживанием базы данных (например, с фоновыми процессами, описанными в разделе «Очистка и обслуживание» на с. 92).

Блокировка чтения-записи

Простейший способ реализации защелки состоит в том, чтобы предоставлять запрашивающему потоку эксклюзивный доступ на чтение/запись. Однако в большинстве случаев нам не нужно изолировать друг от друга *все* процессы. Например, операции чтения могут обращаться к страницам одновременно, не вызывая никаких проблем, и нам остается лишь проследить за тем, чтобы не перекрывались несколько одновременных *операций записи* и чтобы *операции чтения* не пересекались с *операциями записи*. Обеспечить такой уровень гранулярности можно с помощью блокировки чтения-записи, или RW-блокировки.

Блокировка чтения-записи позволяет нескольким операциям чтения обращаться к объекту одновременно; при этом эксклюзивный доступ к объекту должны получать только операции записи (которых обычно меньше). На рис. 5.7 показана таблица совместимости для блокировок чтения-записи: только операции чтения могут совместно владеть блокировкой, в то время как при всех остальных комбинациях операций чтения и операций записи необходимо получать эксклюзивное право доступа.

На рис. 5.8 (*а*) мы видим, что к объекту обращается несколько операций чтения, в то время как операция записи ждет своей очереди, поскольку она не может изменить страницу, пока к ней обращаются операции чтения. На рис. 5.8 (*б*) операция записи 1 удерживает эксклюзивную блокировку объекта, в то время как другая операция записи и три операции чтения вынуждены ждать.

Когда две перекрывающиеся операции чтения пытаются обратиться к одной и той же странице, необходимая синхронизация сводится лишь к тому, чтобы исключить

повторное извлечение страницы с диска кэшем страниц, поэтому операции чтения могут безопасно выполняться параллельно в совмещаемом режиме. Однако как только в игру вступают операции записи, мы должны изолировать их и от параллельных операций чтения, и от других операций записи.

	Операция чтения	Операция записи
Операция чтения	Общая	Эксклюзивная
Операция записи	Эксклюзивная	Эксклюзивная

Рис. 5.7. Таблица совместимости для блокировок чтения-записи

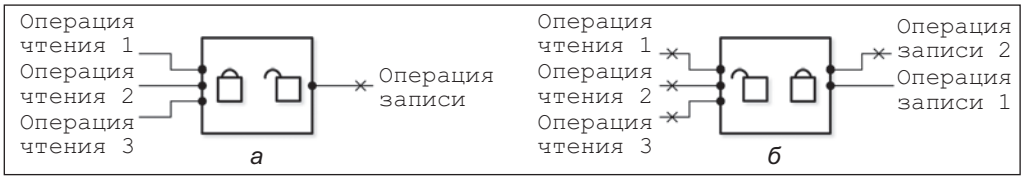

Рис. 5.8. Блокировки чтения-записи

МЕТОДЫ НА ОСНОВЕ АКТИВНОГО ОЖИДАНИЯ И ОЧЕРЕДЕЙ

Для управления общим доступом к страницам мы можем либо использовать алгоритмы блокировки, которые откладывают выполнение потоков и пробуждают их после того, как появляется возможность продолжить работу, либо использовать алгоритмы активного ожидания. Алгоритмы активного ожидания дают потокам возможность подождать в течение некоторого небольшого срока вместо того, чтобы передавать управление планировщику.

Очередь обычно реализуется с помощью операций «Сравнение с обменом», позволяющих выполнять операции, гарантируя установление блокировки и атомарное обновление очереди. Если очередь пуста, поток получает доступ немедленно. В противном случае поток присоединяется к очереди ожидания и ждет изменения переменной, которая может быть обновлена только предстоящим в очереди потоком. Такой подход помогает уменьшить объем запросов к процессору на установление и освобождение блокировок [MELLORCRUMMEY91].

Объединение защелок

Самый простой подход к установке защелок сводится к тому, чтобы захватывать все защелки на пути от корня до целевого листа. При этом мы получаем узкое место в обеспечении параллелизма, чего в большинстве случаев можно избежать. Защелка

должна удерживаться в течение минимально возможного времени. Для этого, в частности, можно использовать такой способ оптимизации, как объединение защелок [RAMAKRISHNAN03].

Объединение защелок — это довольно простой метод, который позволяет уменьшить длительность удержания защелок за счет освобождения их после того, как они перестают быть нужными выполняющейся операции. При этом в ходе чтения защелка родительского узла может быть освобождена после нахождения дочернего узла и установления его защелки.

Во время вставки защелка родительского узла может быть освобождена, если операция гарантированно не приведет к структурным изменениям, способным распространиться до родительского узла. То есть родительская защелка может быть освобождена, если дочерний узел не заполнен.

Аналогичным образом во время удаления защелка родительского узла может быть освобождена, если дочерний узел содержит достаточно элементов и операция не приведет к слиянию одноуровневых узлов.

ОБНОВЛЕНИЕ ЗАЩЕЛОК И ПРЕСЛЕДОВАНИЕ УКАЗАТЕЛЯ

Вместо немедленной установки защелок в эксклюзивном режиме во время обхода узлов можно производить *обновление защелок* (latch upgading). Этот подход подразумевает установку совмещаемых блокировок вдоль пути поиска с обновлением их до исключительной блокировки по мере необходимости.

Изначально операции записи устанавливают исключительные блокировки только на листовом уровне. Если лист нужно подвергнуть разделению или слиянию, алгоритм идет вверх по дереву и пытается обновить удерживаемую родителем совмещаемую блокировку, приобретая эксклюзивное право владения защелками для затрагиваемой части дерева (т. е. узлы, которые также будут подвергнуты разделению или слиянию в результате этой операции). Поскольку несколько потоков могут попытаться установить эксклюзивные блокировки на одном из более высоких уровней, одному из них придется подождать или перезапуститься.

Возможно, вы заметили, что все рассмотренные до сих пор механизмы начинают свою работу с установки защелки на корневом узле. Каждый запрос должен проходить через корневой узел, который быстро становится узким местом. В то же время корень всегда подвергается разделению в последнюю очередь, так как для этого сначала должны заполниться все его потомки. Это означает, что для корневого узла установка защелки может *всегда* производиться согласно оптимистичному подходу и за это придется редко платить необходимостью выполнения повторной попытки (*преследования указателя* (pointer chasing)).

На рис. 5.9 показан процесс прохождения от корня к листу во время вставки:

а) Устанавливается защелка записи на корневом уровне.

б) Находится узел следующего уровня и устанавливается его защелка записи. Узел проверяется на возможные структурные изменения. Поскольку узел не заполнен, родительская защелка может быть освобождена.

в) Операция переходит на следующий уровень. Устанавливается защелка записи, целевой листовой узел проверяется на возможные структурные изменения, освобождается родительская защелка.

Это оптимистичный подход: большинство операций вставки и удаления не вызывают структурных изменений, распространяющихся на несколько уровней вверх.

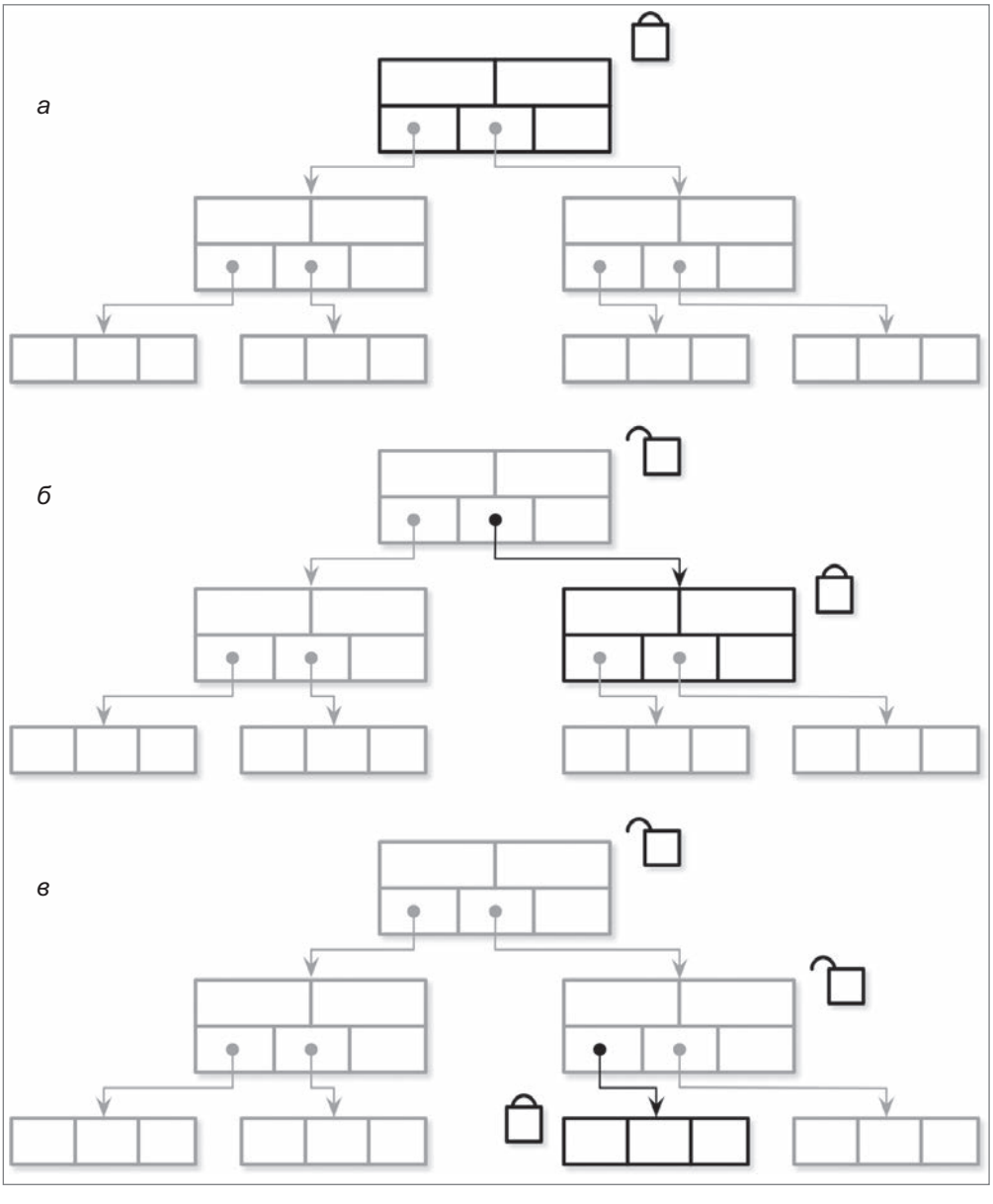

Рис. 5.9. Объединение защелок во время вставки

На самом деле вероятность структурных изменений уменьшается на более высоких уровнях. Большинство операций требуют установки защелки только на целевом узле, и число случаев, когда установку родительской защелки необходимо сохранить, относительно невелико.

Если дочерняя страница еще не загружена в кэш страниц, мы можем либо установить защелку для страницы, подлежащей загрузке, либо освободить родительскую защелку и перезапустить проход от корня к листу после загрузки страницы, чтобы снизить конкуренцию. Хотя может показаться, что повторение прохода от корня к листу — слишком затратный подход, в действительности это придется делать достаточно редко, и, кроме того, мы можем использовать определенные механизмы для выявления структурных изменений, происходящих на более высоких уровнях с начала прохода [GRAEFE10].

B^{link}-деревья

B^{link}-деревья строятся на основе B*-деревьев (см. раздел «Перебалансировка» на с. 87) и дополнительно используют *высокие ключи* (см. подраздел «Высокие ключи узла» на с. 81) и указатели *одноуровневой связи* [LEHMAN81]. Высокий ключ определяет максимально возможный ключ поддерева. В B^{link}-дереве каждый узел, за исключением корня, имеет два указателя: указатель на потомка, спускающийся от родительского узла, и указатель одноуровневой связи, идущий от узла, расположенного на том же уровне слева.

B^{link}-деревья допускают состояние, называемое *полуразделением* (half-split), при котором на узел уже указывает указатель одноуровневой связи, но еще не указывает указатель на потомка от его родителя. Полуразделение выявляется путем сверки с высоким ключом узла. Если искомый ключ превышает высокий ключ узла (что нарушает инвариант высокого ключа), алгоритм поиска делает вывод о том, что структура была изменена параллельным процессом, и продолжает поиск, следуя по указателю одноуровневой связи.

При этом для обеспечения наилучшей производительности необходимо быстро добавить указатель в родительский узел, но это не требует прерывания и перезапуска процесса поиска, так как в дереве доступны все элементы. Преимущество здесь состоит в том, что нам не нужно удерживать родительскую блокировку при спуске на дочерний уровень, даже если потомок будет подвергаться разделению: мы можем сделать новый узел видимым с помощью его указателя одноуровневой связи и обновить родительский указатель в отложенном режиме без ущерба для коректности [GRAEFE10].

Хотя такой подход несколько менее эффективен по сравнению со спуском непосредственно от родителя и требует доступа к дополнительной странице, он обеспечивает корректный спуск от корня к листу, упрощая параллельный доступ. Поскольку операции разделения производятся достаточно редко и B-деревья редко сжимаются, этот случай является исключительным и требует незначительных затрат. Этот подход имеет целый ряд преимуществ: он уменьшает конкуренцию, исключает необходимость в удержании родительской блокировки во время разделения и уменьшает число

блокировок, удерживаемых во время модификации структуры дерева, до некоторого постоянного количества. Что еще более важно, он позволяет производить чтение одновременно с внесением в дерево структурных изменений и исключает возможность возникновения взаимоблокировок из-за параллельного внесения изменений, распространяющихся вверх к родительским узлам.

Итоги

В этой главе мы обсудили компоненты подсистемы хранения, отвечающие за обработку транзакций и восстановление. При реализации процесса обработки транзакций мы сталкиваемся с двумя проблемами:

- Чтобы повысить эффективность, мы должны разрешить одновременное выполнение транзакций.
- Чтобы сохранить корректность, мы должны гарантировать, что одновременно выполняемые транзакции не будут нарушать ACID-свойства.

Одновременное выполнение транзакций может вызывать различные виды аномалий чтения и записи. Наличие или отсутствие этих аномалий определяется и ограничивается путем реализации различных уровней изолированности. Подходы к управлению параллелизмом определяют, как планируются и выполняются транзакции.

Кэш страниц отвечает за сокращение числа обращений к диску: он кэширует страницы в памяти и предоставляет доступ к ним для чтения и записи. Когда кэш достигает предела емкости, страницы вытесняются из кэша и выгружаются обратно на диск. Чтобы гарантировать, что невыгруженные изменения не будут потеряны в случае сбоя узла, и сделать возможным откат транзакций, мы используем журналы упреждающей записи. Кэш страниц и журналы упреждающей записи координируются с помощью политик принуждения и воровства, которые призваны обеспечить эффективное выполнение и откат каждой транзакции без ущерба для долговечности.

Дополнительная литература

Если вы хотите узнать больше о концепциях, упомянутых в этой главе, вы можете обратиться к следующим источникам:

Общие сведения об обработке транзакций и восстановлении

Weikum, Gerhard, and Gottfried Vossen. 2001. Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. San Francisco: Morgan Kaufmann Publishers Inc.

Bernstein, Philip A. and Eric Newcomer. 2009. Principles of Transaction Processing. San Francisco: Morgan Kaufmann.

Graefe, Goetz, Guy, Wey & Sauer, Caetano. 2016. “Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, Media Restore, and System Failover,

(2nd Ed.)" in Synthesis Lectures on Data Management 8, 1–113. 10.2200/S00710ED2V01Y201603DTM044.

Mohan, C., Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging." Transactions on Database Systems 17, no. 1 (March): 94–162. https://doi.org/10.1145/128765.128770.

Управление параллелизмом в B-деревьях

Wang, Paul. 1991. "An In-Depth Analysis of Concurrent B-Tree Algorithms." MIT Technical Report. https://apps.dtic.mil/dtic/tr/fulltext/u2/a232287.pdf.

Goetz Graefe. 2010. A survey of B-tree locking techniques. https://dl.acm.org/doi/10.1145/1806907.1806908. 35, 3, Article 16 (July 2010), 26 pages.

Параллельные и конкурентные структуры данных

McKenney, Paul E. 2012. "Is Parallel Programming Hard, And, If So,What Can You Do About It?". https://arxiv.org/abs/1701.00854.

Herlihy, Maurice and Nir Shavit. 2012. The Art of Multiprocessor Programming, Revised Reprint (1st Ed.). San Francisco: Morgan Kaufmann.

Разработки в области обработки транзакций в хронологическом порядке

Diaconu, Cristian, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. "Hekaton: SQL Server's Memory-Optimized OLTP Engine." In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13), 1243–1254. New York: Association for Computing Machinery. https://doi.org/ 10.1145/2463676.2463710.

Kimura, Hideaki. 2015. "FOEDUS: OLTP Engine for a Thousand Cores and NVRAM." In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15), 691–706. https://doi.org/ 10.1145/2723372.2746480.

Yu, Xiangyao, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. "TicToc: Time Traveling Optimistic Concurrency Control." In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16), 1629–1642. https://doi.org/10.1145/2882903.2882935.

Kim, Kangnyeon, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. "ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads." In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16), 1675–1687. https://doi.org/10.1145/2882903.2882905.

Lim, Hyeontaek, Michael Kaminsky, and David G. Andersen. 2017. "Cicada: Dependably Fast Multi-Core In-Memory Transactions." In Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17), 21–35. https://doi.org/10.1145/3035918.3064015.

ГЛАВА 6

Варианты B-деревьев

У вариантов B-дерева есть ряд общих черт: структура в виде дерева, балансировка с помощью разбиений и слияний, алгоритмы поиска и удаления. Другие детали, связанные с параллелизмом, представлением страниц на диске, ссылками между одноуровневыми узлами и процессами обслуживания, могут различаться в разных реализациях.

В этой главе мы обсудим несколько методов, которые можно применить для реализации эффективных B-деревьев и структур, которые их используют:

- *B-деревья с копированием при записи* (copy on write b-tree) структурированы как B-деревья, но их узлы неизменяемы и не обновляются на месте. Вместо этого страницы копируются, обновляются и записываются в новые места.
- В *ленивых* B-деревьях (lazy btrees) уменьшается количество запросов ввода-вывода, связанных с последовательными операциями записи в одном узле, путем буферизации обновлений узлов. В следующей главе мы также рассмотрим двухкомпонентные LSM-деревья (см. подраздел «Двухкомпонентное LSM-дерево» на с. 151), в которых буферизация используется еще шире для реализации полностью неизменяемых B-деревьев.
- *FD-деревья* используют другой подход к буферизации, чем-то похожий на LSM-деревья (см. раздел «LSM-деревья» на с. 148). FD-деревья буферизуют обновления в небольшом B-дереве. После того как это дерево заполняется, его содержимое записывается в неизменяемый ряд. Обновления распространяются между уровнями неизменяемых рядов каскадным образом, от более высоких уровней к более низким.
- *Bw-деревья* разделяют узлы B-дерева на несколько меньших частей, запись в которые производится только в виде добавления. Это снижает затраты на небольшие операции записи за счет пакетирования обновлений различных узлов.
- *Кэш-независимые B-деревья* (cache oblivious btree) позволяют обрабатывать структуры данных на диске практически так же, как они обрабатываются в памяти.

Копирование при записи

Вместо того чтобы использовать сложные механизмы защелки, некоторые СУБД для обеспечения целостности данных при наличии параллельных операций используют метод *копирования при записи*. В этом случае перед каждым изменением страницы

ее содержимое копируется, вместо оригинала изменяется копия и создается параллельная иерархия дерева.

Прежние версии дерева остаются доступными для операций чтения, выполняющихся одновременно с операцией записи, в то время как операции записи, запрашивающие доступ к изменяемым страницам, должны ждать завершения предыдущих операций записи. После создания новой иерархии страниц указатель на самую верхнюю страницу атомарно обновляется. На рис. 6.1 показано, как параллельно изначальному дереву создается новое, в котором повторно используются незатронутые страницы.

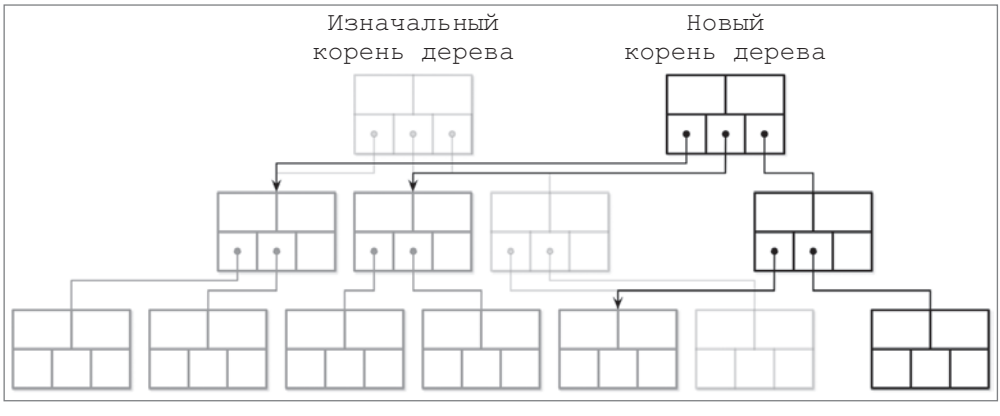

Рис. 6.1. B-деревья с копированием при записи

Очевидным недостатком этого подхода является то, что он требует больше места (даже несмотря на то, что прежние версии хранятся лишь в течение короткого промежутка времени, так как они могут быть освобождены сразу после завершения работающих с ними параллельных операций) и больше процессорного времени, поскольку необходимо копировать все содержимое страницы. Поскольку B-деревья, как правило, имеют небольшую глубину, простота и преимущества этого подхода зачастую перевешивают недостатки.

Самое большое преимущество этого подхода заключается в том, что операции чтения не требуют синхронизации, так как записанные страницы неизменяемы и к ним можно обращаться без установки дополнительной защелки. Поскольку операции записи работают с копиями страниц, операции чтения не блокируют операции записи. Ни одна из операций не увидит страницу в незавершенном состоянии, а системный сбой не оставит страницы в поврежденном состоянии, так как самый верхний указатель переключается только после завершения всех изменений страницы.

Реализация функции копирования при записи: LMDB

Примером подсистемы хранения, использующей копирование при записи, может служить система LMDB (Lightning Memory-Mapped Database, база данных с отобра-

жением в память), которая представляет собой хранилище типа «ключ—значение» и используется в проекте OpenLDAP. Благодаря своему устройству и архитектуре LMDB не нуждается в использовании кэша страниц, журнала упреждающей записи, контрольных точек или уплотнения[1].

Система LMDB реализована как одноуровневое хранилище данных, т. е. операции чтения и записи выполняются путем непосредственного отображения в память без дополнительного промежуточного кэширования на уровне приложения. Это также означает, что страницы не требуют дополнительного физического представления и считывание можно выполнять непосредственно из памяти без копирования данных в промежуточный буфер. Во время обновления каждый узел ветви на пути от корня до целевого листа копируется и при необходимости модифицируется: узлы, на которые распространяются обновления, изменяются, а остальные узлы остаются без изменений.

Система LMDB предусматривает только две версии (https://databass.dev/links/88) корневого узла: последнюю версию и версию, в которой фиксируются новые изменения. Этого достаточно, так как все операции записи должны проходить через корневой узел. После создания нового корня прежний корень становится недоступным для новых операций чтения и записи. После завершения операций чтения, обращающихся к старым разделам дерева, соответствующие страницы высвобождаются и могут использоваться повторно. Так как в системе LMDB данные доступны только для добавления, в ней не используются указатели на одноуровневые элементы, и во время последовательного сканирования приходится подниматься назад к родительскому узлу.

При таком подходе нецелесообразно оставлять старые данные в скопированных узлах: уже существует копия, которая может использоваться для многоверсионного управления конкурентным доступом выполнения текущих транзакций чтения. Сама структура этой базы данных является многоверсионной, и операции чтения могут выполняться без каких-либо блокировок, поскольку они никоим образом не мешают операциям записи.

Абстракции для управления обновлениями

Чтобы обновить страницу на диске, мы, так или иначе, должны сначала обновить ее представление в памяти. Однако существует несколько способов представления узла в памяти: мы можем получить доступ к кэшированной версии узла напрямую, сделать это через объект-обертку или создать представление узла в памяти, нативное для языка реализации.

В языках с неуправляемой моделью памяти можно переинтерпретировать хранящиеся в узлах B-дерева необработанные двоичные данные и использовать для работы с ними нативные указатели. В таком случае узел определяется в терминах структур,

[1] Дополнительные сведения о LMDB см. в комментариях (https://databass.dev/links/86) к коду и презентации (https://databass.dev/links/87).

использующих скрытые за указателем необработанные двоичные данные и операции приведения среды выполнения. Обычно они указывают на область памяти, управляемую кэшем страниц, или используют отображение в память.

Кроме того, узлы B-дерева могут быть представлены физически в виде нативных объектов или структур языка программирования. Эти структуры можно использовать для выполнения операций вставки, обновления и удаления. В ходе выгрузки на диск изменения применяются к страницам в памяти, а затем и на диске. Данный подход упрощает выполнение параллельных обращений за счет того, что модификация базовых необработанных страниц производится независимо от обращений к промежуточным объектам, но приводит к более высокому расходу памяти, так как в памяти приходится хранить две версии одной и той же страницы (необработанную двоичную версию и нативное представление языка).

Третий подход состоит в том, чтобы предоставлять доступ к буферу, поддерживающему узел через объект-обертку, обеспечивающий физическое представление изменений в B-дереве по мере их выполнения. Такой подход обычно используется в языках с управляемой моделью памяти. Объекты-обертки применяют изменения к поддерживающим буферам.

В силу независимого управления версиями страниц, размещенными на диске, в кэше и памяти, у них могут быть разные жизненные циклы. Например, мы можем буферизировать операции вставки, обновления и удаления и синхронизовать внесенные в память изменения с исходными дисковыми версиями во время операций чтения.

Ленивые B-деревья

Некоторые алгоритмы (в рамках этой книги мы будем называть их «ленивыми B-деревьями»[1]) снижают затраты на обновление B-дерева за счет буферизации обновлений и распространения их с задержкой с помощью легковесных резидентных структур, лучше подходящих для организации параллелизма и обновления данных.

WiredTiger

Давайте посмотрим, как мы можем использовать буферизацию для реализации ленивого B-дерева. Для этого мы можем создавать физическое представление узлов B-дерева в памяти по мере их загрузки и использовать эту структуру для сохранения обновлений вплоть до момента их выгрузки на диск.

Такой подход используется в WiredTiger (https://databass.dev/links/89) — дефолтной подсистеме хранения СУБД MongoDB. Ее реализация B-дерева с построчным

[1] Это название не является общепринятым, но, поскольку общим свойством обсуждаемых здесь вариантов B-дерева является то, что они буферизуют обновления B-дерева в промежуточных структурах вместо их непосредственного применения к дереву, мы будем использовать термин «ленивые», который довольно точно определяет это свойство.

хранением использует различные форматы для страниц в памяти и на диске. Перед персистентным сохранением размещенных в памяти страниц они должны пройти через процесс синхронизации.

Представленная на рис. 6.2 схема показывает, как выглядят страницы системы WiredTiger и как они объединяются в B-дерево. *Чистая* страница содержит только индекс, созданный изначально на основе дискового образа страницы. Обновления изначально сохраняются в *буфер обновления*.

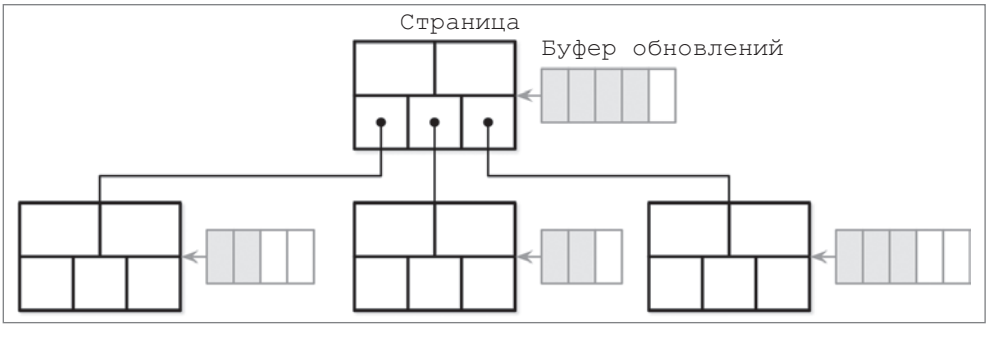

Рис. 6.2. Обобщенная схема системы WiredTiger

Доступ к буферам обновления осуществляется во время операций чтения: их содержимое объединяется с исходным содержимым дисковой версии страницы таким образом, чтобы обеспечивалось возвращение самых свежих данных. При выгрузке страницы содержимое буфера обновления синхронизуется с содержимым страницы и персистентно сохраняется на диске с перезаписью исходной страницы. Если размер синхронизованной страницы больше максимально допустимого, она разбивается на несколько страниц. Буферы обновления реализуются с помощью списков с пропусками, которые сравнимы по сложности с деревьями поиска [PAPADAKIS93], но имеют лучший профиль параллелизма [PUGH90a].

На рис. 6.3 показано, что и «чистые», и «грязные» страницы в WiredTiger имеют резидентные версии и ссылаются на базовый дисковый образ. «Грязные» страницы, помимо этого, еще имеют буфер обновления.

Основное преимущество здесь заключается в том, что обновления страниц и структурные модификации (разделения и слияния) выполняются фоновым потоком и процессы чтения/записи не должны ждать их завершения.

Ленивое адаптивное дерево

Вместо того чтобы буферизировать обновления отдельных узлов, мы можем сгруппировать узлы в поддеревья и добавить буфер обновления, обеспечивающий пакетирование операций *каждого поддерева*. В таком случае буферы обновления будут отслеживать все операции, выполняемые над верхним узлом поддерева и его по-

томками. Этот алгоритм называется «ленивое адаптивное дерево» (Lazy-Adaptive Tree, LA-дерево) [AGRAWAL09].

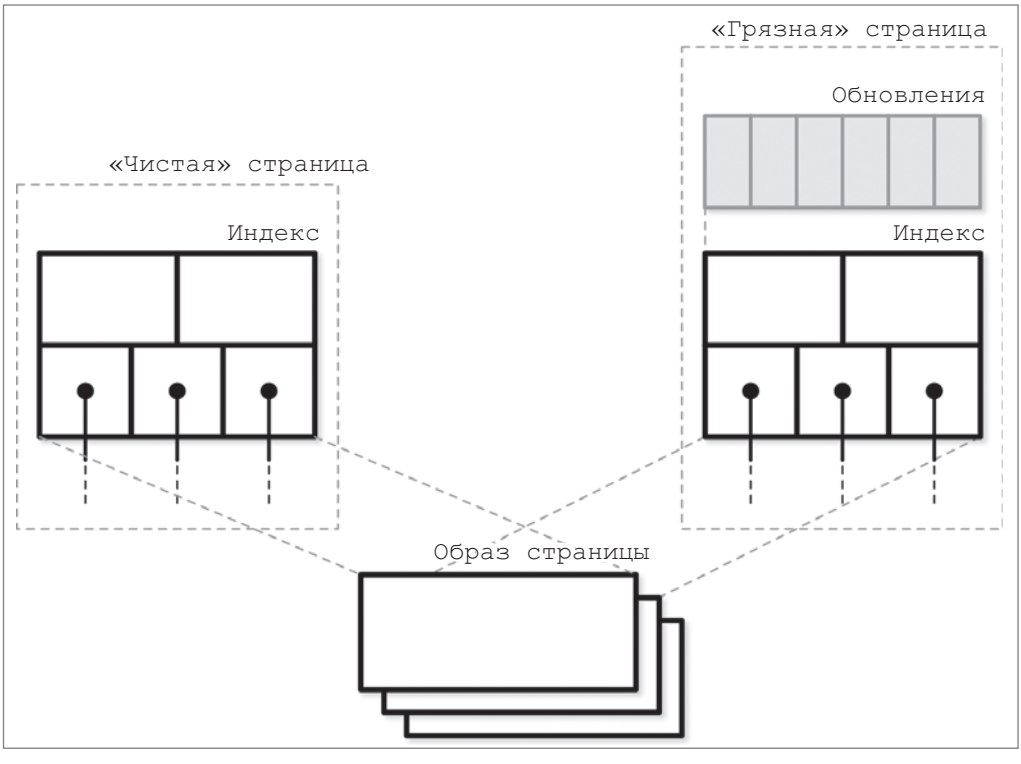

Рис. 6.3. Страницы системы WiredTiger

При вставке записи данных новая запись сначала добавляется в буфер обновления корневого узла. Когда этот буфер заполняется, он опустошается путем копирования и распространения изменений в буферы на более низких уровнях дерева. Если нижние уровни также заполняются, этот процесс может продолжаться рекурсивно, пока не достигнет листовых узлов.

На рис. 6.4 показано LA-дерево с каскадными буферами для узлов, сгруппированных в соответствующие поддеревья. Серым цветом выделены изменения, распространяющиеся от корневого буфера.

Буферы имеют иерархические зависимости и каскадируются: все обновления распространяются от буферов более высокого уровня к буферам более низкого уровня. Когда обновления достигают листового уровня, там выполняются пакетные операции вставки, обновления и удаления с одномоментным применением всех изменений к содержимому дерева и его структуре. Вместо выполнения последовательных обновлений на страницах по отдельности страницы можно обновить одномоментно, что

требует меньше обращений к диску и структурных изменений, поскольку разбиения и слияния распространяются на более высокие уровни пакетами.

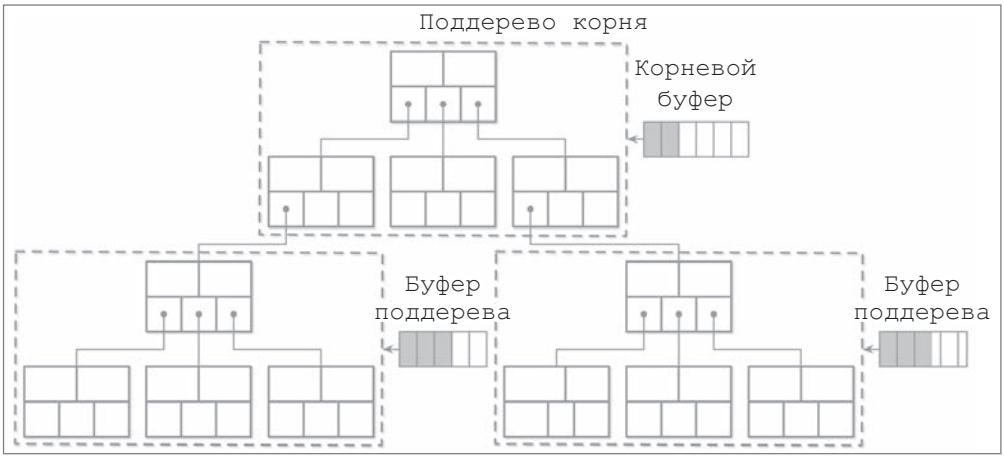

Рис. 6.4. LA-дерево

Описанные здесь методы буферизации оптимизируют время обновления дерева за счет пакетирования операций записи, делая это немного по-разному. При использовании обоих алгоритмов требуется выполнять дополнительный поиск в резидентных буферных структурах, а также слияние/согласование с размещенными на диске старыми данными.

FD-деревья

Буферизация является одной из тех идей, которые широко используются в хранилищах баз данных: она помогает обойтись без множества мелких операций произвольной записи, выполняя вместо этого одну более крупную операцию записи. На жестких дисках операции произвольной записи выполняются медленно из-за необходимости позиционировать головку. У твердотельных накопителей нет движущихся частей, однако повышенное количество связанных с записью операций ввода-вывода ведет к повышенным затратам на сборку мусора.

Обслуживание B-дерева требует множества операций произвольной записи, включая операции записи на листовом уровне и операции разбиения и слияния, распространяющиеся к родительским узлам, но что, если мы могли бы полностью избавиться от операций произвольной записи и обновления узлов?

До сих пор мы обсуждали буферизацию обновлений для отдельных узлов или групп узлов путем создания вспомогательных буферов. Альтернативный подход состоит в том, чтобы группировать обновления, предназначенные для разных узлов, используя доступное только для добавления хранилище и процессы слияния, — эта идея,

в частности, лежит в основе LSM-деревьев (см. раздел «LSM-деревья» на с. 148). Это означает, что при выполнении любой операции записи нам не нужно искать целевой узел: все обновления просто добавляются. Примером использования такого подхода к индексированию является *дерево флэш-диска* (FD-дерево) [LI10].

FD-дерево состоит из небольшого изменяемого *головного дерева* и нескольких неизменяемых отсортированных рядов. При таком подходе размер области, в которой требуется произвольный ввод-вывод для записи, ограничивается головным деревом — небольшим B-деревом, которое буферизует обновления. После того как головное дерево заполняется, его содержимое переносится в неизменяемый ряд. Если размер вновь записанного ряда превышает пороговое значение, его содержимое объединяется со следующим уровнем, с постепенным распространением записей данных с верхних уровней на нижние.

Частичное каскадирование

Для обслуживания указателей между уровнями FD-деревья используют метод *частичного каскадирования* (fractional cascading) [CHAZELLE86]. Этот подход позволяет снизить затраты на поиск элемента в каскаде отсортированных массивов: вы выполняете $\log n$ шагов, чтобы найти искомый элемент в первом массиве, но последующие операции поиска намного менее затратны, так как начинают поиск с ближайшего совпадения из предыдущего уровня.

Укороченные переходы между уровнями создаются путем создания перемычек между массивами соседних уровней, призванных минимизировать количество пробелов — групп элементов без указателей с более высоких уровней. Перемычки создаются путем вытягивания элементов с нижних уровней на верхние, если их там еще нет, с созданием указателя на позицию вытягиваемого элемента в массиве нижнего уровня.

Поскольку в источнике [CHAZELLE86] решается задача поиска в области вычислительной геометрии, он описывает двунаправленные перемычки и алгоритм восстановления инварианта размера пробела, который мы не будем подробно рассматривать здесь. Мы изучим лишь те моменты, которые применимы к хранилищам баз данных в целом и FD-деревьям в частности.

Мы могли бы отобразить каждый элемент массива более высокого уровня на ближайший элемент следующего уровня, но это привело бы к слишком большим затратам на указатели и их обслуживание. Если мы будем отображать только те элементы, которые уже существуют на более высоком уровне, это может привести к тому, что пробелы между элементами будут слишком большими. Чтобы решить эту проблему, мы будем вытягивать из массива нижнего уровня на более высокий уровень каждый N-й элемент.

Например, допустим, что у нас есть несколько отсортированных массивов:

```
A1 = [12, 24, 32, 34, 39]
A2 = [22, 25, 28, 30, 35]
A3 = [11, 16, 24, 26, 30]
```

Чтобы упростить поиск, мы можем заполнить пробелы между элементами, вытягивая в массив с более низким индексом каждый второй элемент массива с более высоким индексом:

```
A1 = [12, 24, 25, 30, 32, 34, 39]
A2 = [16, 22, 25, 26, 28, 30, 35]
A3 = [11, 16, 24, 26, 30]
```

Теперь мы можем использовать эти вытянутые элементы для создания «перемычек» (или «ограждений» (fences), как их называют в статье, посвященной FD-деревьям): указателей от элементов более высокого уровня к их двойникам на более низких уровнях, как показано на рис. 6.5.

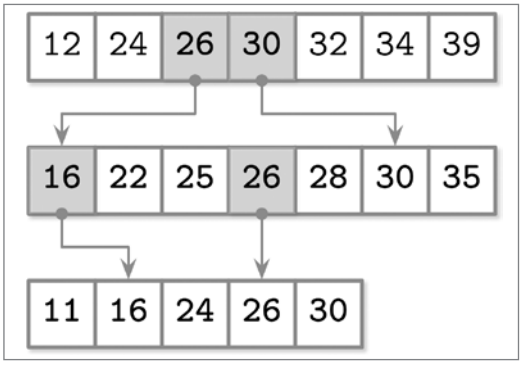

Рис. 6.5. Частичное каскадирование

Для поиска элементов во всех этих массивах мы выполняем двоичный поиск на самом высоком уровне, что значительно сокращает пространство поиска на следующем уровне, так как теперь мы переходим по перемычке примерно туда, где находится искомый элемент. Это позволяет нам соединить несколько отсортированных рядов и снизить затраты на поиск по ним.

Ряды логарифмического размера

FD-дерево сочетает частичное каскадирование с созданием *отсортированных рядов логарифмического размера* — неизменяемых отсортированных массивов с размером, возрастающим в k раз, создаваемых путем слияния предыдущего уровня с текущим.

Ряд самого высокого уровня создается после заполнения головного дерева: содержимое его листов записывается на первый уровень. После того как головное дерево заполняется снова, его содержимое сливается с элементами первого уровня. Результат слияния замещает прежнюю версию первого ряда. Ряды более низкого уровня создаются, когда размеры рядов более высокого уровня достигают порогового значения. Если ряд более низкого уровня уже существует, он замещается результатом слияния его содержимого с содержимым более высокого уровня. Этот процесс очень похож

на уплотнение в LSM-деревьях, при котором неизменяемое содержимое таблиц подвергается слиянию с получением таблиц большего размера.

На рис. 6.6 представлено схематическое изображение FD-дерева с головным B-деревом сверху, двумя рядами логарифмического размера L1 и L2 и перемычками между ними.

Для сохранения адресуемости всех элементов отсортированных рядов в FD-деревьях используют адаптированную версию частичного каскадирования, в которой головные элементы страниц более низкого уровня распространяются как указатели на более высокие уровни. Использование этих указателей снижает затраты на поиск в деревьях более низкого уровня, поскольку он оказывается уже частично выполненным на более высоком уровне, что позволяет продолжить его с наиболее близкого совпадения.

Рис. 6.6. Схематическое изображение FD-дерева

Поскольку FD-деревья не обновляют страницы на месте и может случиться так, что записи данных для одного и того же ключа присутствуют на нескольких уровнях, удаление в FD-деревьях производят путем вставки отметок полного удаления (в статье о FD-деревьях они называются *элементами фильтрации,* filter entries), которые указывают, что запись данных, связанная с соответствующим ключом, помечена для удаления и все записи данных для этого ключа на нижних уровнях должны быть удалены. После распространения отметки полного удаления до самого нижнего уровня ее можно удалить, поскольку при этом уже не остается элементов, которые она может затенить.

Bw-деревья

Увеличение объема записи является одной из наиболее существенных проблем при реализации обновления на месте в B-деревьях: последовательные обновления страницы могут потребовать обновления копии страницы, находящейся на диске, при каждом обновлении. Вторая проблема — увеличение пространства: мы резервируем дополнительное пространство для обновления. Это также означает, что для каждого передаваемого полезного байта, несущего запрошенные данные, мы должны передать несколько пустых байтов и остальную часть страницы. Третья проблема — сложность решения задач параллелизма и работы с защелками.

Чтобы решить все три проблемы сразу, мы должны использовать подход, совершенно отличный от тех, которые мы до сих пор обсуждали. Буферизация обновлений помогает с увеличением объема записи и пространства, но не предлагает решения проблем параллелизма.

Мы можем пакетно обновлять различные узлы, используя хранилище с доступом только для добавления, связывать узлы в цепочки и использовать резидентную структуру данных, позволяющую устанавливать указатели между узлами с помощью одной операции «Сравнение с обменом», что позволит не использовать блокировку при реализации дерева. Такой подход называется *Buzzword*-деревом (Bw-деревом) [LEVANDOSKI14].

Цепочки обновлений

Bw-дерево записывает базовый узел отдельно от его модификаций. Модификации («дельта-узлы») образуют цепочку: связанный список от самой новой модификации к более старым с базовым узлом в конце. Каждое обновление может храниться отдельно, без необходимости перезаписывать существующий узел на диске. Дельта-узлы могут представлять вставки, обновления (которые неотличимы от вставок) или удаления.

Поскольку размеры базового узла и дельта-узлов вряд ли будут выровнены по страницам, имеет смысл хранить их последовательно, и поскольку ни базовый узел, ни дельта-узлы не изменяются во время обновления (все модификации просто добавляют узел к существующему связанному списку), нам не нужно резервировать дополнительное пространство.

Наличие узла как логической, а не физической сущности представляет собой интересное изменение парадигмы: нам не нужно предварительно выделять пространство, требовать, чтобы узлы имели фиксированный размер, или даже держать их в смежных сегментах памяти. Это, безусловно, имеет обратную сторону: во время чтения все дельты должны быть пройдены и применены к базовому узлу, чтобы воспроизвести реальное состояние узла. Это несколько похоже на то, что делается в LA-деревьях (см. подраздел «Ленивое адаптивное дерево» на с. 133): сохранение обновлений отдельно от основной структуры и их воспроизведение при чтении.

Укрощение параллелизма с помощью операции «Сравнение с обменом»

Было бы довольно затратно поддерживать на диске структуру дерева, позволяющую добавлять элементы к дочерним узлам: это потребовало бы от нас постоянного обновления родительских узлов добавлением указателей на самую новейшую дельту. Вот почему узлы Bw-дерева, состоящие из цепочки дельт и базового узла, имеют логические идентификаторы и используют резидентную таблицу отображения идентификаторов на их позиции на диске. Использование отображения также позволяет избавиться от защелок: вместо получения эксклюзивного права доступа во время

записи в Bw-деревьях применяют операции «Сравнение с обменом» к физическим смещениям в таблице отображения.

Рисунок 6.7 показывает простое Bw-дерево. Каждый логический узел состоит из одного базового узла и нескольких связанных дельта-узлов.

Чтобы обновить узел Bw-дерева, алгоритм выполняет следующие действия:

1. Целевой логический листовой узел находится путем обхода дерева от корня к листу. Таблица отображения содержит виртуальные ссылки на целевые базовые узлы или последние дельта-узлы в цепочке обновлений.
2. Создается новый дельта-узел с указателем на базовый узел (или на последний дельта-узел), найденный в ходе шага 1.
3. Обновляется таблица отображения с добавлением в нее указателя на новый дельта-узел, созданный в ходе шага 2.

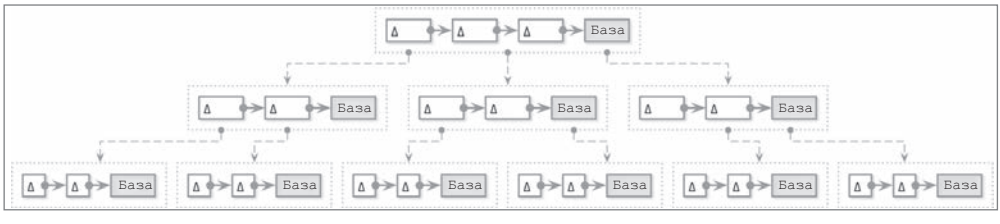

Рис. 6.7. Bw-дерево. Пунктирные линии представляют виртуальные связи между узлами, заданные с помощью таблицы отображения. Сплошные линии представляют собой указатели реальных данных между узлами

Операция обновления во время шага 3 может быть выполнена с помощью операции «Сравнение с обменом», которая является атомарной, поэтому все чтения, параллельные обновлению указателя, разделяются на операции, происходящие либо до, либо после записи, без блокировки операций чтения и записи. Операции чтения, происходящие до обновления, используют прежний указатель и не видят новый дельта-узел, так как он еще не был задан. Операции чтения, происходящие после обновления, используют новый указатель и видят обновление. Если два потока пытаются установить новый дельта-узел на один и тот же логический узел, то только один из них сможет успешно выполнить эту операцию, а другому придется предпринять повторную попытку выполнения.

Операции модификации структуры

Bw-дерево логически структурировано подобно B-дереву, что означает, что его узлы тоже иногда становятся слишком большими или почти пустыми (происходит, соответственно, переполнение или потеря значимости) или требуют таких операций модификации структуры (structure modification operations), как разделение и слияние. Семантика разделений и слияний здесь аналогична семантике в B-деревьях (см. подразделы «Разделение узлов B-дерева» на с. 57 и «Слияние узлов B-дерева» на с. 59), но реализация различается.

Операции разделения начинаются с консолидации логического содержимого разделяемого узла, применения дельт к его базовому узлу и создания новой страницы, содержащей элементы справа от точки разделения. После этого процесс продолжается в два этапа [WANG18]:

1. *Разделение*: специальный *дельта-узел разделения* добавляется к разделяемому узлу, чтобы уведомить операции чтения о продолжающемся разделении. Дельта-узел разделения содержит ключ — разделитель средней точки для аннулирования записей в разделяемом узле и ссылку на новый логический одноуровневый узел.
2. *Обновление родительского узла*: на этом этапе ситуация аналогична ситуации с *полуразделением* B^{link}-дерева (см. подраздел «B^{link}-деревья» на с. 126), поскольку узел доступен через указатель дельта-узла разделения, но на него еще нет ссылки от родительского узла, и операции чтения должны пройти через старый узел, а затем пройти по указателю одноуровневого элемента, чтобы достичь вновь созданного одноуровневого узла. Новый узел добавляется как дочерний к родительскому узлу, так что операции чтения могут добраться до него непосредственно, а не перенаправляться через разделяющийся узел. Разделение завершается.

Обновление указателя родительского узла необходимо для оптимизации производительности: все узлы и их элементы остаются доступными, даже если указатель родительского узла не обновляется. Bw-деревья не содержат защелок, поэтому любой поток может столкнуться с неполным выполнением операции модификации структуры. Поток должен принять эстафету и закончить операцию структурного изменения, прежде чем продолжить свое выполнение. Следующий поток будет идти по установленному указателю родительского узла и не должен будет проходить через указатель одноуровневого узла.

Операции слияния работают аналогичным образом:

1. *Удаление одноуровневого узла*: создается специальный *дельта-узел удаления* и добавляется к правому одноуровневому узлу, тем самым обозначается начало операции слияния и отмечается для удаления правый одноуровневый узел.
2. *Слияние*: *дельта-узел слияния* создается в левом одноуровневом узле, чтобы указать на содержимое правого одноуровневого узла и сделать его логической частью левого одноуровневого узла.
3. *Обновление родительского узла*: в этот момент содержимое правого одноуровневого узла доступно из левого. Чтобы завершить процесс слияния, необходимо удалить из родительского узла ссылку на правый одноуровневый узел.

Параллельные операции модификации структуры требуют создания в родительском узле дополнительного *дельта-узла прерывания*, чтобы исключить одновременное выполнение операций разделения и слияния [WANG18]. Дельта-узел прерывания работает аналогично блокировке записи: в каждый момент только один поток может иметь доступ на запись и любой поток, который пытается добавить новую запись к этому дельта-узлу, будет прерван. По завершении операции модификации структуры дельта-узел прерывания можно удалить из родительского элемента.

Высота Bw-дерева возрастает во время разделения корневого узла. Когда корневой узел становится слишком большим, он разделяется надвое и вместо прежнего создается новый корень, причем прежний корень и вновь созданный одноуровневый узел становятся его дочерними элементами.

Консолидация и сборка мусора

Цепочки дельта-узлов сами по себе могут разрастаться до любой длины. Поскольку операции чтения становятся более затратными по мере удлинения цепочек, мы должны стараться держать длину в разумных пределах. При достижении настраиваемого порога мы перестраиваем узел, объединяя содержимое базового узла со всеми дельтами, консолидируя их в один новый базовый узел. Затем новый узел записывается в новое место на диске и обновляется указатель на узел в таблице отображения. Мы рассмотрим этот процесс более подробно в разделе «LLAMA и тщательное многоуровневое совмещение» на с. 180, поскольку базовое журналированное хранилище отвечает за сборку мусора, консолидацию узлов и перемещение.

После консолидации узла его прежнее содержимое (базовый узел и все дельта-узлы) больше не адресуется из таблицы отображения. Однако мы не можем сразу освободить занимаемую ими память, так как часть содержимого может все еще использоваться текущими операциями. Поскольку нет защелок, удерживаемых операциями чтения (им не нужно было проходить через какой-либо барьер или регистрироваться где-либо, чтобы получить доступ к узлу), нам необходимо найти другие средства для отслеживания актуальных страниц.

Чтобы отделить потоки, которые могли столкнуться с определенным узлом, от тех, которые не могли его увидеть, в *Bw*-деревьях используют метод, известный как *восстановление на основе эпох*. Если некоторые узлы и дельты удаляются из таблицы отображения из-за консолидации, которая заменила их в течение некоторой эпохи, исходные узлы сохраняются до завершения каждой операции чтения, начатой в ту же эпоху или ранее. После этого их можно безопасно удалить при сборке мусора, так как более поздние операции чтения гарантированно никогда не видели эти узлы, поскольку они не были адресуемы ко времени запуска этих операций.

Bw-дерево — это интересный вариант B-дерева, улучшающий структуру по нескольким важным параметрам: увеличению объема записи, неблокирующему доступу и удобству кэширования. Модифицированная версия была реализована в экспериментальной подсистеме хранения Sled (https://databass.dev/links/90). Группа по базам данных Университета Карнеги — Меллона разработала резидентную версию Bw-дерева под названием *OpenBw-дерево* (https://databass.dev/links/91) и выпустила практическое руководство по реализации [WANG18].

В этой главе мы коснулись только самых общих концепций Bw-деревьев, связанных с B-деревьями, и продолжим их обсуждение в разделе «LLAMA и тщательное многоуровневое совмещение» на с. 180, включая обсуждение базового журналированного хранилища.

Кэш-независимые B-деревья

Размер блока, размер узла, выравнивание строк кэша и другие настраиваемые параметры влияют на производительность B-дерева. Новый класс структур данных под названием *«кэш-независимые структуры»* (cache oblivious data structures) [DEMAINE02] обеспечивает асимптотически оптимальную производительность независимо от базовой иерархии памяти и без необходимости в настройке этих параметров. Это означает, что алгоритму не нужно знать размеры строк кэша, блоков файловой системы и страниц диска. Кэш-независимые структуры разработаны так, чтобы хорошо работать на нескольких машинах с различными конфигурациями без необходимости модификаций.

До сих пор мы в основном рассматривали B-деревья исходя из двухуровневой иерархии памяти (за исключением системы LMDB, описанной в разделе «Копирование при записи» на с. 129). Узлы B-дерева хранятся в страницах на диске, а кэш страниц используется для обеспечения эффективного доступа к ним в оперативной памяти.

Двумя уровнями этой иерархии являются *кэш страниц* (который быстрее, но ограничен в объеме) и диск (который обычно медленнее, но имеет большую емкость) [AGGARWAL88]. Здесь у нас есть только два параметра, что делает довольно простым проектирование алгоритмов, поскольку лишь для двух уровней придется разработать специфичные модули кода, которые отвечают за всю специфику каждого из этих уровней.

Диск разбивается на блоки, и данные передаются между диском и кэшем поблочно: даже если алгоритм должен найти один элемент в блоке, необходимо загрузить весь блок. Такой подход является *кэш-зависимым*.

При разработке программного обеспечения, критичного к производительности, мы часто программируем исходя из более сложной модели, принимая во внимание кэши ЦП, а иногда даже иерархии дисков (как, например, в случае использования горячего/холодного хранилища, построения иерархии жесткий диск / твердотельный накопитель / хранилище с энергонезависимой памятью или поэтапного переключения данных с одного уровня на другой). В большинстве случаев такой код трудно обобщить. В разделе «Резидентные и дисковые СУБД» на с. 27 мы говорили, что доступ к диску на несколько порядков медленнее, чем к оперативной памяти, что побудило разработчиков баз данных приспособиться к этой разнице.

Кэш-независимые алгоритмы позволяют рассматривать структуры данных в терминах двухуровневой модели памяти, обеспечивая при этом преимущества многоуровневой иерархической модели. Такой подход позволяет отказаться от специфичных для платформы параметров, но гарантирует, что количество переходов между двумя уровнями иерархии находится в пределах постоянного показателя. Если структура данных оптимизирована так, чтобы оптимально работать для любых двух уровней иерархии памяти, она также оптимально работает для двух соседних уровней иерархии. Это достигается благодаря работе на максимально возможном высоком уровне кэша.

Структура ван Эмде Боаса

Кэш-независимое B-дерево состоит из статического B-дерева и структуры упакованного массива [BENDER05]. Статическое B-дерево строится с использованием структуры ван Эмде Боаса. Согласно этой структуре, дерево разделяется посередине от крайних значений. Затем каждое поддерево рекурсивно разделяется аналогичным образом, в результате чего получаются поддеревья размером `sqr(N)`. Ключевая идея этой структуры заключается в том, что любое рекурсивное дерево хранится в непрерывном блоке памяти.

На рис. 6.8 приведен пример структуры ван Эмде Боаса. Узлы, логически сгруппированные вместе, располагаются близко друг к другу. Сверху вы можете увидеть логическое представление структуры (т. е. как узлы образуют дерево), а снизу — как узлы дерева располагаются в памяти и на диске.

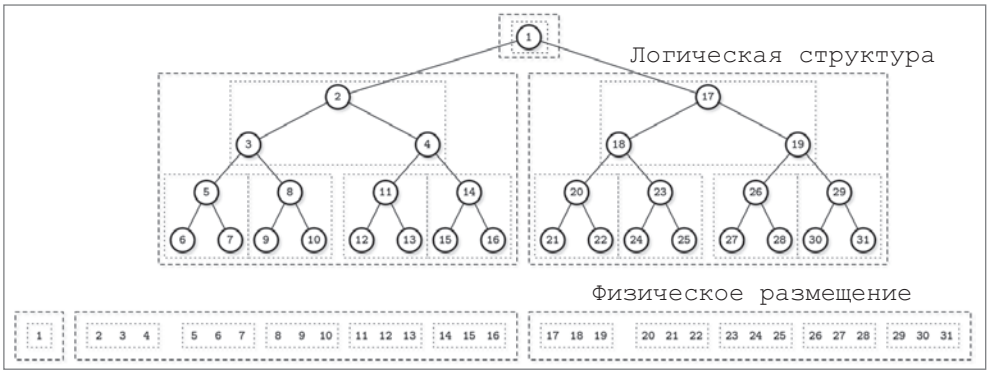

Рис. 6.8. Структура ван Эмде Боаса

Чтобы сделать структуру данных динамической (т. е. разрешить вставки, обновления и удаления), в кэш-независимых деревьях используют структуру данных «Упакованный массив», которая использует смежные сегменты памяти для хранения элементов, но содержит пробелы, зарезервированные для будущих вставок. Пробелы оставляют исходя из *порога плотности*. На рис. 6.9 показана структура упакованного массива, в котором элементы разнесены для создания пробелов.

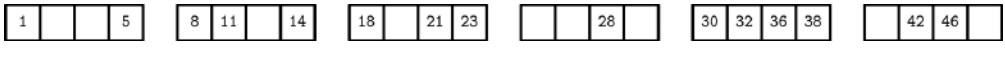

Рис. 6.9. Упакованный массив

Такой подход позволяет вставлять элементы в дерево с меньшим количеством перемещений. Элементы перемещают только для создания пробела для вновь вставляемого элемента, если пробел еще не существует. Когда упакованный массив становится слишком заполненным или малозаполненным, структуру необходимо перестроить, чтобы увеличить или уменьшить массив.

Статическое дерево используется в качестве индекса для упакованного массива нижнего уровня и должно обновляться в соответствии с перемещаемыми элементами, чтобы указывать на корректные элементы на нижнем уровне.

Этот подход интересен, и его идеи можно использовать для построения эффективных реализаций B-дерева. Он позволяет строить структуры на диске способами, очень похожими на то, как строятся структуры в оперативной памяти. Однако на момент написания книги мне не было известно о каких-либо неакадемических реализациях кэш-независимого B-дерева.

Возможной причиной этого является предположение, что когда загрузка кэша абстрагируется, а данные загружаются и записываются обратно поблочно, загрузка в кэш и вытеснение из него все равно оказывают негативное влияние на результат. Другая возможная причина заключается в том, что с точки зрения передачи блоков сложность кэш-независимых B-деревьев такая же, как и у кэш-зависимых. Такое положение дел может измениться, когда более широкое распространение получат более эффективные энергонезависимые запоминающие устройства с байтовой адресацией.

Итоги

Исходная схема B-дерева имеет несколько узких мест, которые часто не представляют проблем при использовании вращающихся дисков, но делают его менее эффективным при использовании твердотельных накопителей. B-деревья отличаются высокой степенью *увеличения объема записи* (в силу необходимости перезаписывать страницы) и *расхода пространства* (в силу необходимости резервировать пространство в узлах для будущих операций записи).

Степень увеличения объема записи можно снизить с помощью *буферизации*. Ленивые B-деревья, такие как деревья системы WiredTiger и LA-деревья, присоединяют резидентные буферы к отдельным узлам или группам узлов, чтобы уменьшить количество требуемых операций ввода-вывода путем буферизации в памяти последовательных обновлений страниц.

Чтобы снизить степень увеличения пространства, в FD-деревьях используют *неизменяемость*: записи данных сохраняются в неизменяемых отсортированных рядах, а размер изменяемого B-дерева ограничивается.

В Bw-деревьях проблему увеличения пространства решают, также используя неизменяемость. Узлы B-дерева и обновления к ним хранятся в отдельных местах на диске и сохраняются в журналированном хранилище. Степень увеличения объема записи снижается по сравнению с исходной схемой B-дерева, так как синхронизация содержимого, принадлежащего одному логическому узлу, происходит относительно редко. Для Bw-деревьев не требуются защелки для защиты страниц от параллельных обращений, так как виртуальные указатели между логическими узлами хранятся в памяти.

Дополнительная литература

Если вы хотите узнать больше о концепциях, упомянутых в этой главе, вы можете обратиться к следующим источникам:

B-деревья с копированием при записи

Driscoll, J. R., N. Sarnak, D. D. Sleator, and R. E. Tarjan. 1986. "Making data structures persistent." In Proceedings of the eighteenth annual ACM symposium on Theory of computing (STOC '86), 109–121. https://dx.doi.org/ 10.1016/0022-0000(89)90034-2.

Ленивые адаптивные деревья

Agrawal, Devesh, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh. 2009. "Lazy-Adaptive Tree: an optimized index structure for flash devices." Proceedings of the VLDB Endowment 2, no. 1 (January): 361–372. https://doi.org/10.14778/1687627.1687669.

FD-деревья

Li, Yinan, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. 2010. "Tree Indexing on Solid State Drives." Proceedings of the VLDB Endowment 3, no. 1–2 (September): 1195–1206. https://doi.org/10.14778/1920841.1920990.

Bw-деревья

Wang, Ziqi, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. "Building a Bw-Tree Takes More Than Just Buzz Words." Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18), 473–488. https://doi.org/ 10.1145/3183713.3196895

Levandoski, Justin J., David B. Lomet, and Sudipta Sengupta. 2013. "The Bw-Tree: A B-tree for new hardware platforms." In Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE '13), 302–313. IEEE. https:// doi.org/10.1109/ICDE.2013.6544834.

Кэш-независимые B-деревья

Bender, Michael A., Erik D. Demaine, and Martin Farach-Colton. 2005. "Cache-Oblivious B-Trees." SIAM Journal on Computing 35, no. 2 (August): 341–358. https://doi.org/10.1137/S0097539701389956.

ГЛАВА 7

Журналированное хранилище

Бухгалтеры не пользуются ластиками, иначе попадут в тюрьму.

Пэт Хелланд

Если бухгалтерам нужно изменить запись, вместо того чтобы стереть существующее значение, они создают новую запись с поправкой. При публикации квартальный отчет может содержать незначительные изменения, корректирующие результаты предыдущего квартала. Чтобы подвести общий итог, вы должны пройти через записи и вычислить промежуточные суммы [HELLAND15].

Точно так же неизменяемые структуры хранения не позволяют вносить изменения в существующие файлы: таблицы записываются один раз и больше никогда не изменяются. Вместо этого к новому файлу добавляются новые записи, и, чтобы найти конечное значение (или сделать вывод об его отсутствии), записи необходимо восстановить из нескольких файлов. И наоборот, изменяемые структуры хранения изменяют записи на диске на месте.

Неизменяемые структуры данных часто используются в функциональных языках программирования и становятся все более популярными из-за их характеристик безопасности: после создания неизменяемая структура не изменяется, возможен параллельный доступ ко всем ее ссылкам, а целостность гарантируется тем, что ее нельзя изменить.

На высоком уровне существует строгое различие между тем, как данные обрабатываются внутри структуры хранения и как — вне ее. Внутри неизменяемые файлы могут содержать несколько копий, более поздние из которых перезаписывают более старые, в то время как изменяемые файлы обычно содержат только самое последнее значение. При обращении к неизменяемым файлам они обрабатываются, избыточные копии согласуются, а самые последние возвращаются клиенту.

Как и в других книгах и статьях по этой теме, мы используем B-деревья в качестве типичного примера изменяемой структуры, а *журналированные деревья слияния* (LSM-деревья) — в качестве примера неизменяемой структуры. В неизменяемых LSM-деревьях используют хранилище, доступное только для добавления, и согласование слиянием, а в B-деревьях размещают записи данных на диске и обновляют страницы по их первоначальным смещениям в файле.

Структуры хранения с обновлением на месте оптимизированы для повышения производительности чтения [GRAEFE04]: запись может быть возвращена клиенту сразу

после нахождения ее данных на диске. За это приходится платить снижением производительности записи: чтобы обновить запись данных на месте, сначала нужно найти ее на диске. С другой стороны, хранилище, доступное только для добавления, оптимизировано для повышения производительности записи. Операциям записи не нужно искать записи на диске, чтобы перезаписать их. Однако за это приходится платить снижением производительности операций чтения, которые должны извлекать несколько версий записей данных и согласовывать их.

До сих пор мы в основном говорили об изменяемых структурах хранения. Мы затронули тему неизменяемости, обсуждая B-деревья с копированием при записи (см. раздел «Копирование при записи» на с. 129), FD-деревья (см. раздел «FD-деревья» на с. 135) и Bw-деревья (см. раздел «Bw-деревья» на с. 138). Но есть и другие способы реализации неизменяемых структур.

Из-за особенностей структуры и способа построения изменяемых B-деревьев большинство операций ввода-вывода во время чтения, записи и обслуживания носит *произвольный характер*. Каждая операция записи сначала должна найти страницу, содержащую запись данных, и лишь после этого может ее модифицировать. B-деревья нуждаются в операциях разделения и слияния узлов, которые меняют положение уже записанных записей. По прошествии некоторого времени страницы B-дерева обычно нуждаются в обслуживании. Страницы фиксированы по размеру, и некоторое свободное пространство зарезервировано для будущих записей. Другая проблема заключается в том, что при изменении только одной ячейки страницы необходимо заново перезаписать всю страницу.

Существуют альтернативные подходы, которые могут помочь смягчить эти проблемы, сделать некоторые операции ввода-вывода последовательными и избежать перезаписи страниц во время изменений. Один из таких подходов — использовать неизменяемые структуры. В этой главе мы сосредоточимся на LSM-деревьях: как они построены, каковы их свойства и чем они отличаются от B-деревьев.

LSM-деревья

Говоря о B-деревьях, мы пришли к выводу, что показатели расхода пространства и увеличения объема записи можно улучшить с помощью буферизации. Как правило, существуют два способа применения буферизации в различных структурах хранения: отложить распространение записи на страницы, находящиеся на диске (как мы видели в разделе «FD-деревья» на с. 135 и подразделе «WiredTiger» на с. 132), и сделать операции записи последовательными.

В одной из самых популярных разновидностей дисковых неизменяемых структур хранения, LSM-деревьях, для обеспечения последовательной записи используются буферизация и хранилище, доступное только для добавления. LSM-дерево — это вариант дисковой структуры, подобной B-дереву, где узлы полностью заняты и оптимизированы для последовательного доступа к диску. Впервые эта концепция была

представлена в работе Патрика О'Нила и Эдварда Ченга [ONEIL96]. Журналированные деревья слияния (LSM-деревья) берут свое название от журналированных файловых систем, которые записывают все модификации на диск в журналоподобный файл [ROSENBLUM92].

LSM-деревья записывают неизменяемые файлы и со временем объединяют их. Эти файлы обычно содержат собственный индекс, чтобы помочь операциям чтения эффективно находить данные. Несмотря на то что LSM-деревья часто представляют как альтернативу B-деревьям, B-деревья широко используются в качестве внутренней структуры индексации для неизменяемых файлов LSM-деревьев.

Слово «слияние» (merge) в LSM-деревьях указывает на то, что из-за их неизменяемости содержимое дерева объединяется с использованием подхода, аналогичного сортировке слияния. Этот процесс производится во время обслуживания для высвобождения места, занятого избыточными копиями, и во время чтения перед возвращением содержимого пользователю.

В LSM-деревьях запись файлов данных откладывается, а изменения буферизуются в таблице в памяти. Эти изменения затем распространяются путем записи их содержимого в неизменяемые файлы на диске. Все записи данных остаются доступными в памяти до тех пор, пока файлы не будут полностью сохранены на персистентном носителе.

Неизменяемость файлов данных упрощает выполнение последовательных операций записи: данные записываются на диск за один проход, а файлы доступны только для добавления. В изменяемых структурах в ходе одного прохода могут предварительно выделяться блоки (как, например, при использовании индексно-последовательного метода доступа (indexed sequential access method, ISAM) [RAMAKRISHNAN03] [LARSON81]), но последовательные обращения все равно потребуют произвольных операций чтения и записи. Неизменяемые структуры позволяют нам размещать записи данных последовательно, чтобы предотвратить фрагментацию. Кроме того, неизменяемые файлы обладают более высокой *плотностью*: мы не оставляем дополнительное пространство для записей данных, которые будут записаны позже, или для случаев, когда обновленные записи требуют больше места, чем первоначально записанные.

Поскольку файлы являются неизменяемыми, операции вставки, обновления и удаления не требуют поиска записей данных на диске, что значительно повышает производительность записи и пропускную способность. Вместо этого допускается дублирование содержимого, а конфликты разрешаются во время чтения. LSM-деревья особенно полезны для приложений, где операции записи гораздо более распространены по сравнению с операциями чтения, что часто имеет место в современных системах с интенсивным использованием данных, учитывая постоянно растущие объемы данных и скорость их приема.

Чтение и запись не пересекаются по определению, поэтому данные на диске можно читать и записывать без блокировки сегментов, что значительно упрощает параллельный доступ. Для сравнения, изменяемые структуры используют иерархические блокировки и защелки (дополнительную информацию о блокировках и защелках можно найти в разделе «Управление параллелизмом» на с. 111), чтобы обеспечить целостность дисковой структуры данных. Такие структуры допускают несколько одновременных операций чтения, но для записи требуется эксклюзивное право доступа к поддереву. Подсистемы хранения на основе LSM-деревьев используют линеаризуемые представления данных и индексных файлов в памяти, и для них необходимо лишь организовать параллельный доступ к структурам, управляющим данными.

Как B-деревья, так и LSM-деревья требуют некоторых вспомогательных действий для оптимизации производительности, но по разным причинам. Поскольку число выделенных файлов постоянно растет, LSM-деревья должны объединять и перезаписывать файлы, чтобы во время чтения производился доступ к минимально возможному числу файлов, поскольку запрошенные записи данных могут оказаться распределенными по нескольким файлам. С другой стороны, изменяемые файлы можно частично или полностью перезаписывать, чтобы уменьшать степень фрагментации и высвобождать место, занимаемое обновленными или удаленными записями. Конечно, точный объем работы, выполняемой вспомогательным процессом, в значительной степени зависит от конкретной реализации.

Структура LSM-дерева

Мы начнем с упорядоченных LSM-деревьев [ONEIL96], в которых файлы содержат отсортированные записи данных. Позже, в разделе «Неупорядоченное LSM-хранилище» на с. 171, мы также обсудим структуры, позволяющие хранить записи данных в порядке вставки, что имеет некоторые очевидные преимущества при прохождении пути записи.

Как мы только что обсуждали, LSM-деревья состоят из малых компонентов, расположенных в памяти, и больших компонентов, расположенных на диске. Чтобы записать неизменяемое содержимое файла на диск, необходимо сначала буферизировать его в памяти и отсортировать содержимое.

Находящийся в памяти компонент (часто называемый *резидентной таблицей*, memtable), является изменяемым: он буферизует записи данных и служит в качестве цели для операций чтения и записи. Содержимое таблицы в памяти сохраняется на диске, когда его объем достигает настраиваемого порога. Для обновления резидентной таблицы не требуется доступ к диску, и оно не приводит к затратам на ввод-вывод. Отдельный файл журнала упреждающей записи, подобный тому, что мы обсуждали в разделе «Восстановление» на с. 106, необходим для гарантии долговечности записей данных. Записи данных добавляются в журнал и фиксируются в памяти до подтверждения операции клиенту.

Буферизация выполняется в памяти: все операции чтения и записи применяются к резидентной таблице, которая поддерживает отсортированную структуру данных,

допускающую параллельный доступ, — обычно некоторую форму резидентного отсортированного дерева или любую структуру данных, которая может обеспечить аналогичные характеристики производительности.

Дисковые компоненты создаются путем *выгрузки на диск* буферизованного в памяти содержимого. Эти компоненты используются только для чтения: буферизованное содержимое сохраняется на персистентном носителе и файлы никогда не изменяются. Это позволяет нам мыслить в терминах простых операций: запись в резидентную таблицу, чтение с диска и из резидентных таблиц, слияние и удаление файлов.

На протяжении всей этой главы под словом «таблица» будет пониматься таблица, расположенная на диске. Поскольку мы обсуждаем семантику подсистемы хранения, этот термин не конфликтует с концепцией таблицы в более широком контексте СУБД.

Двухкомпонентное LSM-дерево

Мы различаем двух- и многокомпонентные LSM-деревья. *Двухкомпонентные* LSM-деревья имеют только один дисковый компонент, состоящий из нескольких неизменяемых сегментов. Дисковый компонент здесь организован как B-дерево со 100%-ной заполняемостью узлов и страницами, доступными только для чтения.

Содержимое резидентного дерева выгружается на диск по частям. Во время выгрузки для каждого выгружаемого из памяти поддерева мы находим соответствующее поддерево на диске и записываем в новый сегмент на диске результат объединения содержимого сегмента, находящегося в памяти, и содержимого поддерева, находящегося на диске. На рис. 7.1 показаны деревья в памяти и на диске перед слиянием.

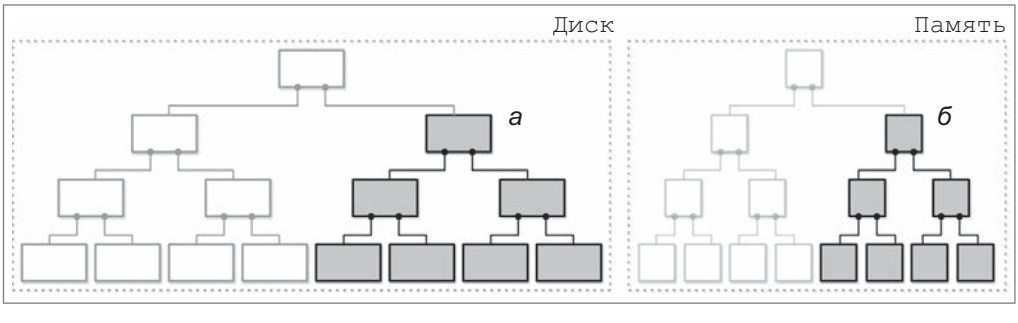

Рис. 7.1. Двухкомпонентное LSM-дерево перед выгрузкой. Выгружаемые сегменты, находящиеся в памяти и на диске, выделены серым цветом

После выгрузки поддерева заменяемые поддеревья в памяти и на диске удаляются и заменяются результатом их слияния, который становится адресуемым из ранее существовавших разделов дискового дерева. На рис. 7.2 показан результат процесса слияния, уже записанный в новое место на диске и присоединенный к остальной части дерева.

Операция слияния реализуется с помощью пошагового продвижения итераторов, производящих одновременное считывание расположенных на диске листовых уз-

лов и содержимого резидентного дерева. Поскольку оба источника отсортированы, для получения отсортированного объединенного результата нам нужно знать лишь текущие значения обоих итераторов на каждом шаге процесса слияния.

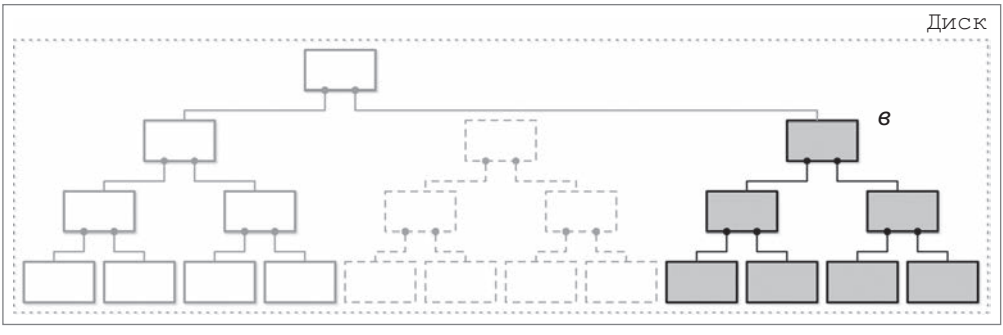

Рис. 7.2. Двухкомпонентное LSM-дерево после выгрузки. Объединенное содержимое выделено серым цветом. Прямоугольники с пунктирными линиями показывают удаленные сегменты, находившиеся на диске

Этот подход является логическим расширением и продолжением нашего разговора о неизменяемых B-деревьях. B-деревья с копированием при записи (см. раздел «Копирование при записи» на с. 129) используют структуру B-дерева, но их узлы заполнены не полностью и для них требуются копирование страниц на пути от корня до листа и создание параллельной структуры дерева. Здесь мы делаем нечто подобное, но, так как мы буферизуем записи в памяти, мы снижаем затраты на обновление дерева, находящегося на диске.

При реализации слияния и выгрузки поддеревьев необходимо обеспечить следующее:

1. *С момента начала* процесса выгрузки все новые записи должны помещаться в новую резидентную таблицу.
2. *Во время* выгрузки поддерева и дисковое поддерево, и выгружаемое резидентное поддерево должны оставаться доступными для чтения.
3. *После* выгрузки публикация объединенного содержимого и удаление оставшегося необъединенного содержимого на диске и в памяти должны выполняться атомарно.

Несмотря на то что двухкомпонентные LSM-деревья могут быть полезны для ведения индексных файлов, на момент написания книги автору не было известно о каких-либо реализациях. Это можно объяснить характерным для этого подхода увеличением объема записи: операции слияния производятся достаточно часто, так как они запускаются при выгрузке резидентной таблицы.

Многокомпонентные LSM-деревья

Давайте рассмотрим альтернативную структуру — многокомпонентные LSM-деревья, которые имеют несколько дисковых таблиц. При этом все содержимое резидентной таблицы выгружается на диск за один подход.

Очень быстро становится очевидным, что после многократных выгрузок мы получим множество дисковых таблиц и с течением времени их количество будет только расти. Поскольку мы не всегда точно знаем, какие таблицы содержат необходимые записи данных, для нахождения искомых данных иногда требуется получать доступ к нескольким файлам.

Необходимость читать из нескольких источников вместо одного может привести к большим затратам. Чтобы смягчить эту проблему и свести количество таблиц к минимуму, запускается периодический процесс слияния, называемый *уплотнением* (compaction) (см. подраздел «Обслуживание в LSM-деревьях» на с. 159). Процесс уплотнения выбирает несколько таблиц, считывает и объединяет их содержимое, после чего записывает результат объединения в новый файл. Прежние таблицы удаляются одновременно с появлением новой объединенной таблицы.

На рис. 7.3 показан жизненный цикл данных многокомпонентного LSM-дерева. Данные сначала буферизуются в компоненте, находящемся в памяти. Когда он становится слишком большим, его содержимое выгружается на диск для создания дисковых таблиц. Позже несколько таблиц объединяется для создания более крупных таблиц.

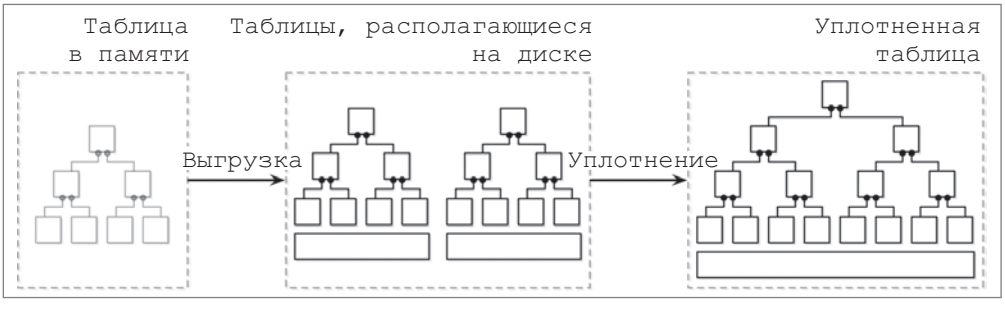

Рис. 7.3. Жизненный цикл данных многокомпонентного LSM-дерева

Остальная часть этой главы будет посвящена многокомпонентным LSM-деревьям, их составным частям и процессам обслуживания.

Резидентные таблицы

Выгрузка резидентной таблицы может запускаться периодически или при превышении максимально допустимого размера. Перед выгрузкой резидентную таблицу необходимо произвести ее *переключение* (switch): при этом выделяется новая резидентная таблица, которая становится объектом для всех новых операций записи, в то время как прежняя таблица переходит в состояние выгрузки. Эти два шага должны выполняться атомарно. Выгружаемая резидентная таблица остается доступной для чтения до момента полной выгрузки ее содержимого. После этого прежняя резидентная таблица удаляется и заменяется вновь записанной дисковой таблицей, которая становится доступной для чтения.

На рис. 7.4 показаны компоненты LSM-дерева, связи между ними и операции, выполняемые для перехода от одного к другому:

Текущая резидентная таблица

Принимает записи и обеспечивает чтение.

Выгружаемая резидентная таблица

Доступна для чтения.

Цель выгрузки, находящаяся на диске

Не участвует в операциях чтения, так как ее содержимое является неполным.

Выгруженные таблицы

Доступны для чтения сразу после удаления выгруженной резидентной таблицы.

Таблицы в процессе уплотнения

Дисковые таблицы, подвергаемые слиянию в данный момент.

Уплотненные таблицы

Создаются из выгруженных или других уплотненных таблиц.

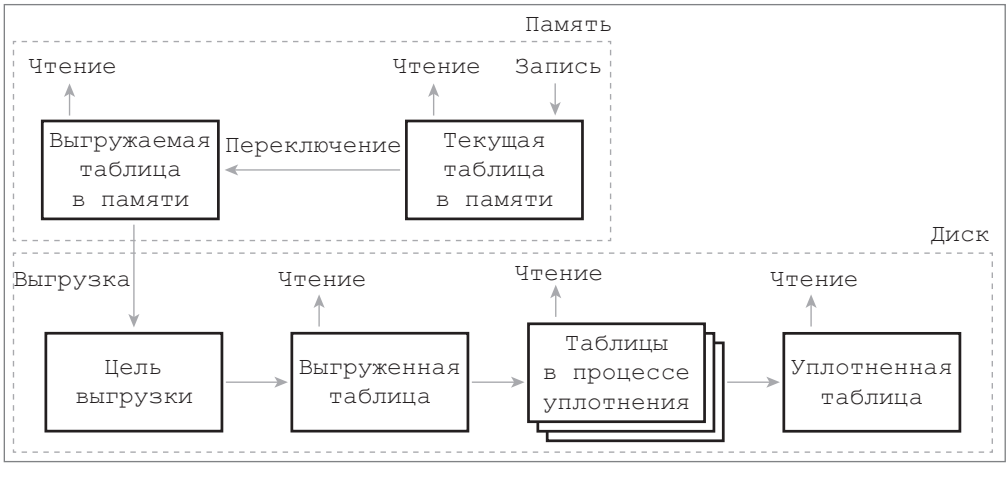

Рис. 7.4. Компоненты LSM-дерева

Поскольку уже в памяти данные размещаются в отсортированном виде, дисковую таблицу можно создать путем последовательной записи резидентного содержимого на диск. Во время выгрузки для чтения доступны и выгружаемая, и текущая резидентная таблицы.

До момента полной выгрузки резидентной таблицы единственная дисковая версия ее содержимого хранится в журнале упреждающей записи. После полной выгрузки на диск резидентной таблицы можно произвести *усечение* журнала — удалить область

журнала со сведениями об операциях, примененных к выгруженной резидентной таблице.

Обновления и удаления

В LSM-деревьях для выполнения операций вставки, обновления и удаления не требуется искать записи данных на диске. Вместо этого обновления и удаления представлены как отдельные записи, которые согласовываются во время чтения.

При этом недостаточно лишь удалить записи данных из резидентной таблицы, так как другие дисковые или резидентные таблицы могут содержать записи данных с таким же ключом. Если бы мы реализовали удаление с помощью простого удаления элементов из резидентной таблицы, то наши операции удаления в конечном итоге либо не оказывали бы никакого влияния, либо заново восстанавливали бы прежние значения.

Давайте рассмотрим пример. Выгруженная дисковая таблица содержит запись данных `v1`, связанную с ключом `k1`, а резидентная таблица содержит новое значение этой записи `v2`:

```
Дисковая таблица   Резидентная таблица
| k1 | v1 |         | k1 | v2 |
```

Если мы просто удалим значение `v2` из резидентной таблицы и выгрузим его, мы фактически восстановим значение `v1`, так как оно станет единственным значением, связанным с этим ключом:

```
Дисковая таблица   Резидентная таблица
| k1 | v1 |         ∅
```

По этой причине удаление необходимо записывать *явным образом*. Это можно обеспечить, вставляя специальную *запись удаления* (иногда называемую *отметкой полного удаления* или *сертификатом неактивности*), указывающую на удаление записи данных, связанной с определенным ключом:

```
Дисковая таблица   Резидентная таблица
| k1 | v1 |         | k1 | <отметка полного удаления> |
```

Процесс согласования собирает отметки полного удаления и отфильтровывает затененные значения.

Иногда бывает полезно удалить последовательный диапазон ключей, а не один ключ. Это можно сделать с помощью *предикатных операций удаления*, которые добавляют запись удаления согласно предикату, отсеивающему элементы в соответствии с обычными правилами сортировки записей. В процессе согласования записи данных, соответствующие предикату, пропускаются и не возвращаются клиенту.

Предикаты могут представлять собой выражение вида `DELETE FROM table WHERE key ≥ "k2" AND key < "k4"`, задающее любой нужный диапазон. В СУБД Apache Cassandra этот подход реализован с помощью так называемых отметок полного

удаления диапазона (*range tombstones*). Отметка полного удаления диапазона вместо одного ключа охватывает целый диапазон ключей.

При использовании отметок полного удаления диапазона необходимо тщательно продумывать правила резолюции с учетом перекрывающихся диапазонов и границ дисковых таблиц. Например, следующая комбинация скроет из конечного результата записи данных, связанные с ключами `k2` и `k3`:

```
Дисковая таблица 1          Дисковая таблица 2
| k1 | v1 |                 | k2 | <start_tombstone_inclusive> |
| k2 | v2 |                 | k4 | <end_tombstone_exclusive> |
| k3 | v3 |
| k4 | v4 |
```

Поиск в LSM-деревьях

LSM-деревья состоят из нескольких компонентов. Во время поиска, как правило, производится обращение к нескольким компонентам, поэтому их содержимое должно быть объединено и согласовано перед возвращением клиенту. Чтобы лучше понять процесс слияния, давайте посмотрим, как производится итерация по таблицам во время слияния и как объединяются конфликтующие записи.

Итерация слиянием

Поскольку содержимое дисковых таблиц является отсортированным, мы можем использовать алгоритм сортировки многоканальным слиянием. Например, у нас есть три источника: две дисковые и одна резидентная таблица. Обычно подсистемы хранения позволяют использовать *курсор* (или *итератор*) для перемещения по содержимому файла. Курсор содержит смещение последней использованной записи данных, позволяет проверить, завершилась ли итерация, и может быть использован для извлечения следующей записи данных.

В алгоритме сортировки многоканальным слиянием используется *очередь с приоритетами*, такая как *min-куча* [SEDGEWICK11], которая содержит до `N` элементов (где `N` — число итераторов), сортирует свое содержимое и подготавливает для возврата следующий в очереди наименьший элемент. Головной элемент каждого итератора помещается в очередь. При этом в голове очереди находится минимальный элемент среди всех итераторов.

Очередь с приоритетами — это структура данных, используемая для обслуживания упорядоченной очереди элементов. В то время как обычная очередь сохраняет элементы в порядке их добавления (первый вошел — первый вышел), очередь с приоритетами пересортировывает элементы при вставке, помещая в голову очереди элемент с самым высоким (или самым низким) приоритетом. Это особенно полезно для итерации слиянием, так как мы должны выводить элементы в отсортированном порядке.

Когда наименьший элемент удаляется из очереди, связанный с ним итератор проверяется на наличие следующего значения, которое затем помещается в очередь, которая пересортировывается для сохранения порядка.

Поскольку все содержимое итератора расположено в отсортированном виде, при повторной вставке значения из итератора, который содержал предыдущее наименьшее значение среди всех головных элементов итераторов, по-прежнему сохраняется инвариант, сводящийся к тому, что в голове очереди должен находиться минимальный элемент среди всех итераторов. Всякий раз, когда один из итераторов исчерпывается, алгоритм продолжает работу без повторной вставки головного элемента следующего итератора. Алгоритм продолжает работу до тех пор, пока не будут выполнены условия запроса или не будут исчерпаны все итераторы.

На рис. 7.5 схематически изображен только что описанный процесс слияния: головные элементы (светло-серые элементы исходных таблиц) помещаются в очередь с приоритетами. Из очереди с приоритетами элементы возвращаются в выходной итератор. Полученный результат является отсортированным.

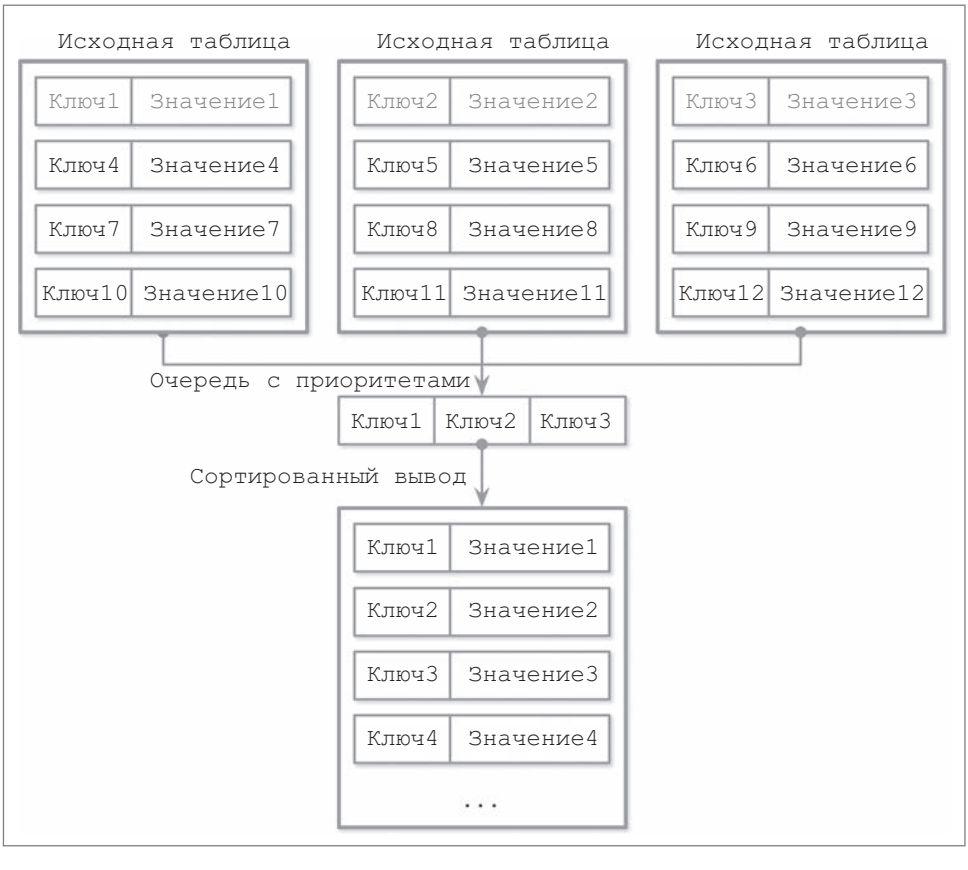

Рис. 7.5. Механика слияния в LSM-деревьях

Может случиться так, что во время итерации слиянием мы столкнемся с несколькими записями данных для одного и того же ключа. Из инвариантов очереди с приоритетами и итератора мы знаем, что если каждый итератор содержит только одну запись данных на каждый ключ и мы получаем в очереди несколько записей для одного и того же ключа, эти записи данных, очевидно, получены от разных итераторов.

Давайте рассмотрим один пример шаг за шагом. В качестве входных данных у нас есть итераторы по двум дисковым таблицам:

```
Итератор 1:           Итератор 2:
{k2: v1} {k4: v2}     {k1: v3} {k2: v4} {k3: v5}
```

Очередь с приоритетами заполняется головными элементами итераторов:

```
Итератор 1:       Итератор 2:         Очередь с приоритетами:
{k4: v2}          {k2: v4} {k3: v5}   {k1: v3} {k2: v1}
```

Ключ `k1` является наименьшим ключом в очереди и добавляется к результату. Поскольку он получен из итератора 2, мы пополняем очередь из него:

```
Итератор 1:  Итератор 2:   Очередь с приоритетами:  Объединенный результат:
{k4: v2}     {k3: v5}      {k2: v1} {k2: v4}        {k1: v3}
```

Теперь в очереди у нас есть две записи для ключа `k2`. Согласно вышеупомянутым инвариантам, мы можем быть уверены, что в любом из итераторов нет других записей с тем же ключом. Записи с одинаковыми ключами объединяются и добавляются к объединенному результату.

Очередь пополняется данными из обоих итераторов:

```
Итератор 1:  Итератор 2:  Очередь с приоритетами:   Объединенный результат:
{}           {}           {k3: v5} {k4: v2}         {k1: v3} {k2: v4}
```

Поскольку все итераторы теперь пусты, мы добавляем оставшееся содержимое очереди к выходным данным:

```
Объединенный результат:
{k1: v3} {k2: v4} {k3: v5} {k4: v2}
```

Таким образом, для создания объединенного итератора необходимо повторить следующие шаги:

1. Сначала заполните очередь первыми элементами из каждого итератора.
2. Извлеките из очереди наименьший (головной) элемент.
3. Пополните очередь из соответствующего итератора, если этот итератор не исчерпан.

В плане сложности алгоритм слияния итераторов аналогичен слиянию отсортированных коллекций. Его затраты по памяти составляют `O(N)`, где `N` — число итераторов. Отсортированная коллекция головных элементов итераторов поддерживается со сложностью `O(log N)` (средний случай) [KNUTH98].

Согласование

Итерация слиянием — это всего лишь один аспект того, что необходимо сделать для объединения данных из нескольких источников. Другим важным аспектом является *согласование* (reconciliation) и *разрешение конфликтов* записей данных, связанных с одним и тем же ключом.

Различные таблицы могут содержать записи данных для одного и того же ключа, такие как обновления и удаления, и их содержимое необходимо согласовывать. Реализация очереди с приоритетами из предыдущего примера должна предусматривать наличие нескольких значений, связанных с одним и тем же ключом, и инициировать согласование.

Операция, которая вставляет запись в базу данных, если та не существует, или обновляет существующую в противном случае, называется *upsert* (обновление или вставка). В *LSM*-деревьях операции вставки и обновления неразличимы, так как они не пытаются найти записи данных, ранее связанные с ключом во всех источниках, и переназначить его значение, поэтому мы можем сказать, что по умолчанию для записей мы выполняем операцию обновления или вставки *upsert*.

Чтобы согласовать записи данных, нам нужно понять, какая из них имеет приоритет. Записи данных содержат необходимые для этого метаданные, такие как временные метки. Чтобы установить порядок поступления элементов из нескольких источников и выяснить, какой из них является более поздним, мы можем сравнить их временные метки.

Записи, затененные записями с более поздними временными метками, не возвращаются клиенту и не записываются во время уплотнения.

Обслуживание в LSM-деревьях

Подобно изменяемым B-деревьям, LSM-деревья требуют обслуживания. На характер этих процессов большое влияние оказывают инварианты, поддерживаемые алгоритмами.

В B-деревьях процесс обслуживания собирает неиспользуемые ячейки и дефрагментирует страницы, высвобождая пространство, занимаемое удаленными и затененными записями. В LSM-деревьях число дисковых таблиц постоянно растет, но его можно уменьшить путем периодического уплотнения.

Процесс уплотнения выбирает несколько дисковых таблиц, перебирает все их содержимое, используя вышеупомянутые алгоритмы слияния и согласования, и записывает результаты во вновь созданную таблицу.

Поскольку содержимое дисковой таблицы является отсортированным, а также из-за того, как работает сортировка слиянием, уплотнение работает с теоретической верхней границей использования памяти, поскольку оно должно содержать в памяти

только головные элементы итераторов. Все содержимое таблицы перебирается последовательно. Получаемые результаты слияния также записываются последовательно. Этот процесс может иметь некоторые специфические отличия в разных реализациях из-за дополнительных оптимизаций.

Уплотняемые таблицы остаются доступными для чтения до завершения процесса уплотнения, а это означает, что в течение всего процесса уплотнения на диске должно быть достаточно свободного места для записи уплотненной таблицы.

В любой момент времени в СУБД может выполняться несколько операций уплотнения. Однако эти параллельные процессы уплотнения обычно работают с непересекающимися наборами таблиц. Записывающий процесс уплотнения может и объединять несколько таблиц в одну, и разбивать одну таблицу на несколько таблиц.

ОТМЕТКИ ПОЛНОГО УДАЛЕНИЯ И УПЛОТНЕНИЕ

Отметки полного удаления представляют собой важную часть информации, необходимой для правильного согласования, поскольку в какой-то другой таблице все еще может храниться устаревшая запись данных, затеняемая такой отметкой.

Во время уплотнения отметки полного удаления не сбрасываются сразу. Они сохраняются до тех пор, пока подсистема хранения не убедится, что ни в одной другой таблице нет записи данных для того же ключа с меньшей временной меткой. В СУБД RocksDB отметки полного удаления хранятся, пока они не достигнут самого нижнего уровня (https://databass.dev/links/74). В СУБД Apache Cassandra отметки полного удаления хранятся в течение периода отсрочки сборки мусора (https://databass.dev/links/75) в силу конечно согласованного характера базы данных, что гарантирует, что эти отметки увидят другие узлы. Сохранение отметок полного удаления в процессе уплотнения важно, потому что необходимо избежать восстановления старых данных.

Поуровневое уплотнение

Уплотнение открывает множество возможностей для оптимизации, и существует множество различных *стратегий уплотнения* (compaction strategy). Одна из часто реализуемых стратегий называется *поуровневым уплотнением* (leveled compaction). Она используется, например, в СУБД RocksDB (https://databass.dev/links/76).

Поуровневое уплотнение разделяет дисковые таблицы на уровни. Таблицы на каждом уровне имеют целевые размеры, и каждый уровень имеет соответствующий индексный номер (идентификатор). Не совсем интуитивным является то, что уровень с самым высоким индексом считается самым нижним уровнем. Для ясности в этом разделе не используются термины «более высокий уровень» и «более низкий уровень», а используются определения, согласующиеся с индексом уровня. То есть поскольку 2 больше 1, уровень 2 имеет более высокий индекс, чем уровень 1. Термины «предыдущий» и «следующий» имеют ту же семантику порядка, что и индексы уровней.

Таблицы уровня 0 создаются путем выгрузки содержимого резидентных таблиц. Таблицы уровня 0 могут содержать перекрывающиеся диапазоны ключей. Как толь-

ко количество таблиц на уровне 0 достигает порогового значения, их содержимое сливается с созданием новых таблиц для уровня 1.

Диапазоны ключей для таблиц уровня 1 и всех уровней с более высоким индексом не перекрываются, поэтому таблицы уровня 0 должны быть секционированы во время уплотнения, разбиты на диапазоны и объединены с таблицами, содержащими соответствующие диапазоны ключей. Кроме того, процесс уплотнения может включать все таблицы уровней 0 и 1, а также выходные секционированные таблицы уровня 1.

На уровнях с более высокими индексами операции уплотнения выбирают таблицы из двух последовательных уровней с перекрывающимися диапазонами и создают новую таблицу на более высоком уровне. На рис. 7.6 схематично показано, как процесс уплотнения переносит данные между уровнями. Процесс уплотнения таблиц уровней 1 и 2 приведет к созданию новой таблицы на уровне 2. В зависимости от способа секционирования таблиц для уплотнения можно выбрать несколько таблиц одного уровня.

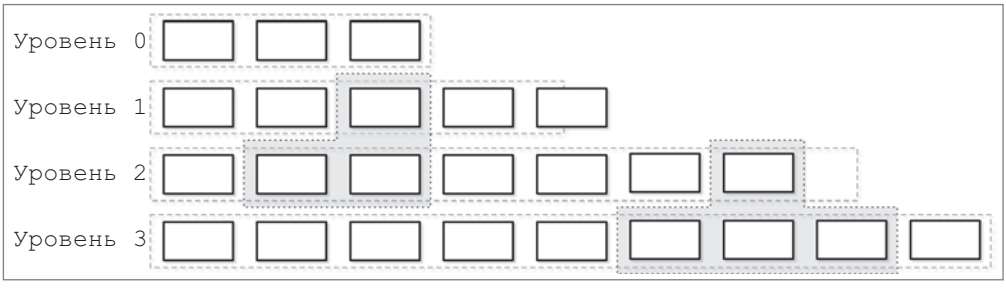

Рис. 7.6. Процесс уплотнения. Серые прямоугольники с пунктирными линиями представляют уплотняющиеся в текущий момент таблицы. Прямоугольники шириной на весь уровень показывают целевой порог размера данных на уровне. На уровне 1 — превышение лимита

Сохранение разных диапазонов ключей в отдельных таблицах уменьшает количество таблиц, к которым осуществляется доступ во время чтения. Уменьшение достигается путем проверки метаданных таблицы и отсеивания таблиц, диапазоны которых не содержат искомого ключа.

Каждый уровень имеет ограничение на размер таблицы и максимальное количество таблиц. Как только число таблиц на уровне 1 или любом уровне с более высоким индексом достигает порогового значения, таблицы текущего уровня объединяются с таблицами следующего уровня, содержащими перекрывающийся диапазон ключей.

Размеры увеличиваются экспоненциально при переходе от уровня к уровню: таблицы на каждом следующем уровне экспоненциально больше, чем таблицы на предыдущем. Таким образом, самые свежие данные всегда находятся на уровне с самым низким индексом, а старые данные постепенно мигрируют на более высокие уровни.

Многоуровневое уплотнение по размеру

Другая популярная стратегия уплотнения называется *многоуровневым уплотнением по размеру* (size-tiered compaction). При использовании этого подхода вместо группировки дисковых таблиц по уровням они группируются по размеру: меньшие таблицы группируются с меньшими, а бóльшие — с бóльшими.

Уровень 0 содержит самые малые таблицы из тех, которые были либо выгружены из памяти, либо созданы в процессе уплотнения. Когда таблицы уплотняются, полученная объединенная таблица записывается на уровень, содержащий таблицы с соответствующими размерами. Этот процесс продолжает рекурсивно увеличивать уровни, уплотняя и продвигая бóльшие таблицы на более высокие уровни и понижая меньшие таблицы на более низкие уровни.

Существуют и другие широко реализуемые стратегии уплотнения, которые могут быть оптимальными для различных рабочих нагрузок. Например, в СУБД Apache Cassandra реализована стратегия сжатия временных окон (time window compaction strategy, https://databass.dev/links/77), которая особенно полезна для работы с временными рядами, когда записи имеют заданный срок жизни (т. е. срок актуальности элемента истекает через определенный период времени).

Одной из проблем многоуровневого уплотнения по размеру является *недостаток таблиц* (table starvation): если уплотненные таблицы все еще достаточно малы после уплотнения (например, записи были затенены отметками полного удаления и не попали в объединенную таблицу), более высокие уровни могут содержать слишком мало таблиц для выполнения уплотнения, и их отметки полного уплотнения не будут приняты во внимание, что увеличит затраты на операции чтения. В подобных случаях уплотнение должно выполняться для уровня принудительно, даже если он не содержит достаточное количество таблиц.

Стратегия уплотнения временных окон учитывает временные метки операций записи и позволяет отбрасывать целые файлы, которые содержат данные для уже истекшего временного диапазона, без необходимости в уплотнении и перезаписи их содержимого.

Чтение, запись и увеличение пространства

При реализации оптимальной стратегии уплотнения мы должны учитывать множество факторов. Один из подходов заключается в том, чтобы тратить меньше пространства за счет высвобождения места, занимаемого дублирующими записями, что повышает показатель увеличения объема записи в силу последовательной перезаписи таблиц. Альтернативой является отказ от последовательной перезаписи данных, что повышает показатель увеличения объема чтения (затраты на согласование записей данных, связанных с одним и тем же ключом во время чтения) и показатель увеличения пространства (так как избыточные записи сохраняются в течение более длительного времени).

Предметом больших споров в сообществе баз данных является вопрос о том, у какой разновидности деревьев меньше показатель увеличения объема записи: у B-деревьев или у LSM-деревьев. Чрезвычайно важно понять источник увеличения объема записи в обоих случаях. В B-деревьях это происходит из-за операций обратной записи и последовательных обновлений одного и того же узла. В LSM-деревьях увеличение объема записи вызвано миграцией данных из одного файла в другой во время уплотнения. Прямое сравнение этих двух факторов может привести к неверным предположениям.

Таким образом, при хранении неизменяемых данных на диске мы сталкиваемся с тремя проблемами:

Увеличение объема чтения

Возникает в результате необходимости обращения к нескольким таблицам для извлечения данных.

Увеличение объема записи

Вызывается последовательными операциями перезаписи в процессе уплотнения.

Увеличение пространства

Возникает в результате хранения нескольких записей, связанных с одним и тем же ключом.

Мы будем затрагивать каждую из них на протяжении всей оставшейся части главы.

Гипотеза RUM

Одна из популярных моделей затрат для структур хранения учитывает три фактора: чтение, обновление и объем занимаемой памяти. Она называется гипотезой RUM [ATHANASSOULIS16].

Гипотеза RUM гласит, что снижение затрат по двум параметрам неизбежно приводит к изменению в худшую сторону третьего и что оптимизация может быть выполнена только за счет одного из трех параметров. Мы можем сравнить различные подсистемы хранения данных с точки зрения этих трех параметров и понять, какие из них оптимизированы в каждом случае и какие потенциальные компромиссы это подразумевает.

Идеальное решение обеспечило бы самые низкие затраты на чтение при сохранении низкого расхода памяти и низких затрат на запись, но в реальности это недостижимо, и нам предлагается компромисс.

B-деревья оптимизированы для чтения. Для записи в B-дерево необходимо найти запись на диске, и в случае последовательных операций записи на одной и той же странице часто требуется обновлять страницу на диске несколько раз. Зарезервированное дополнительное пространство для будущих обновлений и удалений увеличивает объем занимаемого пространства.

В случае LSM-деревьев не требуется находить запись на диске во время записи и резервировать дополнительное пространство для будущих операций записи. Но для хранения избыточных записей все равно требуется некоторый объем пространства. В конфигурации по умолчанию чтение обходится дороже, так как для возвращения полных результатов требуется доступ к нескольким таблицам. Однако оптимизации, которые мы обсуждаем в этой главе, помогают смягчить эту проблему.

Как мы видели в главах, посвященных B-деревьям, и увидим в этой главе, есть способы улучшить эти характеристики благодаря различным оптимизациям.

Эта модель затрат неидеальна, поскольку не учитывает другие важные показатели, такие как время задержки, схемы доступа, сложность реализации, затраты на обслуживание и аппаратные особенности. Понятия более высокого уровня, важные для распределенных баз данных, такие как влияние согласованности и затраты на репликацию, также не рассматриваются. Однако эту модель можно использовать в качестве первого приближения и эмпирического правила, поскольку она помогает понять, что может предложить та или иная подсистема хранения.

Подробнее о реализации

Мы рассмотрели основную механику LSM-деревьев: как данные считываются, записываются и уплотняются. Однако рассмотрения заслуживают и некоторые другие общие аспекты многих реализаций LSM-дерева: как реализуются резидентные и дисковые таблицы, как работают вторичные индексы, как можно сократить количество дисковых таблиц, доступ к которым осуществляется во время чтения, и наконец, новые идеи, связанные с журналированным хранилищем.

Сортированные таблицы строк SSTable

До сих пор мы обсуждали иерархическую и логическую структуры LSM-деревьев (то, что они состоят из нескольких компонентов, расположенных в памяти и на диске), но еще не обсуждали, как реализуются дисковые таблицы и как их устройство сочетается с остальной частью системы.

Дисковые таблицы обычно реализуются с помощью *отсортированных строковых таблиц* (SSTable). Как следует из названия, записи данных в этих таблицах сортируются и располагаются согласно порядку ключей. Обычно они состоят из двух компонентов: индексных файлов и файлов данных. Индексные файлы реализуются с помощью некоторой структуры, обеспечивающей логарифмический поиск, такой как B-дерево, или структуры с постоянным временем поиска, такой как хэш-таблица.

Поскольку файлы данных содержат записи согласно порядку ключей, использование хэш-таблиц для индексирования не помешает нам реализовать сканирование диапазона, поскольку доступ к хэш-таблице осуществляется только для нахождения первого ключа в диапазоне, а сам диапазон можно считывать из файла данных последовательно — пока выполняется предикат диапазона.

Индексный компонент содержит ключи и записи данных (смещения в файле данных, указывающее место размещения реальных записей данных). Компонент данных состоит из сцепленных пар «ключ–значение». Устройство ячеек и форматы записей данных, которые мы обсуждали в главе 3, во многом применимы и к отсортированным строковым таблицам. Основное отличие здесь заключается в том, что ячейки записываются последовательно и не изменяются в течение жизненного цикла таблицы. Поскольку индексные файлы содержат указатели на записи данных, хранящиеся в файле данных, смещения данных должны быть известны к моменту создания индекса.

Во время уплотнения файлы данных можно читать последовательно, не обращаясь к индексу, так как записи данных в них уже упорядочены. Поскольку таблицы, подвергаемые слиянию во время уплотнения, отсортированы в одинаковом порядке и итерация слиянием сохраняет порядок, результирующая объединенная таблица также создается путем последовательного занесения записей данных за один подход. Как только файл полностью записан, он считается неизменяемым и его находящееся на диске содержимое не подвергается модификации.

ПРИКРЕПЛЕННЫЕ К SSTABLE ВТОРИЧНЫЕ ИНДЕКСЫ

Одной из интересных разработок в области индексации LSM-дерева являются индексы SASI (SSTable-Attached Secondary Indexes, прикрепленные к SSTable вторичные индексы), реализованные в СУБД Apache Cassandra. Чтобы индексировать содержимое таблицы не только по первичному ключу, но и по любому другому полю, индексные структуры и их жизненные циклы сопрягаются с жизненным циклом отсортированных строковых таблиц и создается индекс для каждой такой таблицы. При выгрузке резидентной таблицы ее содержимое записывается на диск и вместе с индексом первичного ключа для таблиц SSTable создаются файлы вторичных индексов.

Поскольку LSM-деревья буферизуют данные в памяти и индексы должны работать и для резидентного, и для дискового содержимого, концепция SASI поддерживает отдельную резидентную структуру, индексирующую содержимое резидентной таблицы.

Во время чтения первичные ключи искомых записей находятся путем поиска и слияния содержимого индексов, а записи данных объединяются и согласовываются так же, как в случае обычного поиска по LSM-дереву.

Одним из преимуществ совмещения жизненного цикла таблиц SSTable является то, что индексы могут быть созданы во время выгрузки или уплотнения резидентной таблицы.

Фильтры Блума

Источником увеличения объема чтения в LSM-деревьях является то, что для выполнения операции чтения мы должны обращаться к нескольким дисковым таблицам. Это происходит потому, что мы не всегда заранее знаем, содержит ли дисковая таблица запись данных для искомого ключа.

Один из способов обойтись без поиска по таблице сводится к тому, чтобы хранить диапазон ее ключей (наименьший и наибольший ключи, хранящиеся в этой таблице)

в метаданных и проверять, принадлежит ли искомый ключ диапазону этой таблицы. Эта информация неточна и сообщает только о том, *может ли* запись данных присутствовать в таблице. В качестве усовершенствования многие реализации, включая Apache Cassandra (https://databass.dev/links/78) и RocksDB (https://databass.dev/links/79), используют структуру данных, называемую *фильтром Блума*.

Фильтр Блума, разработанный Бертоном Говардом Блумом в 1970 году [BLOOM70], представляет собой экономичную по занимаемому пространству вероятностную структуру данных, которую можно использовать для проверки, входит ли элемент во множество или нет. Он может выдавать ложноположительные совпадения (сообщать, что элемент является членом множества, в то время как его там нет), но не может выдавать ложноотрицательные совпадения (если возвращается отрицательное совпадение, элемент гарантированно не является членом множества).

Вероятностные структуры данных, как правило, более эффективны с точки зрения объема занимаемого пространства, чем их «обычные» аналоги. Например, чтобы проверить принадлежность множеству, определить мощность (узнать количество различных элементов во множестве) или частоту (количество вхождений определенного элемента), мы должны были бы сохранить все элементы множества и перебрать все множество данных, чтобы получить результат. Вероятностные структуры позволяют сохранять ориентировочную информацию и выполнять запросы, выдающие результат с некоторой долей неопределенности. Некоторые общеизвестные примеры таких структур данных: фильтр Блума (для принадлежности к множеству), алгоритм HyperLogLog (для оценки мощности) [FLAJOLET12] и структура Count-Min Sketch (для оценки частоты) [CORMODE12].

Другими словами, фильтр Блума позволяет выяснить, что ключ *может находиться в таблице* или что он *определенно не находится в таблице*. Файлы, для которых фильтр Блума возвращает отрицательное совпадение, пропускаются во время выполнения запроса. Остальные файлы проверяются на наличие искомой записи данных. Применение фильтров Блума к таблицам на диске позволяет значительно сократить число таблиц, к которым осуществляется доступ во время чтения.

В фильтре Блума используются большой битовый массив и несколько хэш-функций. Хэш-функции применяются к ключам записей таблицы для поиска индексов, биты которых в битовом массиве установлены в `1`. Биты, установленные в `1` во всех позициях, определенных хэш-функциями, указывают на присутствие ключа во множестве. В ходе поиска при проверке наличия элемента в фильтре Блума для ключа снова вычисляются хэш-функции, и если биты, определяемые всеми хэш-функциями, равны `1`, мы возвращаем положительный результат, утверждающий, что элемент является членом множества с определенной вероятностью. Если хотя бы один из битов равен `0`, мы можем точно сказать, что элемент отсутствует во множестве.

Хэш-функции, применяемые к разным ключам, могут возвращать одну и ту же позицию бита и приводить к *хэш-конфликту*, а установленные в `1` биты означают только то, что некоторая хэш-функция выдала эту позицию бита для некоторого ключа.

Вероятность ложных срабатываний настраивается посредством настройки размера набора битов и количества хэш-функций: в большем наборе битов меньше вероятность конфликта; аналогично, имея больше хэш-функций, мы можем проверить больше битов и получить более точный результат.

Больший набор битов занимает больше памяти, а вычисление большего числа хэш-функций может оказать отрицательное влияние на производительность, поэтому мы должны найти разумную середину между приемлемой вероятностью и понесенными затратами. Вероятность может быть вычислена на основе ожидаемого размера множества. Поскольку таблицы в LSM-деревьях неизменяемы, размер множества (количество ключей в таблице) известен заранее.

Давайте рассмотрим простой пример, показанный на рис. 7.7. У нас есть 16-битный массив и 3 хэш-функции, которые выдают значения `3`, `5` и `10` для ключа `key1`. Теперь мы устанавливаем биты в этих позициях. Добавляется следующий ключ, и хэш-функции выдают для ключа `key2` значения `5`, `8` и `14`, для которых мы также устанавливаем биты.

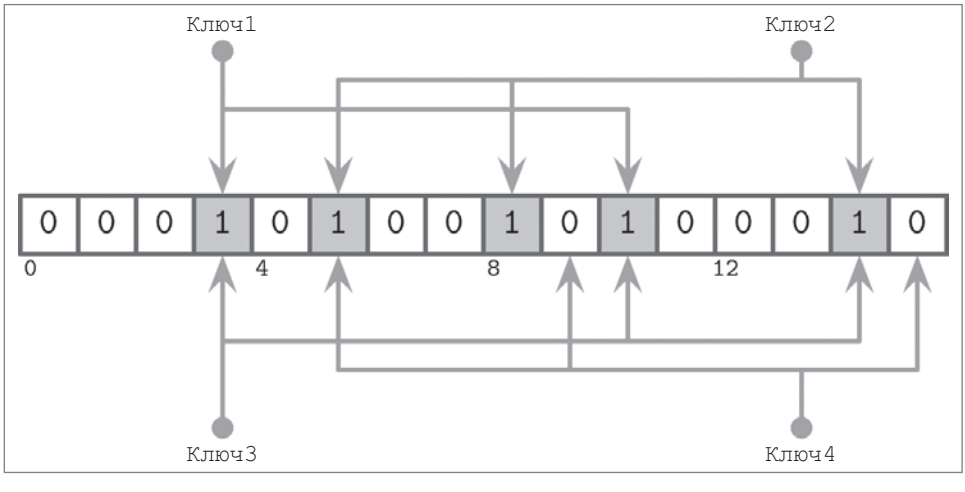

Рис. 7.7. Фильтр Блума

Теперь мы пытаемся проверить, присутствует ли во множестве ключ `key3` — хэш-функции выдают `3`, `10` и `14`. Поскольку все три бита были установлены при добавлении ключей `key1` и `key2`, мы имеем ситуацию, в которой фильтр Блума возвращает ложноположительное совпадение: ключ `key3` никогда не добавлялся туда, но все вычисленные биты установлены. Однако поскольку фильтр Блума только утверждает, что элемент может быть в таблице, этот результат приемлем.

Если мы попытаемся выполнить поиск для ключа `key4` и получим значения `5`, `9` и `15`, то обнаружим, что установлен только бит `5`, а два других бита не установлены. Если хотя бы один из битов не установлен, мы точно знаем, что элемент никогда не добавлялся к фильтру.

Список с пропусками

Существует множество различных структур данных для хранения отсортированных данных в памяти, и одна из них, которая в последнее время становится все более популярной благодаря своей простоте, называется список с пропусками [PUGH90b]. С точки зрения реализации список с пропусками не намного сложнее, чем односвязный список, и его вероятностные гарантии сложности близки к таковым у деревьев поиска.

Списки с пропусками не требуют вращения или перемещения для операций вставки и обновления, а вместо этого используют вероятностную балансировку. Списки с пропусками, как правило, менее эффективны при кэшировании, чем резидентные B-деревья, поскольку узлы списков малы и распределяются в памяти произвольным образом. Некоторые реализации повышают эффективность кэширования, используя развернутые связанные списки (https://databass.dev/links/80).

Список с пропусками состоит из ряда узлов разной высоты, выстраивающих связанные иерархии, позволяющие пропускать диапазоны элементов. Каждый узел содержит ключ, и, в отличие от узлов в связанном списке, некоторые узлы имеют более одного последующего элемента. Узел с высотой `h` связан с одним или несколькими предшествующими узлами с высотой не более `h`. Узлы на самом нижнем уровне могут быть связаны с узлами любой высоты.

Высота узла определяется случайной функцией и вычисляется во время вставки. Узлы с одинаковой высотой образуют уровень. Количество уровней ограничено, чтобы избежать бесконечного роста, а максимальная высота выбирается исходя из максимального количества элементов, вмещаемого структурой. На каждом следующем уровне узлов экспоненциально меньше.

Поисковые запросы выполняются путем перехода по указателям узлов на самом высоком уровне. Как только процесс поиска обнаруживает узел, содержащий ключ, который *больше*, чем искомый, совершается переход по ссылке предшественника на узел, находящийся на следующем уровне. То есть если искомый ключ *больше*, чем ключ текущего узла, поиск продолжается по направлению вперед. Если искомый ключ *меньше* ключа текущего узла, то поиск продолжается с предыдущего узла на следующем уровне. Этот процесс повторяется рекурсивно до тех пор, пока не будет найден искомый ключ или его предшественник.

Например, поиск ключа `7` в списке с пропусками, показанном на рис. 7.8, можно выполнить следующим образом:

1. Переходим по указателю на самом высоком уровне к узлу, который содержит ключ `10`.
2. Поскольку искомый ключ `7` *меньше* `10`, переходим по указателю следующего уровня от головного узла и находим узел, содержащий ключ `5`.
3. Переходим по указателю самого высокого уровня в этом узле и снова находим узел, содержащий ключ `10`.
4. Искомый ключ `7` *меньше* `10`, поэтому переходим по указателю следующего уровня от узла, содержащего ключ `5`, и находим узел, содержащий искомый ключ `7`.

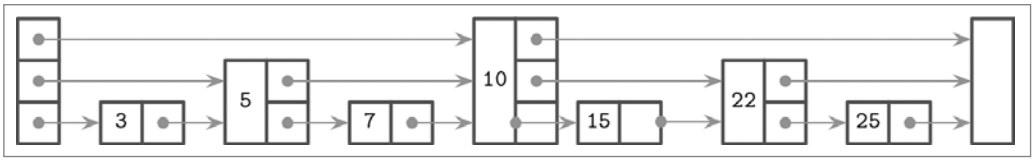

Рис. 7.8. Список с пропусками

Во время вставки точка вставки (узел, содержащий ключ или его предшественник) находится с помощью вышеупомянутого алгоритма и создается новый узел. Чтобы построить древовидную иерархию и сохранить баланс, высота узла определяется с помощью случайного числа, генерируемого на основе распределения вероятностей. Указатели в предшествующих узлах с ключами, *не превышающими* ключ вновь созданного узла, направляются на этот узел. При этом их указатели более высокого уровня остаются нетронутыми. Указатели во вновь созданном узле связываются с соответствующими последующими узлами на каждом уровне.

Во время удаления прямые указатели удаляемого узла помещаются на предшествующие узлы на соответствующих уровнях.

Мы можем создать версию списка с пропусками, работающую в параллельном окружении, реализовав схему линеаризуемости, которая использует дополнительный флаг `fully_link`, указывающий, обновлены ли полностью указатели узла. Этот флаг можно установить с помощью операции «Сравнение с обменом» [HERLIHY10]. Это необходимо, поскольку указатели узлов должны обновляться на нескольких уровнях, чтобы полностью восстановить структуру списка с пропусками.

В языках с неуправляемой моделью памяти можно использовать подсчет ссылок или *указатели опасности*, чтобы узлы, на которые в данный момент установлена ссылка, не освобождались при параллельном доступе к ним [RUSSEL12]. Этот алгоритм не подвержен взаимоблокировкам, так как доступ к узлам всегда осуществляется с более высоких уровней.

В СУБД Apache Cassandra списки с пропусками используются для реализации резидентной таблицы со вторичными индексами (https://databass.dev/links/81).

В системе WiredTiger списки с пропусками используются для некоторых операций в памяти.

Доступ к диску

Поскольку большая часть содержимого таблиц находится на диске, а устройства хранения обычно позволяют осуществлять доступ к данным поблочно, многие реализации LSM-дерева используют кэш страниц для доступа к диску и промежуточного кэширования. Многие методы, описанные в разделе «Организация буферизации данных» на с. 98, такие как вытеснение страниц и закрепление, также применимы и к журналированному хранилищу.

Наиболее заметным отличием является то, что содержимое в памяти неизменяемо и поэтому не требует дополнительных блокировок или защелок для параллельного доступа. Подсчет ссылок применяется, чтобы не допустить вытеснения из памяти страниц, к которым в текущий момент осуществляется доступ, и чтобы текущие запросы были выполнены до удаления файлов, лежащих в основе структуры, во время уплотнения.

Другое отличие состоит в том, что записи данных в LSM-деревьях не обязательно выравнены по страницам, а указатели могут быть реализованы для адресации с использованием абсолютных смещений, а не идентификаторов страниц. На рис. 7.9 показаны записи с содержимым, не выровненным по дисковым блокам. Некоторые записи выходят за границы страниц и требуют загрузки в память нескольких страниц.

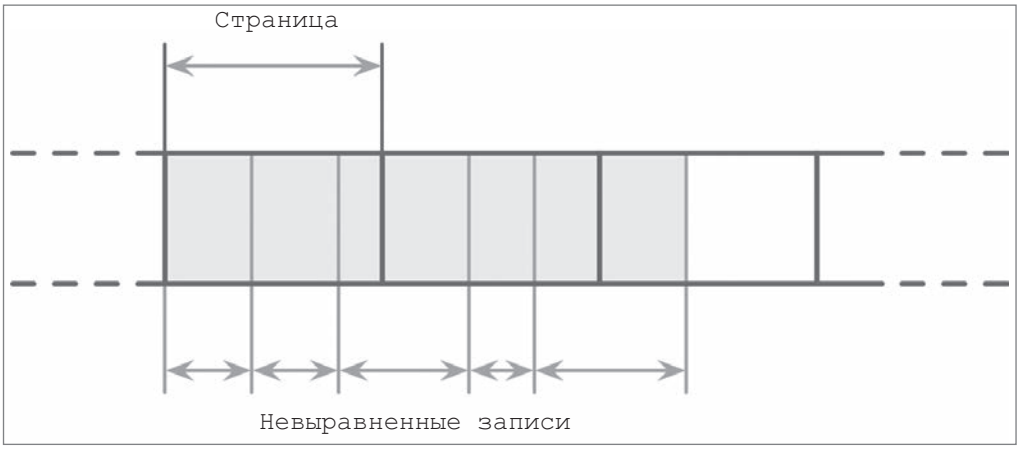

Рис. 7.9. Невыровненные записи данных

Сжатие

Мы уже обсуждали сжатие в контексте B-деревьев (см. раздел «Сжатие» на с. 91). Аналогичные идеи применимы и к LSM-деревьям. Основное различие здесь заключается в том, что таблицы LSM-дерева являются неизменяемыми и обычно записываются за один проход. При постраничном сжатии данных сжатые страницы не выравниваются по страницам, так как их размеры меньше, чем у несжатых.

Чтобы иметь возможность адресовать сжатые страницы, нам нужно отслеживать границы адресов при записи их содержимого. Мы могли бы заполнять сжатые страницы нулями, выравнивая их по размеру страницы, но тогда не было бы пользы от сжатия.

Для обеспечения адресуемости сжатых страниц нам нужен уровень косвенности, сохраняющий смещения и размеры сжатых страниц. На рис. 7.10 показан процесс отображения сжатых блоков на несжатые блоки. Сжатые страницы всегда меньше исходных страниц, так как в противном случае нет смысла их сжимать.

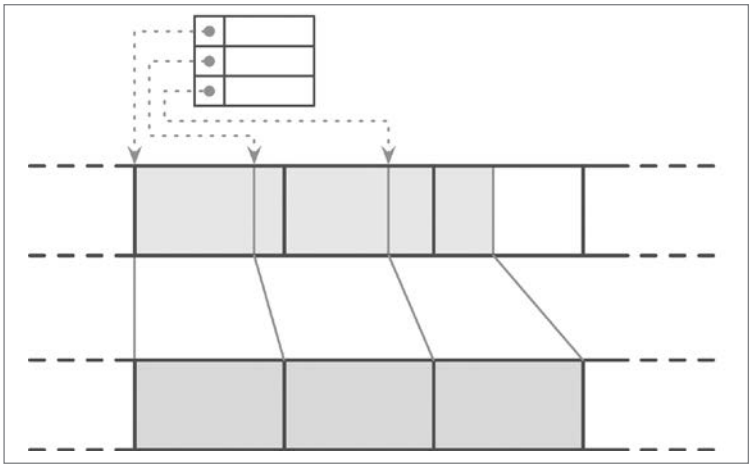

Рис. 7.10. Чтение сжатых блоков. Пунктирные линии представляют указатели из таблицы отображения на смещенное положение сжатых страниц на диске. Несжатые страницы обычно находятся в кэше страниц

Во время уплотнения и выгрузки на диск сжатые страницы добавляются последовательно, а информация о сжатии (исходное смещение несжатой страницы и фактическое смещение сжатой страницы) сохраняется в отдельном сегменте файла. Во время чтения ищутся смещение сжатой страницы и ее размер, после чего страница может быть распакована и размещена в памяти.

Неупорядоченное LSM-хранилище

Большинство рассмотренных до сих пор структур хранения хранят данные *в упорядоченном виде*. В изменяемых и неизменяемых страницах B-деревьев, в отсортированных рядах FD-деревьев и в отсортированных строковых таблицах LSM-деревьев записи данных хранятся в соответствии с порядком ключей. Порядок в этих структурах поддерживается по-разному: страницы B-дерева обновляются на месте, ряды FD-деревьев создаются путем слияния содержимого двух рядов, а отсортированные строковые таблицы создаются путем буферизации и сортировки записей данных в памяти.

В этом разделе мы рассмотрим структуры, хранящие записи в произвольном порядке. Неупорядоченные хранилища, как правило, не требуют отдельного журнала и позволяют нам снизить затраты на операции записи за счет сохранения записей данных в порядке вставки.

Bitcask

Bitcask (https://databass.dev/links/82), одна из подсистем хранения, используемых в СУБД Riak (https://databass.dev/links/83), является неупорядоченным журналированным хранилищем [SHEEHY10b]. В отличие от рассмотренных выше реализаций

журналированного хранилища, она *не* использует резидентные таблицы для буферизации и хранит записи данных непосредственно в файлах журналов.

Чтобы сделать значения доступными для поиска, в Bitcask используется структура данных *keydir*, которая содержит ссылки на *последние* записи данных с соответствующими ключами. Старые записи данных могут по-прежнему присутствовать на диске, но на них нет ссылок из keydir и они удаляются при сборке мусора в процессе уплотнения. Структура keydir реализуется как резидентная хэш-карта и воссоздается заново из файлов журнала во время запуска.

При выполнении операции *записи* ключ и запись данных последовательно добавляются в файл журнала, а в структуру keydir помещается указатель на позицию вновь сделанной записи данных.

Операции чтения просматривают структуру keydir, чтобы найти искомый ключ, и переходят по соответствующему указателю к файлу журнала, находя таким образом запись данных. Поскольку в любой момент у ключа может быть только одно связанное с ним значение в структуре keydir, точечные запросы не должны приводить к слиянию данных из нескольких источников.

На рис. 7.11 показан процесс отображения ключей на записи в файлах данных в подсистеме хранения Bitcask. Файлы журнала содержат записи данных, а структура keydir указывает на последнюю актуальную запись данных, связанную с каждым ключом. Затененные записи в файлах данных (замещаемые последующими операциями записи или удаления) выделены серым цветом.

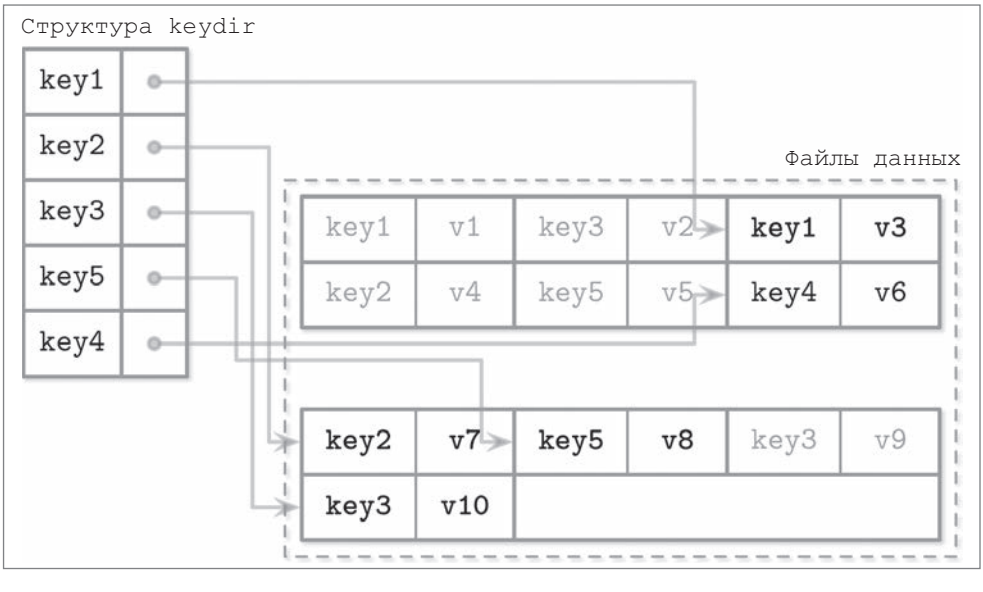

Рис. 7.11. Отображение структуры keydir на файлы данных в подсистеме хранения Bitcask. Сплошные линии представляют указатели от ключа к последнему связанному с ним значению. Затененные пары «ключ–значение» выделены светло-серым цветом

Во время уплотнения содержимое всех файлов журнала последовательно считывается, подвергается слиянию и записывается в новое место. Сохраняются только текущие записи данных, а затененные — отбрасываются. Структура keydir обновляется новыми указателями на перемещенные записи данных.

Записи данных хранятся непосредственно в файлах журналов, поэтому вести отдельный журнал упреждающей записи не требуется, что уменьшает и объем занимаемого пространства, и показатель увеличения объема записи. Недостатком этого подхода является то, что он предлагает только точечные запросы и не позволяет сканировать диапазон, так как элементы неупорядочены и в структуре keydir, и в файлах данных.

Преимуществами такого подхода являются простота и высокая производительность точечных запросов. Несмотря на то что существует несколько версий записей данных, только последняя версия адресуется структурой keydir. Однако необходимость хранить все ключи в памяти и заново воссоздавать структуру keydir при запуске является ограничением, которое может стать решающим отрицательным фактором для некоторых сценариев использования. Хотя этот подход отлично подходит для точечных запросов, он не поддерживает запросы диапазона.

WiscKey

Для многих сфер применения важны запросы диапазона, и было бы здорово иметь структуру хранения, которая обладала бы преимуществами по объему записи и использованию пространства, свойственными неупорядоченному хранению, в то же время позволяя нам выполнять сканирование диапазона.

Хранилище WiscKey [LU16] отделяет сортировку от сборки мусора за счет размещения ключей в отсортированном виде в LSM-деревьях с размещением записей данных в неупорядоченных файлах, доступных только для добавления, называемых «журналами значений» (*vLog*). Такой подход может убрать две проблемы, упомянутые при обсуждении подсистемы Bitcask: необходимость в сохранении всех ключей в памяти и в воссоздании хэш-таблицы при запуске.

На рис. 7.12 показаны ключевые компоненты хранилища WiscKey и процесс отображения ключей на файлы журнала. Файлы vLog содержат неупорядоченные записи данных. Ключи хранятся в отсортированных LSM-деревьях, указывая на последние записи данных в файлах журнала.

Поскольку ключи обычно намного меньше, чем связанные с ними записи данных, их уплотнение гораздо эффективнее. Такой подход может быть особенно полезен в случаях с низкой частотой обновлений и удалений, когда сборка мусора освобождает не так уж много места на диске.

Основная проблема здесь заключается в том, что, поскольку данные файлов vLog неотсортированы, сканирование диапазона требует произвольного ввода-вывода. В хранилище WiscKey используется внутренний параллелизм твердотельных накопителей для параллельной предварительной выборки блоков во время сканирования диапазона и снижения затрат на произвольный ввод-вывод. С точки зрения

поблочной передачи затраты все равно остаются высокими: чтобы извлечь одну запись данных во время сканирования диапазона, необходимо прочитать всю страницу, на которой она находится.

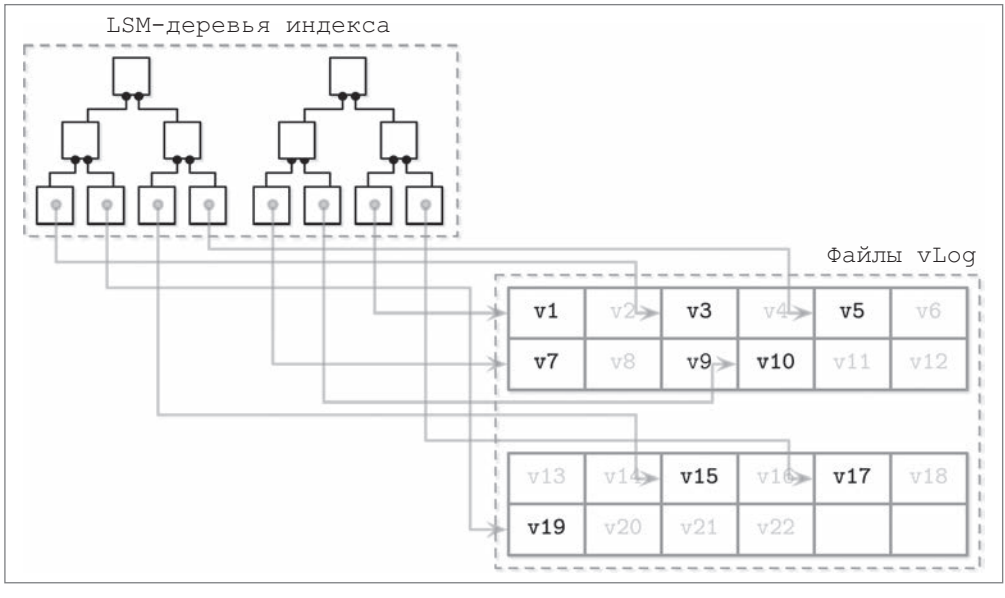

Рис. 7.12. Ключевые компоненты хранилища WiscKey: LSM-деревья индекса и файлы vLog, а также связи между ними. Затененные записи в файлах данных (замещаемые последующими операциями записи или удаления) выделены серым цветом. Сплошные линии представляют собой указатели от ключа в LSM-дереве к последнему значению в файле журнала

Во время уплотнения содержимое файла vLog последовательно считывается, подвергается слиянию и записывается в новое место. Указатели (значения в LSM-дереве с ключами) обновляются таким образом, чтобы указывать на эти новые позиции. Чтобы избежать сканирования всего содержимого файла vLog, в хранилище WiscKey используются указатели на голову и хвост, содержащие информацию о сегментах файла vLog, которые содержат актуальные ключи.

Поскольку данные в файле vLog неотсортированы и не содержат информации об актуальности, то, чтобы определить, какие значения все еще актуальны, необходимо просканировать дерево ключей. Выполнение этих проверок во время сборки мусора создает дополнительную сложность — традиционные LSM-деревья могут находить содержимое файла во время уплотнения без обращения к индексу с ключами.

Параллелизм в LSM-деревьях

Основные проблемы параллелизма в LSM-деревьях связаны с переключением *представлений таблиц* (коллекций резидентных и дисковых таблиц, которые изменяются

во время выгрузки и уплотнения) и с синхронизацией журналов. Доступ к резидентным таблицам также обычно осуществляется одновременно (за исключением секционированных в своей основе хранилищ, таких как ScyllaDB), но рассмотрение параллельных резидентных структур данных выходит за рамки этой книги.

Во время выгрузки на диск необходимо соблюдать следующие правила:

- Новая резидентная таблица должна стать доступной для чтения и записи.
- Прежняя (выгружаемая) резидентная таблица должна оставаться видимой для операций чтения.
- Выгружаемая резидентная таблица должна быть записана на диск.
- Операции удаления выгружаемой резидентной таблицы и создания выгруженной дисковой таблицы должны выполняться как атомарные операции.
- Необходимо удалить сегмент журнала упреждающей записи, который содержит записи журнала, относящиеся к операциям, примененным к выгружаемой резидентной таблице.

Например, в СУБД Apache Cassandra эти проблемы решаются с помощью барьеров порядка операций (https://databass.dev/links/84): перед началом выгрузки резидентной таблицы будет ожидаться завершение всех операций, принятых для записи. Таким образом, процесс выгрузки (выступающий в роли потребителя) знает, какие другие процессы (выступающие в роли производителей) зависят от него.

В общем случае мы имеем следующие точки синхронизации:

Переключение резидентной таблицы

После этого все операции записи производятся только в новой резидентной таблице, которая становится основной, в то время как прежняя таблица остается доступной для чтения.

Завершение выгрузки

В представлении таблицы прежняя резидентная таблица замещается выгруженной дисковой таблицей.

Усечение журнала упреждающей записи

Удаляется сегмент журнала, содержащий записи, относящиеся к выгружаемой резидентной таблице.

Из этого можно сделать серьезные выводы в отношении корректности данных. Продолжение записи в прежнюю резидентную таблицу может привести к потере данных, например, в случае записи в уже выгруженный раздел резидентной таблицы. Аналогично если не оставить прежнюю резидентную таблицу доступной для чтения до того момента, когда будет готова ее дисковая версия, то это приведет к получению неполных результатов.

Во время уплотнения представление таблицы также изменяется, но этот процесс несколько проще: прежние таблицы на диске удаляются, а вместо них добавляется

уплотненная версия. Старые таблицы должны оставаться доступными для чтения до тех пор, пока новая таблица не будет полностью записана и готова заменить их при выполнении операций чтения. Следует также не допускать участия одних и тех же таблиц в нескольких параллельных процессах уплотнения.

В случае B-деревьев, чтобы гарантировать долговечность, усечение журнала необходимо координировать с удалением «грязных» страниц из кэша страниц. В случае LSM-деревьев присутствует аналогичное требование: операции записи буферизуются в резидентной таблице, и их содержимое становится долговечным лишь после полного завершения выгрузки, поэтому усечение журнала должно координироваться с выгрузкой резидентной таблицы. По завершении выгрузки диспетчеру журнала предоставляется информация о последнем выгруженном сегменте журнала, после чего его содержимое может быть безопасно удалено.

Отсутствие синхронизации между операциями усечения журнала и выгрузки также приведет к потере данных: если сегмент журнала будет удален до завершения выгрузки и произойдет сбой узла, то содержимое журнала не будет восстановлено и нельзя будет восстановить данные этого сегмента.

Многоуровневое совмещение журналов

Многие современные файловые системы имеют структуру журнала: они буферизуют операции записи в сегменте памяти и после его заполнения выгружают его содержимое на диск с доступом только для добавления. Твердотельные накопители также используют журналированное хранилище, чтобы справляться с мелкими операциями произвольной записи, минимизировать затраты на запись, эффективнее выполнять выравнивание нагрузки и повысить срок службы устройства.

Журналированные хранилища (log-structured storage, LSS) начали набирать популярность по мере того, как твердотельные накопители становились более доступными. LSM-деревья и твердотельные накопители хорошо сочетаются, так как последовательные рабочие нагрузки и запись с доступом только для добавления помогают ослабить эффект роста затрат от обновлений на месте, негативно влияющий на производительность твердотельных дисков.

При размещении нескольких журналированных систем поверх друг друга мы можем столкнуться с некоторыми из тех проблем, для решения которых используются журналированные хранилища, включая увеличение объема записи, фрагментацию и низкую производительность. Поэтому при разработке приложений необходимо как минимум принять во внимание файловую систему и слой преобразования флэш-памяти (FTL) твердотельных накопителей [YANG14].

Слой преобразования флэш-памяти (FTL)

Использование журналированного слоя отображения в твердотельных накопителях мотивировано двумя факторами: тем, что мелкие операции произвольной записи

необходимо группировать на физической странице, и тем, что твердотельный накопитель работает посредством циклов программирование-стирание. Запись производится только на предварительно *стертые* страницы. Это означает, что вы не можете *запрограммировать* страницу (т. е. произвести запись на нее), если она не пуста (или, иначе говоря, не была *стерта*).

Нельзя стереть только *одну* страницу — стирание производится *блоками* сгруппированных страниц (обычно включающими в себя от 64 до 512 страниц). На рис. 7.13 показано схематическое представление страниц, сгруппированных в блоки. Слой преобразования флэш-памяти транслирует логические адреса страниц в соответствующие физические позиции и отслеживает статус страниц (которые могут быть актуальными, удаленными или пустыми). Когда свободные страницы заканчиваются, слой преобразования флэш-памяти должен выполнить сборку мусора и стереть удаленные страницы.

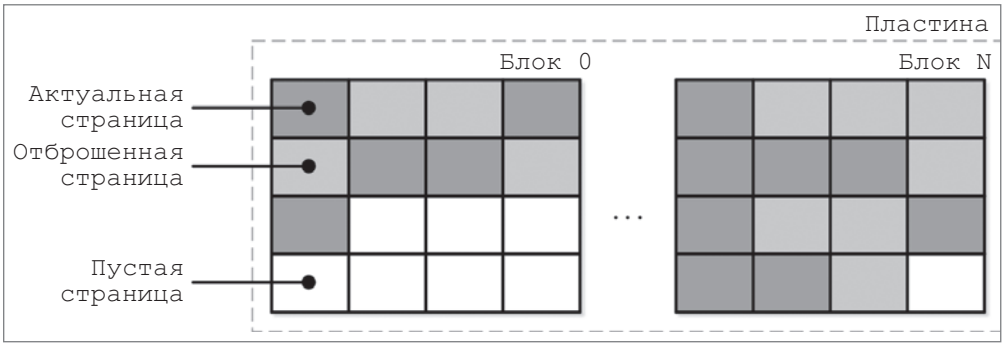

Рис. 7.13. Страницы твердотельного накопителя, сгруппированные в блоки

Нет никаких гарантий в том, что подвергаемый стиранию блок будет содержать исключительно удаленные страницы. Поэтому перед стиранием блока слой преобразования флэш-памяти должен переместить имеющиеся в нем актуальные страницы в один из блоков, содержащих пустые страницы. На рис. 7.14 показан процесс перемещения актуальных страниц из одного блока в другой.

После перемещения всех актуальных страниц блок можно безопасно стереть, после чего его пустые страницы становятся доступными для записи. Поскольку слой преобразования флэш-памяти располагает информацией о статусе страниц и изменениях статуса, а также всей другой необходимой информацией, он также отвечает за *выравнивание нагрузки* (load balancing) твердотельного накопителя.

Таким образом, журналированные хранилища используются в твердотельных накопителях для того, чтобы снизить затраты на ввод-вывод путем пакетирования мелких операций произвольной записи, что обычно приводит к уменьшению числа операций и, следовательно, снижает количество запусков сборки мусора.

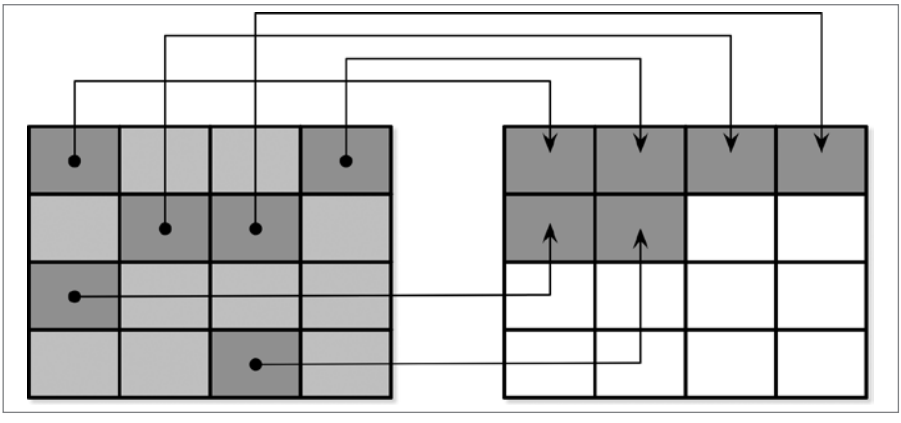

Рис. 7.14. Перемещение страницы во время сборки мусора

Процесс выравнивания нагрузки равномерно распределяет нагрузку по всему накопителю, исключая образование «горячих» точек, в которых блоки преждевременно выходят из строя из-за большого числа циклов программирования-стирания. Необходимость в этом обусловлена тем, что ячейки флэш-памяти рассчитаны только на ограниченное число циклов программирования-стирания, и равномерное использование ячеек помогает продлить срок службы устройства.

Журналирование файловой системы

Поверх всего этого у нас располагаются файловые системы, многие из которых также используют методы журналирования для буферизации операций записи, чтобы ослабить эффект увеличения объема записи и оптимально использовать нижележащее оборудование.

Многоуровневое совмещение журналов проявляется в нескольких аспектах. Во-первых, каждому слою необходимо вести свой собственный учет и обычно нижележащий журнал не раскрывает информацию, необходимую для того, чтобы избежать дублирования работы.

На рис. 7.15 показан случай, когда отображение журнала более высокого уровня (например, уровня приложения) на журнал более низкого уровня (например, уровня файловой системы) ведет к избыточному журналированию и разным схемам сборки мусора [YANG14]. Запись сегментов без выравнивания может еще больше ухудшить положение, так как удаление сегмента журнала более высокого уровня может привести к фрагментации и перемещению частей соседних сегментов.

Поскольку слои не уведомляют процесс планирования журналированного хранилища о своих действиях (например, об удалении или перемещении сегментов), подсистемы более низкого уровня могут выполнять избыточные операции над удаленными или запланированными к удалению данными. Аналогично, поскольку нет единого

стандартного размера сегмента, может случиться так, что невыровненные сегменты более высокого уровня будут занимать несколько сегментов более низкого уровня. Все эти издержки можно сократить или полностью исключить.

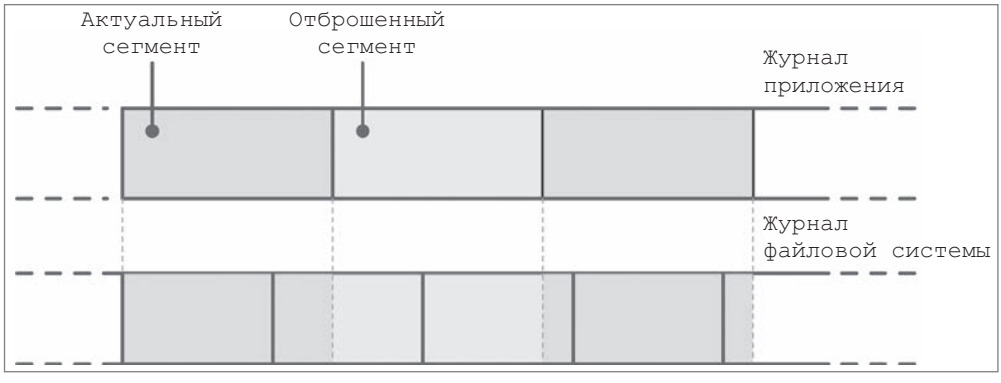

Рис. 7.15. Записи без выравнивания и удаление сегмента журнала более высокого уровня

Несмотря на то что считается, что журналированное хранилище призвано главным образом обеспечить последовательный ввод-вывод, нужно иметь в виду, что в СУБД может быть несколько потоков записи (например, может производиться параллельная запись журнала и записей данных) [YANG14]. Чередующиеся последовательные потоки записи могут не преобразоваться в такую же последовательную схему на аппаратном уровне: блоки не всегда будут располагаться в порядке записи. На рис. 7.16 показано, как несколько накладывающихся во времени потоков делают записи, размеры которых не совпадают с размером базовой аппаратной страницы.

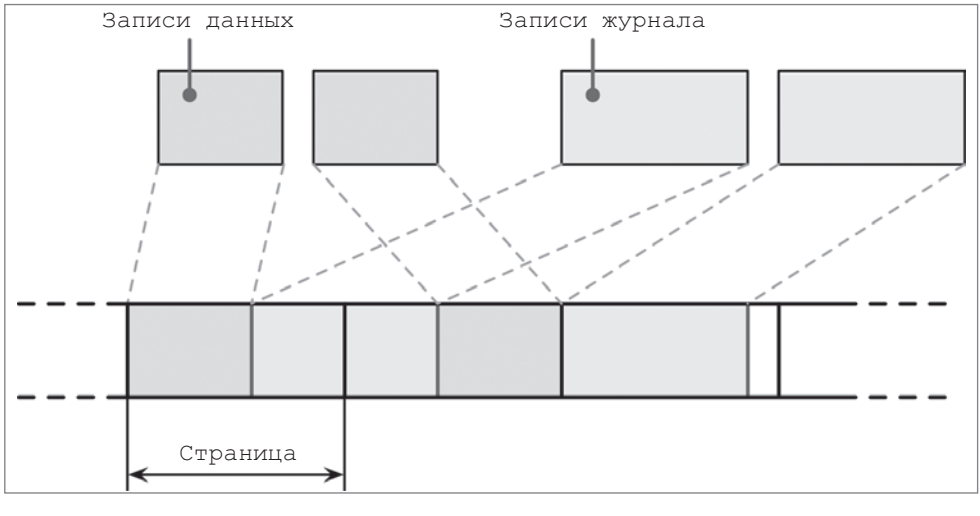

Рис. 7.16. Невыровненные многопоточные операции записи

Все это ведет к фрагментации, которой мы пытались избежать. Чтобы уменьшить степень чередования потоков, некоторые поставщики баз данных рекомендуют размещать журнал на отдельном устройстве, что позволяет изолировать рабочие нагрузки и производить независимую оценку их производительности и схем доступа. Однако более важно, чтобы разделы были выравнены в соответствии с базовым оборудованием [INTEL14], а операции записи — в соответствии с размером страницы [KIM12].

LLAMA и тщательное многоуровневое совмещение

> Ну, вы никогда не поверите, но та лама, на которую вы смотрите, когда-то была человеком. И не просто каким-то человеком. Этот парень был императором. Весь полный харизмы.
>
> *Куско из «Похождений императора»*

В разделе «Bw-деревья» на с. 138 мы обсуждали неизменяемую версию B-дерева, называемую Bw-деревом. Bw-дерево располагается поверх подсистемы хранения типа LLAMA (latch-free, log-structured, access-method aware — без защелок, журналированная, учитывающая метод доступа). Такое разделение на слои позволяет Bw-деревьям расти и сжиматься динамически, оставляя сборку мусора и управление страницами прозрачными для дерева. Здесь нас больше всего интересует *учет метода доступа*, демонстрирующий преимущества координации между слоями программного обеспечения.

Напомним, что *логический* узел Bw-дерева состоит из связанного списка физических дельта-узлов (цепочки обновлений от самого нового до самого старого), за которым следует базовый узел. Логические узлы связаны с помощью резидентной таблицы отображения, указывающей положение последнего обновления на диске. Ключи и значения добавляются и удаляются из логических узлов, но их физические представления остаются неизменными.

Журналированное хранилище буферизует обновления узлов (дельта-узлы) в буферах выгрузки по 4 Мб. Как только страница заполняется, она выгружается на диск. Периодически сборщик мусора высвобождает пространство, занимаемое неиспользуемыми дельта-узлами и базовыми узлами, и перемещает актуальные узлы, чтобы освободить фрагментированные страницы.

Без учета метода доступа перемежающиеся дельта-узлы, принадлежащие разным логическим узлам, будут записываться в порядке их вставки. Учет устройства Bw-дерева в подсистеме LLAMA позволяет консолидировать несколько дельта-узлов в одно непрерывное физическое место. Если два обновления в дельта-узлах *отменяют* друг друга (как в случае вставки с последующим удалением), также может быть выполнена их *логическая* консолидация с сохранением на персистентном носителе только последней операции удаления.

Сборщик мусора журналированного хранилища также может позаботиться о консолидации содержимого логических узлов Bw-дерева. Это означает, что сборка мусора

будет не только высвобождать свободное пространство, но и существенно снижать степень фрагментации физических узлов. Если бы сборщик мусора лишь последовательно перезаписывал несколько дельта-узлов, они все равно занимали бы такой же объем пространства и операциям чтения приходилось бы применять обновления дельта-узлов к базовому узлу. В то же время если бы система более высокого уровня консолидировала узлы и записывала их последовательно в новые места, журналированному хранилищу все равно приходилось бы удалять старые версии с помощью сборки мусора.

Зная семантику Bw-дерева, можно перезаписывать несколько дельт как один базовый узел, применяя все дельты во время сборки мусора. Это позволит уменьшить общий объем пространства, используемого для представления этого узла Bw-дерева, а также время задержки, необходимое для чтения страницы при высвобождении пространства, занятого удаленными страницами.

Можно увидеть, что при вдумчивой реализации многоуровневое совмещение может дать целый ряд преимуществ. Необязательно всегда создавать тесно связанные одноуровневые структуры. Хорошие API в сочетании с правильным экспонированием информации могут обеспечить серьезное повышение эффективности.

Твердотельные накопители с открытым каналом

Альтернативой многоуровневому совмещению программных слоев является пропуск всех уровней косвенности с использованием аппаратного обеспечения напрямую. Например, можно исключить использование файловой системы и слоя преобразования флэш-памяти, выполнив разработку с расчетом на твердотельные накопители с открытым каналом. Таким образом мы можем избежать по крайней мере двух слоев журналирования и получить больший контроль над выравниванием нагрузки, сборкой мусора, размещением данных и планированием. Примером реализации, использующей такой подход, является LOCS (хранилище типа «ключ–значение» на основе LSM-дерева на твердотельных накопителях с открытым каналом) [ZHANG13]. Еще одним примером использования твердотельных накопителей с открытым каналом является интерфейс LightNVM, реализованный в ядре Linux [BJØRLING17].

Слой преобразования флэш-памяти, как правило, отвечает за размещение данных, сборку мусора и перемещение страниц. Твердотельные накопители с открытым каналом предоставляют доступ к своим внутренним компонентам, управлению накопителем и планированию ввода-вывода без необходимости в использовании слоя преобразования флэш-памяти. Хотя это, безусловно, требует от разработчика гораздо большего внимания к деталям, такой подход может привести к значительному повышению производительности. Здесь можно провести аналогию с использованием флага O_DIRECT для обхода кэша страниц ядра, что обеспечивает бóльшую степень контроля, но требует ручного управления страницами.

Программно-определяемая флэш-память (Software defined flash, SDF) [OUYANG14], представляющая собой аппаратно-программную систему на базе твердотельных накопителей с открытым каналом, экспонирует асимметричный интерфейс ввода-вы-

вода, учитывающий специфику твердотельных накопителей. Размеры блоков чтения и записи различны, а размер блока записи соответствует размеру блока стирания, что значительно снижает показатель увеличения объема записи. Такая система идеально подходит для журналированного хранилища, поскольку существует только один программный уровень, который выполняет сборку мусора и перемещение страниц. Кроме того, разработчики имеют доступ к внутреннему параллелизму твердотельного накопителя, так как каждый канал в программно-определяемой флэш-памяти доступен как отдельное блочное устройство, что можно использовать для дальнейшего повышения производительности.

Сокрытие сложности за простым API кажется очень интересным решением, но может вызвать трудности в случаях, когда программные слои имеют различную семантику. Для улучшения интеграции часто полезно экспонировать только *некоторые* внутренние элементы нижележащей системы.

Итоги

Журналированные хранилища используются повсеместно, начиная со слоя преобразования флэш-памяти и заканчивая файловыми системами и СУБД. Они помогают снизить показатель увеличения объема записи за счет пакетирования в памяти мелких произвольных операций записи. Чтобы высвобождать место, занятое удаленными сегментами, журналированное хранилище периодически запускает сборку мусора.

LSM-деревья заимствуют некоторые идеи журналированного хранилища, позволяя создавать индексные структуры, управляемые наподобие журналов: операции записи группируются в памяти и выгружаются на диск, а затененные записи данных стираются во время уплотнения.

Важно помнить, что многие программные слои используют журналированные хранилища, и обеспечивать оптимальное многоуровневое совмещение слоев. В качестве альтернативы можно действовать в обход уровня файловой системы, обращаясь к аппаратному обеспечению напрямую.

Дополнительная литература

Если вы хотите узнать больше о концепциях, упомянутых в этой главе, то можете обратиться к следующим источникам:

Общая информация

Luo, Chen, and Michael J. Carey. 2019. “LSM-based Storage Techniques: A Survey.” The VLDB Journal. https://doi.org/10.1007/s00778-019-00555-y.

LSM-деревья

O’Neil, Patrick, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. 1996. “The log-structured merge-tree (LSM-tree).” Acta Informatica 33, no. 4: 351–385. https://doi.org/10.1007/s002360050048.

Bitcask

Justin Sheehy, David Smith. “Bitcask: A Log-Structured Hash Table for Fast Key/Value Data.” 2010.

WiscKey

Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. https://doi.org/10.1145/3033273, 1, Article 5 (March 2017), 28 pages.

LOCS

Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. https://doi.org/10.1145/2592798.2592804. In Proceedings of the Ninth European Conference on Computer Systems (EuroSys ’14). ACM, New York, NY, USA, Article 16, 14 pages.

LLAMA

Justin Levandoski, David Lomet, and Sudipta Sengupta. 2013. http://www.vldb.org/pvldb/vol6/p877-levandoski.pdf. Endow. 6, 10 (August 2013), 877–888.

ЧАСТЬ I

Заключение

В части I мы говорили о подсистемах хранения. Начав с высокоуровневой архитектуры и классификации СУБД, мы узнали, как реализуются дисковые структуры хранения и как они вписываются в общую картину с другими компонентами.

Мы познакомились с несколькими видами структур хранения, начав с B-деревьев. Наш обзор включал в себя далеко не все виды структур — существует и много других интересных разработок. Однако эти примеры все же являются хорошей иллюстрацией трех свойств, которые мы выделили в начале этой части: *буферизации*, *неизменяемости* и *упорядоченности*. Эти свойства полезны для описания, запоминания и иллюстрации различных аспектов структур хранения.

На рис. I.1 представлена сводная информация о рассмотренных видах структур хранения и указано, обладают ли они этими свойствами.

Использование резидентных буферов всегда положительно влияет на показатель увеличения объема записи. В структурах с обновлением на месте, таких как WiredTiger и LA-деревья, буферизация в памяти помогает снизить затраты на операции записи, выполняемые на одной и той же странице, за счет их объединения. То есть буферизация помогает снизить эффект увеличения объема записи.

В неизменяемых структурах, таких как многокомпонентные LSM-деревья и FD-деревья, буферизация оказывает аналогичный положительный эффект, но за счет последующей перезаписи при перемещении данных с одного неизменяемого уровня на другой. То есть использование неизменяемости может привести к отложенному увеличению объема записи. В то же время использование неизменяемости положительно влияет на параллелизм и показатель увеличения пространства, поскольку большинство рассмотренных неизменяемых структур использует полностью заполненные страницы.

Используя неизменяемость (без буферизации), мы получаем неупорядоченные структуры хранения, такие как Bitcask и WiscKey (за исключением B-деревьев с копированием при записи, которые копируют, пересортировывают и перемещают свои страницы). Хранилище WiscKey хранит ключи в отсортированных LSM-деревьях, позволяя извлекать записи в порядке ключей, используя индекс ключей. В Bw-деревьях некоторые узлы (те, которые были консолидированы) содержат записи

данных в порядке ключей, в то время как у остальных логических узлов Bw-дерева могут иметься дельта-обновления, разбросанные по разным страницам.

	Буферизация	Изменяемость	Упорядоченность
B⁺-деревья	Нет	Да	Да
WiredTiger	Да	Да	Да
LA-деревья	Да	Да	
B-деревья с копированием при записи	Нет	Нет	Да
Двухкомпонентные LSM-деревья	Да	Нет	Да
Многокомпонентные LSM-деревья	Да	Нет	Да
FD-деревья	Да	Нет	Да
Bitcask	Нет	Нет	Нет
WiscKey	Да (1)	Нет	Да (1)
BW-деревья	Нет	Нет	Нет (2)

Рис. I.1. Буферизация, неизменяемость и упорядоченность рассмотренных структур хранения. (1) WiscKey использует буферизацию только для поддержания порядка сортировки ключей. (2) В Bw-деревьях только консолидированные узлы содержат упорядоченные записи

Как видите, эти три свойства можно использовать в различном сочетании для достижения желаемых характеристик. К сожалению, внутреннее устройство подсистемы хранения обычно предполагает компромиссы: вы увеличиваете затраты на одну операцию для уменьшения затрат на другую операцию.

С этими знаниями вы сможете более внимательно изучить код большинства современных СУБД. Ссылки на код и источники, с которых можно начать, приводятся на протяжении всей книги. Знание и понимание терминологии облегчат этот процесс.

Многие современные СУБД используют вероятностные структуры данных [FLAJOLET12] [CORMODE04], кроме того, ряд новых исследований посвящен вопросу внедрения в СУБД идей машинного обучения [KRASKA18]. Нас ждут новые направления исследований и дальнейшие перемены в отрасли, поскольку энергонезависимые хранилища с байтовой адресацией становятся все более распространенными и доступными [VENKATARAMAN11].

Знание фундаментальных концепций, описанных в этой книге, должно помочь вам в понимании и применении новых разработок, поскольку эти концепции лежат в их основе или послужили источником вдохновения или идей для этих разработок. Главное преимущество знания теории и истории состоит в том, что нет ничего совершенно нового и, как видно из этой книги, прогресс идет постепенно.

ЧАСТЬ II

Распределенные системы

> Распределенная система — это система, в которой сбой компьютера, о котором вы даже не подозревали, может сделать ваш собственный компьютер непригодным для использования.
>
> *Лесли Лэмпорт*

Без распределенных систем мы не смогли бы совершать телефонные звонки, переводить деньги или обмениваться информацией на больших расстояниях. Мы ежедневно используем распределенные системы. Иногда, даже не осознавая этого, любое клиент-серверное приложение является распределенной системой.

Для многих современных программных систем *вертикальное* масштабирование (масштабирование за счет запуска одного и того же программного обеспечения на более крупной и быстрой машине с бóльшим количеством ЦП, ОЗУ или более быстрыми дисками) нежизнеспособно. Более производительные машины стоят дороже, труднее поддаются замене и часто требуют специального обслуживания. Альтернативой является *горизонтальное* масштабирование — запуск программного обеспечения на нескольких машинах, подключенных по сети и работающих как единый логический объект.

Распределенные системы могут отличаться как по размеру (от нескольких до сотен машин), так и по особенностям их участников (от небольших портативных или сенсорных устройств до высокопроизводительных компьютеров).

Давно прошли времена, когда СУБД в основном работали на одном узле. Большинство современных СУБД использует несколько узлов, соединенных в кластеры, чтобы увеличить емкость хранилища, повысить производительность и доступность.

Хотя некоторые из теоретических открытий в области распределенных вычислений не новы, их практическое применение по большей части началось относительно недавно. Сегодня мы видим растущий интерес к этой области, все больше исследований и новые разработки.

Основные определения

В распределенной системе у нас есть несколько *участников* (иногда называемых *процессами, узлами* или *репликами*). Каждый участник обладает своим локальным *состоянием*. Участники общаются, обмениваясь *сообщениями* через *каналы связи* между ними.

Процессы могут запрашивать текущее время, используя *часы*, которые могут быть *логическими* или *физическими*. Логические часы реализуются с помощью некоторой разновидности монотонно растущего счетчика. Физические часы, также называемые *системными часами,* привязаны к времени физического мира и доступны через локальные для процесса средства, например через операционную систему.

Невозможно говорить о распределенных системах, не упоминая присущие им трудности, вызванные тем, что их части расположены отдельно друг от друга. Удаленные процессы обмениваются данными посредством каналов, которые могут быть медленными и ненадежными, что усложняет получение точной информации о состоянии удаленного процесса.

Бóльшая часть исследований в области распределенных систем связана с тем фактом, что ничто не является полностью надежным: в каналах связи бывают задержки, каналы могут доставить сообщения с задержкой, в другом порядке или вообще не справиться с их доставкой; процессы могут приостановиться, замедлиться, дать сбой, выйти из-под контроля или внезапно перестать отвечать.

Области параллельного и распределенного программирования имеют много общего, поскольку центральные процессоры представляют собой миниатюрные распределенные системы с каналами, обработчиками и протоколами обмена данными. Вы увидите много аналогий с параллельным программированием в разделе «Модели согласованности» на с. 239. Однако большинство примитивов параллельного программирования нельзя напрямую использовать в распределенном программировании в силу затрат на обмен данными между удаленными участниками и ненадежности каналов и процессов.

Чтобы преодолеть трудности работы в распределенной среде, нам нужно использовать специальный класс алгоритмов — *распределенные алгоритмы*, которые используют концепции локального и удаленного состояния и выполнения и способны работать, несмотря на ненадежность сетей и сбои компонентов. Эти алгоритмы описываются в терминах *состояния* и *шагов* (или *фаз*) с *переходами* между ними. Каждый процесс выполняет шаги алгоритма локально, а комбинация локальных выполнений и взаимодействий процессов составляет распределенный алгоритм.

Распределенные алгоритмы описывают локальное поведение и взаимодействие нескольких независимых узлов. Узлы поддерживают связь, отправляя сообщения друг другу. Алгоритмы определяют роли участников, пересылаемые сообщения, состояния, переходы, выполняемые шаги, свойства среды доставки, предположения

о времени, модели отказов и другие характеристики, описывающие процессы и их взаимодействие.

Распределенные алгоритмы обеспечивают следующее:

Координация

Контролирование процессом действий и поведения нескольких рабочих процессов.

Взаимодействие

Использование участниками друга на друга для выполнения своих задач.

Распространение

Взаимодействие процессов для быстрого и надежного предоставления информации всем заинтересованным сторонам.

Консенсус

Достижение согласованности между несколькими процессами.

В этой книге мы говорим об алгоритмах в контексте их использования и предпочитаем практический подход, а не чисто академический материал. Сначала мы рассмотрим все необходимые абстракции, процессы и связи между ними, а затем перейдем к созданию более сложных моделей взаимодействия. Мы начнем с простого протокола UDP, согласно которому отправитель не получает никаких гарантий относительно того, достигнет ли его сообщение пункта назначения, а в конце рассмотрим процесс консенсуса, когда несколько процессов вырабатывают определенное согласованное значение.

ГЛАВА 8

Введение и обзор

В чем состоит неотъемлемое отличие распределенных систем от одноузловых? Давайте рассмотрим простой пример и попробуем разобраться. В однопоточной программе мы определяем переменные и процесс выполнения (набор шагов).

Например, мы можем определить переменную и выполнить над ней простые арифметические операции:

```
int i = 1;
i += 2;
i *= 2;
```

У нас есть одна история выполнения: мы объявляем переменную, увеличиваем ее на два, затем умножаем на два и получаем результат: `6`. Допустим, что вместо одного потока выполнения, выполняющего эти операции, у нас есть два потока, у которых есть доступ на чтение и запись к переменной `x`.

Конкурентное выполнение

Если два потока выполнения получат доступ к переменной и мы при этом не синхронизуем шаги между потоками, то точный результат конкурентного выполнения шага будет непредсказуемым. Вместо одного возможного результата мы получаем четыре, как показано на рис. 8.1[1].

а) `x` = 2, если оба потока считывают начальное значение, сумматор записывает свое значение, но оно перезаписывается результатом умножения.

б) `x` = 3, если оба потока считывают начальное значение, умножитель записывает свое значение, но оно перезаписывается результатом сложения.

в) `x` = 4, если умножителю удается прочитать начальное значение и выполнить свою операцию до запуска сумматора.

г) `x` = 6, если сумматору удается прочитать начальное значение и выполнить свою операцию до запуска умножителя.

Еще до выхода за пределы одного узла мы сталкиваемся с первой проблемой распределенных систем — *конкурентностью* (Concurrency). Каждая конкурентно вы-

[1] Вариант чередования, при котором умножитель производит чтение перед сумматором, для краткости опущен, поскольку дает тот же результат, что и вариант а).

полняемая программа обладает некоторыми свойствами распределенной системы. Потоки получают доступ к общему состоянию, выполняют некоторые операции локально и передают результаты обратно в общие переменные.

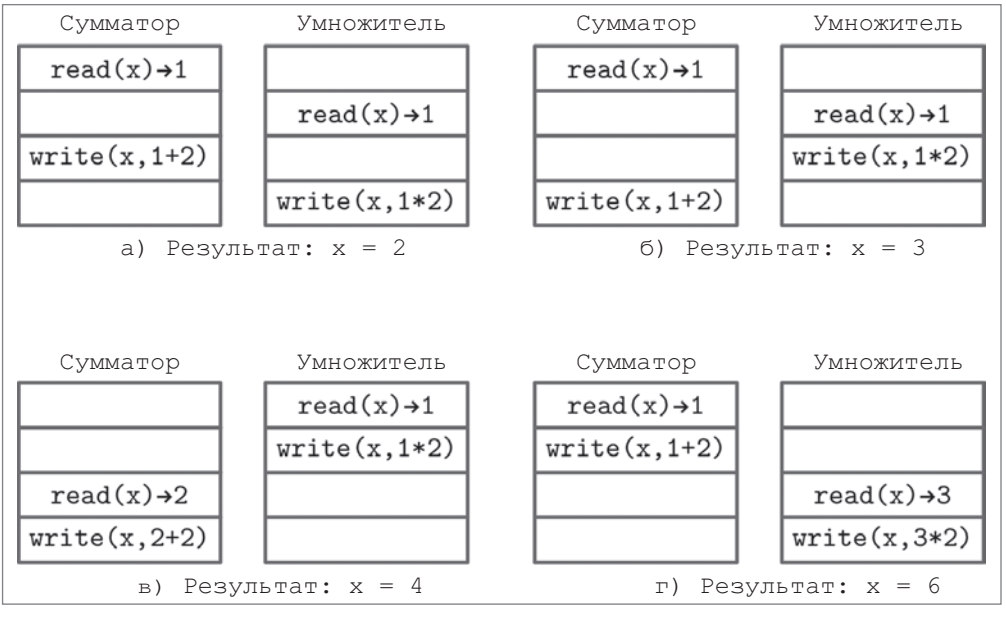

Рис. 8.1. Возможные варианты чередования при конкурентном выполнении

КОНКУРЕНТНЫЕ И ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ

Мы часто используем термины *конкурентные* и *параллельные вычисления* как синонимы, но у этих понятий есть небольшое семантическое различие. Когда две последовательности шагов выполняются конкурентно, они обе находятся в процессе выполнения, но в каждый момент выполняется только одна из них. Когда две последовательности выполняются параллельно, их шаги могут выполняться одновременно. Конкурентные операции перекрываются во времени, тогда как параллельные операции выполняются несколькими процессорами [WEIKUM01].

Джо Армстронг, создатель языка программирования Erlang, привел такой пример: конкурентное выполнение — это как две очереди к одной кофемашине, а параллельное выполнение — как две очереди на две кофемашины. Тем не менее подавляющее большинство источников использует термин *конкурентность* для описания систем с несколькими параллельными потоками выполнения, а термин *параллелизм* используется редко.

Чтобы точно определить историю выполнения и сократить количество возможных результатов, нам нужны *модели согласованности* (consistency models). Модели согласованности описывают выполнение конкурентных потоков и определяют, в каком

порядке можно выполнять операции и делать их результаты видимыми для участников. Используя различные модели согласованности, мы можем ограничить число возможных состояний системы нужным нам образом.

Области распределенных систем и конкурентных вычислений имеют много общего в части терминологии и проводимых исследований, однако есть и некоторые различия. В конкурентной системе у нас может быть *общая память* (shared memory), которую процессоры могут использовать для обмена информацией. В распределенной системе каждый процессор имеет свое локальное состояние, а участники поддерживают связь, обмениваясь сообщениями.

Общее состояние в распределенной системе

Мы можем попытаться ввести понятие общей памяти в распределенную систему, например в виде такого единого источника информации, как база данных. Однако даже если мы решим проблемы с конкурентным доступом к ней, мы все равно не сможем гарантировать, что все процессы будут синхронизированы.

Для получения доступа к этой базе данных процессы должны обмениваться сообщениями посредством коммуникационной среды для запрашивания или модификации состояния. Однако что произойдет, если один из процессов не получит ответ от базы данных в течение длительного времени? Чтобы ответить на этот вопрос, мы сначала должны определить, что означает «длительное». Для этого систему необходимо описать в терминах *синхронности* (synchrony): является ли обмен данными полностью асинхронным и имеются ли какие-либо допущения в отношении распределения времени. Допущения в отношении времени позволяют нам ввести время ожидания операций и повторные попытки.

Мы не знаем, по какой причине может не отвечать база данных: из-за того, что она перегружена, недоступна или медленно работает, или из-за проблем с сетью на пути к ней. Это определяет *характер* отказа: причиной сбоя может стать неспособность процесса принять участие в дальнейших шагах алгоритма, его временный сбой или пропуск некоторых сообщений. Прежде чем решать, как следует устранять отказы, мы должны определить *модель отказов* (failure model) и описать пути их возникновения.

Свойство, которое описывает надежность системы и ее способность продолжать корректную работу в случае отказов, называется *отказоустойчивостью* (fault tolerance). Отказы неизбежны, поэтому нам необходимо создавать системы с надежными компонентами, и первым шагом в этом направлении может стать устранение единой точки отказа в виде вышеупомянутой одноузловой базы данных. Сделать это можно, введя некоторую *избыточность* (redundancy) и добавив резервную базу данных. Однако теперь мы сталкиваемся с другой проблемой: как синхронизовать *несколько копий* общего состояния?

Пока что попытка ввести общее состояние в нашу простую систему оставила нам больше вопросов, чем ответов. Теперь мы знаем, что для совместного использования

состояний недостаточно просто добавить базу данных, и потому мы должны использовать более детальный подход, описывая взаимодействия в терминах независимых процессов и передачи сообщений между ними.

Ошибочные представления о распределенных вычислениях

В идеальном случае, когда два компьютера общаются по сети, все работает просто отлично: процесс устанавливает соединение, отправляет данные, получает ответы и все довольны. Предполагать, что операции всегда выполняются успешно и все пройдет гладко, опасно, поскольку, когда что-то ломается, а предположения оказываются неверными, поведение систем трудно или невозможно предсказать.

Разумно предположить, что бóльшую часть времени сеть работает надежно. Она и должна обладать хотя бы некоторой степенью надежности, чтобы быть пригодной для использования. Но мы все бывали в ситуации, когда пытаешься установить соединение с удаленным сервером, а вместо этого получаешь ошибку «Сеть недоступна» (Network is unreachable). Даже успешное первоначальное соединение с сервером не гарантирует стабильной работы сети — соединение может прерваться в любой момент. Возможно, сообщение дошло до удаленной стороны, но полученный ответ потерялся либо соединение было прервано еще до доставки ответа.

В любой момент может сломаться сетевой коммутатор, отсоединиться кабель или измениться конфигурация сети. При создании своей системы мы должны обеспечить устойчивость к каждому из этих сценариев.

Даже если соединение стабильно, мы не можем рассчитывать на то, что удаленные вызовы будут столь же быстрыми, как и локальные. Мы должны сделать как можно меньше предположений в отношении времени задержки, ни в коем случае не рассчитывая на то, что *задержка (latency) равна нулю*. Чтобы дойти до удаленного сервера, наше сообщение должно пройти через несколько программных слоев и такую физическую среду, как оптическое волокно или кабель. Ничто из этого не может происходить мгновенно.

Майкл Льюис в своей книге *Flash Boys*[1] рассказывает о компаниях, которые готовы потратить миллионы долларов на то, чтобы сократить время задержки хотя бы на несколько миллисекунд и тем самым обеспечить себе более быстрый доступ к фондовым биржам по сравнению с конкурентами. Это отличный пример использования времени задержки в качестве конкурентного преимущества, но стоит упомянуть, что, согласно некоторым другим исследованиям, таким как [BARTLETT16], низкая вероятность спекулятивного использования более свежей информации о котировках (получения прибыли за счет способности узнавать цены и выполнять заказы быстрее конкурентов) не позволяет быстрым трейдерам эксплуатировать рынки.

Усвоив этот урок, мы добавили повторные попытки и повторные подключения и избавились от предположения о мгновенном выполнении, но этого все же оказалось

[1] *Льюис М. Flash Boys.* Высокочастотная революция на Уолл-стрит. М.: Альпина Паблишер, 2019. — *Примеч. ред.*

недостаточно. При увеличении количества, скоростей и размеров пересылаемых сообщений или при добавлении новых процессов в существующую сеть мы не должны рассчитывать на то, что *пропускная способность (throughput) является неограниченной*.

В 1994 году Питер Дойч опубликовал широко известный в настоящее время список утверждений под названием «Ошибочные представления о распределенных вычислениях», описывающий аспекты распределенных вычислений, которые можно легко упустить из виду. В дополнение к предположениям о надежности сети, времени задержки и пропускной способности он описывает некоторые другие проблемы. Например, сетевая безопасность, возможное присутствие злоумышленников, преднамеренные и непреднамеренные изменения топологии, которые могут сделать неверными наши предположения о наличии и расположении конкретных ресурсов, транспортные издержки с точки зрения времени и ресурсов и, наконец, существование единого органа управления, имеющего информацию обо всей сети и контролирующего всю сеть.

Хотя составленный Дойчем список ошибочных представлений о распределенных вычислениях является достаточно полным, он сосредоточен на том, что может пойти не так при отправке сообщений от одного процесса другому через канал. Это вполне актуальные проблемы, включающие в себя наиболее общие и низкоуровневые трудности, но, к сожалению, при разработке и реализации распределенных систем также можно сделать много других предположений, способных привести к проблемам при эксплуатации.

Обработка

Прежде чем удаленный процесс сможет отправить ответ на только что полученное сообщение, он должен локально выполнить некоторую работу, поэтому нельзя предполагать, что *обработка* (processing) *происходит мгновенно*. Недостаточно лишь учесть сетевую задержку, поскольку операции, выполняемые удаленными процессами, также не являются мгновенными.

Более того, нет гарантии, что обработка начнется сразу после доставки сообщения. Сообщение может попасть в очередь ожидания на удаленном сервере, и ему придется ждать завершения обработки всех сообщений, поступивших до него.

Узлы могут располагаться ближе или дальше друг от друга, иметь разные ЦП, объемы ОЗУ, разные диски или использовать разные версии и конфигурации программного обеспечения. Мы не можем рассчитывать на то, что они будут обрабатывать запросы с одинаковой скоростью. Если для выполнения задачи нам нужно дождаться ответа нескольких удаленных серверов, работающих параллельно, то общая скорость выполнения будет определяться скоростью самого медленного удаленного сервера.

Вопреки широко распространенному убеждению *емкость очереди не бесконечна* и системе не принесет пользы накопление большого количества запросов. Стратегия *замедленной обратной реакции* (backpressure) — это стратегия, которая посредством

замедления работы производителей позволяет нам справляться с теми производителями, которые публикуют сообщения со скоростью, превышающей скорость, с которой потребители способны их обрабатывать. Замедленная обратная реакция — одна из мало ценимых и редко применяемых концепций в распределенных системах, в силу чего ее часто надстраивают над уже готовой системой, вместо того чтобы включить ее в состав системы заранее.

Хотя увеличение емкости очереди кажется полезной мерой, которая может помочь в конвейеризации, распараллеливании и эффективном планировании запросов, следует учесть, что с сообщениями ничего не происходит, пока они находятся в очереди. Увеличение размера очереди может отрицательно повлиять на время задержки, поскольку, внося это изменение, мы не меняем скорость обработки.

В общем случае локальные для процессов очереди используются для достижения следующих целей:

Развязка (decoupling)

Прием и обработка разделены во времени и происходят независимо друг от друга.

Конвейеризация (pipelining)

Запросы на разных этапах обрабатываются независимыми частями системы. Подсистему, отвечающую за получение сообщений, не требуется блокировать до завершения обработки предыдущего сообщения.

Амортизация кратковременных всплесков

Загрузка системы обычно непостоянна, но время поступления запросов скрыто от компонента, отвечающего за обработку запросов. Общая задержка системы увеличивается из-за времени, проведенного в очереди, но обычно это все же лучше, чем выдача в качестве ответа сообщения об ошибке с последующей необходимостью выполнять запрос повторно.

Размер очереди зависит от рабочей нагрузки и специфики приложения. В случае сравнительно стабильной рабочей нагрузки мы можем определить размер очередей, измеряя время выполнения задачи и среднее время нахождения в очереди для каждой задачи и следя за тем, чтобы задержка оставалась в допустимых пределах при увеличении объема обрабатываемых данных. В этом случае размеры очередей сравнительно невелики. В случае непредсказуемых рабочих нагрузок, когда возможны всплески в поступлении задач, размер очередей должен выбираться с расчетом на всплески и высокую нагрузку.

Хотя удаленный сервер способен быстро справляться с обработкой запросов, это совсем не значит, что мы всегда будем получать от него положительный ответ. Иногда он будет выдавать сообщение об ошибке в силу того, что ему не удастся произвести запись, не будет найдено искомое значение или будет выявлена программная ошибка. Таким образом, даже самый благоприятный сценарий все равно требует некоторого внимания с нашей стороны.

Часы и время

Время — это иллюзия. А обеденное время — тем более.

Форд Префект, «Автостопом по Галактике»

Предположение о том, что часы на удаленных машинах работают синхронно, также может таить опасность. В сочетании с предположениями о том, что задержка равна нулю, а обработка производится мгновенно, оно приводит к ряду характерных проблем, особенно при обработке временны́х рядов и данных в реальном времени. Например, при сборе и агрегировании данных от участников с различным восприятием времени вы должны учитывать смещение времени между ними и нормализовать время соответственным образом, а не полагаться на временную метку источника. Если только вы не используете специализированные высокоточные источники времени, вы не должны полагаться на временные метки при выполнении синхронизации или упорядочения. Конечно, это не означает, что мы не можем или не должны полагаться на время вообще: в конце концов, любая синхронная система использует локальные часы для отсчета времени ожидания.

Важно всегда учитывать возможную разницу во времени между процессами и время, необходимое для доставки и обработки сообщений. Например, в хранилище Spanner (см. раздел «Распределенные транзакции с использованием протокола Spanner» на с. 286) используется специальный временной API, который возвращает временную метку и границы неопределенности для фиксации строгого порядка транзакций. Некоторые алгоритмы обнаружения отказов опираются на общее время и гарантируют, что уход показаний часов всегда находится в пределах, допустимых для соблюдения корректности [GUPTA01].

Помимо того что синхронизация часов в распределенной системе является сложной задачей, текущее время постоянно меняется: вы можете запросить текущую временную метку POSIX из операционной системы и другую текущую временную метку после выполнения нескольких шагов и эти два значения будут разными. Это довольно очевидное наблюдение, однако вы должны четко понимать и то, с каким источником времени вы имеете дело, и то, в какой именно момент была создана временная метка.

Также часто полезно понимать то, является ли источник часов монотонным (т. е. неспособным идти в обратном направлении) и насколько могут сместиться запланированные связанные со временем операции.

Согласованность состояний

Большинство вышеупомянутых предположений почти попадает в категорию *всегда ложных*, но есть такие, которые лучше описать как *не всегда правдивые*: когда легко принять некоторые шаблоны мышления и упростить модель, рассматривая ее определенным образом, без учета некоторых сложных крайних случаев.

Распределенные алгоритмы не всегда гарантируют строгую согласованность состояний. Некоторые подходы подразумевают более слабые ограничения и допускают

расхождение состояний между репликами и полагаются на *разрешение конфликтов* (способность к выявлению и устранению расхождений в состояниях внутри системы) и на *восстановление данных во время чтения* (синхронизацию реплик во время чтения в тех случаях, когда они выдают различные результаты). Больше информации об этих концепциях можно найти в главе 12. Предположение о том, что состояние полностью согласовано между узлами, может привести к коварным, труднонаходимым ошибкам.

Согласованная, в конечном счете распределенная СУБД может иметь в своем составе логику для обработки расхождений между репликами путем запроса кворума узлов во время чтения, но при этом предполагается, что схема базы данных и представление кластера строго согласованы. Если мы не обеспечим согласованность этой информации, использование такого предположения может привести к серьезным последствиям.

Например, в СУБД Apache Cassandra был баг (https://databass.dev/links/46), вызванный тем, что изменения схемы распространялись на серверы в разные моменты. При попытке чтения из базы данных во время распространения схемы был риск искажения данных, поскольку один сервер кодировал результаты, используя одну схему, а другой декодировал их, используя другую схему.

Еще одним примером является программная ошибка, вызванная расходящимся представлением кольца (https://databass.dev/links/47): если один из узлов предполагает, что другой узел содержит записи данных с некоторым ключом, но этот другой узел имеет другое представление кластера, то чтение или запись этих данных может привести к неправильному размещению записей данных или получению пустого ответа, в то время как записи данных фактически успешно присутствуют в другом узле.

Лучше заранее учесть возможные проблемы, даже если полное решение затратно в реализации. Понимая эти случаи и работая над ними, вы можете встроить средства защиты или изменить устройство системы таким образом, чтобы сделать решение более естественным.

Локальное и удаленное выполнение

Скрытие сложности за API может нести риски. Например, если у вас есть итератор для локального набора данных, вы вполне можете предсказать, что происходит «за кулисами», даже если вам незнакома подсистема хранения. Понимание процесса итерации по удаленному набору данных — совершенно другая проблема: вам необходимо понимать семантику согласованности и доставки, процессы согласования данных, разбиения на страницы и слияния, влияние конкурентного доступа и многое другое.

Простое скрытие и того и другого за одним и тем же интерфейсом, как бы это ни было удобно, может ввести в заблуждение. Для отладки, настройки и обеспечения наблюдаемости могут понадобиться дополнительные параметры API. Мы всегда должны помнить, что *локальное и удаленное выполнение — это не одно и то же* [WALDO96].

Наиболее очевидной проблемой скрытия удаленных вызовов является время задержки: удаленный вызов во много раз затратнее локального, поскольку он включает в себя пересылку в двух направлениях по сети, сериализацию/десериализацию и многие другие шаги. Чередование локальных и блокировка удаленных вызовов могут привести к снижению производительности и непредусмотренным побочным эффектам [VINOSKI08].

Необходимость обработки отказов

Вы вполне можете начать работу над системой, предполагая, что все ее узлы работают нормально, но думать, что так будет происходить всегда, опасно. В долго работающей системе узлы могут отключаться для обслуживания (которое обычно подразумевает постепенное отключение) или отказывать по различным причинам: из-за проблем с программным обеспечением, нехватки памяти [KERRISK10], ошибок времени выполнения, аппаратных проблем и т. д. Отказы процессов случаются, и лучшее, что вы можете сделать, — это быть готовым к отказам и понять, как с ними справиться.

Если удаленный сервер не отвечает, мы не всегда знаем, чем именно это вызвано. Причиной может быть аварийное завершение работы, сетевой сбой, удаленный процесс или медленная связь с ним. Некоторые распределенные алгоритмы используют *протоколы на основе контрольных пакетов* (heartbeat) и *детекторы отказов* (failure detectors) для формирования предположений о том, какие участники активны и доступны.

Распад сети и частичные отказы

Когда два или более сервера не могут связаться друг с другом, мы называем такую ситуацию «распадом сети» (network partition). В исследовании *Perspectives on the CAP Theorem* [GILBERT12] («Перспективы CAP-теоремы») Сет Гилберт и Нэнси Линч проводят различие между случаем, когда два участника не могут связаться друг с другом и когда несколько групп участников изолированы друг от друга, не могут обмениваться сообщениями и продолжить выполнение алгоритма.

Общая ненадежность сети (потеря пакетов, повторное отправление, трудно предсказуемые задержки) раздражает, но допустима, в то время как распад сети может вызвать гораздо больше проблем, поскольку независимые группы могут продолжить выполнение и выдать противоречивые результаты. Также могут происходить асимметричные отказы каналов сети, когда сообщения могут по-прежнему поступать от одного процесса к другому, но не в обратном направлении.

Чтобы построить систему, которая является устойчивой в случае отказа одного или нескольких процессов, мы должны рассмотреть случаи частичных отказов [TANENBAUM06] и то, как система будет продолжать работать, даже если ее часть недоступна или работает некорректно.

Отказы трудно поддаются выявлению и не всегда видны одинаково из разных частей системы. При проектировании высокодоступных систем всегда следует принимать

во внимание пограничные случаи: что, если мы произведем репликацию данных, но не получим подтверждения? Мы должны будем повторить попытку? Будут ли данные по-прежнему доступны для чтения в узлах, отправивших подтверждения?

Закон Мерфи[1] говорит нам, что отказы случаются. Айтишный фольклор добавляет, что отказы будут происходить наихудшим из возможных способов, поэтому наша задача как разработчиков распределенных систем заключается в том, чтобы уменьшить количество сценариев, при которых что-то может пойти не так, и подготовиться к отказам таким образом, чтобы сдержать распространение ущерба, который они могут нанести.

Невозможно предотвратить все отказы, но мы все же можем построить отказоустойчивую систему, которая будет корректно функционировать при их присутствии. Лучший способ проектировать с учетом отказов — это проводить тестирование на их наличие. Почти невозможно продумать каждый возможный сценарий отказа и предсказать поведение нескольких процессов. Применение средств тестирования, способных имитировать распад сети и «гниение битов» [GRAY05], увеличение задержек, рассогласование часов и большую разницу в скорости обработки — лучший способ для этого. Реальные распределенные системы могут быть весьма «вредными», недружественными и «креативными» (в плохом смысле), поэтому при тестировании следует попытаться охватить как можно больше сценариев.

За последние несколько лет мы видели несколько проектов с открытым исходным кодом, которые позволяют воспроизводить различные сценарии отказов. Прокси-сервер Toxiproxy (https://databass.dev/links/48) может помочь в имитации проблем в сети: ограничить пропускную способность, ввести задержку, время ожидания и многое другое. Система Chaos Monkey (https://databass.dev/links/49) использует более радикальный подход и демонстрирует разработчикам возможные сбои на этапе эксплуатации, случайным образом останавливая службы. Файловая система CharybdeFS (https://databass.dev/links/50) помогает имитировать ошибки и отказы файловой системы и оборудования. Вы можете использовать эти инструменты, чтобы протестировать программное обеспечение и убедиться, что оно работает корректно при наличии этих отказов. CrashMonkey (https://databass.dev/links/122), независимая от файловой системы платформа автоматизированного тестирования типа «запись-прогон-и-тестирование» (record-replay-and-test), помогает проверить согласованность данных и метаданных для персистентных файлов.

При работе с распределенными системами мы должны серьезно относиться к отказоустойчивости, способности к восстановлению, возможным сценариям отказов и пограничным случаям. Подобно тому как «при достаточном количестве наблюдателей программные ошибки выплывают на поверхность» (https://databass.dev/links/51), мы можем сказать, что достаточно большой кластер в конце концов затронут все

[1] Закон Мерфи — это изречение, которое можно обобщить как «все, что может пойти не так, пойдет не так»; получило широкое распространение и часто используется в качестве идиомы в популярной культуре.

возможные проблемы. В то же время при условии достаточного тестирования мы сможем в конечном итоге найти все существующие проблемы.

Каскадные отказы

Полная изоляция отказа не всегда возможна: процесс, отказывающий из-за высокой нагрузки, увеличивает нагрузку на остальную часть кластера, повышая вероятность отказа других узлов. *Каскадные отказы* (cascading failures) могут распространяться от одной части системы к другой, увеличивая масштаб проблемы.

Иногда каскадные отказы вызываются благими намерениями. Например, узел некоторое время находился в автономном режиме и не получал самые последние обновления. После того как он снова появляется в сети, услужливые одноранговые узлы хотят помочь ему узнать о последних событиях и начинают потоковую передачу данных, которых ему не хватает, и исчерпывают тем самым ресурсы сети или вызывают сбой узла вскоре после запуска.

Для защиты системы от распространяющихся отказов и корректной обработки сценариев отказа можно использовать *размыкатели цепи* (circuit breakers). В электротехнике автоматические размыкатели цепи защищают дорогие и трудно заменяемые детали от перегрузки или короткого замыкания путем отключения тока. При разработке программного обеспечения размыкатели цепи выявляют отказы и запускают резервные механизмы, способные защитить систему путем отключения ее от давшей сбой службы, предоставления ей некоторого времени для восстановления и корректной обработки сбойных вызовов.

Когда не удается установить соединение с одним из серверов или сервер не отвечает, клиент запускает цикл повторных подключений. К этому моменту перегруженный сервер уже не справляется с новыми запросами на подключение, и повторные попытки на стороне клиента в коротком цикле ситуацию не улучшают. Чтобы избежать этого, мы можем использовать стратегию *отсрочки* (backoff strategy). Вместо немедленного выполнения повторной попытки клиенты ждут некоторое время. Отсрочка может помочь избежать роста проблем путем планирования повторных попыток и увеличения временного интервала между последовательными запросами.

Отсрочка используется для увеличения периодов между запросами от одного клиента. Однако разные клиенты, использующие одну и ту же стратегию отсрочки, также могут создать значительную нагрузку. Чтобы исключить возможность одновременного выполнения повторной попытки разными клиентами после периода отсрочки, мы можем ввести некоторый *разброс* (jitter). Разброс добавляет к отсрочке небольшие случайные периоды и тем самым снижает вероятность того, что клиенты проснутся и повторят попытку одновременно.

Отказы оборудования, «гниение битов» (bitrot) и программные ошибки могут привести к искажению, которое распространится дальше через стандартные механизмы доставки. Например, искаженные записи данных могут быть реплицированы на

другие узлы, если они не будут подвержены проверке. Без механизмов проверки система может распространить искаженные данные на другие узлы, что может привести к перезаписи неискаженных записей данных. Чтобы избежать этого, мы должны использовать контрольную сумму и валидацию для проверки целостности всего того содержимого, которым обмениваются узлы.

Перегрузок и горячих точек можно избежать, планируя и координируя выполнение. Вместо того чтобы позволить одноранговым узлам выполнять шаги операции независимо, мы можем использовать координатор, который будет составлять план выполнения на основе доступных ресурсов и прогнозировать нагрузку на основе доступных данных о выполнении предыдущих операций.

Таким образом, мы всегда должны рассматривать случаи, когда отказы в одной части системы могут вызвать проблемы в других местах. Мы должны снабдить свои системы автоматическими размыкателями цепи, механизмами отсрочки, проверки и координации. Обработать небольшие изолированные проблемы всегда проще, чем пытаться восстановиться после большого отказа.

Мы посвятили целый раздел обсуждению проблем и возможных сценариев отказов в распределенных системах, но необходимо рассматривать это как предупреждение, а не как нечто, что должно нас пугать.

Понимание того, что может пойти не так, а также тщательные проектирование и тестирование наших систем делают их более надежными и устойчивыми. Знание этих аспектов может помочь вам определить и найти потенциальные источники проблем во время разработки, а также устранить неисправности в ходе эксплуатации.

Абстракции распределенных систем

Говоря о языках программирования, мы используем общепринятую терминологию и описываем наши программы в терминах функций, операторов, классов, переменных и указателей. Наличие общепринятого словаря помогает не придумывать новые слова каждый раз, когда мы что-либо описываем. Чем точнее и менее двусмысленны наши определения, тем легче слушателям понять нас.

Прежде чем перейти к алгоритмам, сначала необходимо рассмотреть словарь распределенных систем: определения, с которыми вы часто будете встречаться в беседах, книгах и газетах.

Каналы

Сети ненадежны: сообщения могут потеряться, задержаться и оказаться переупорядоченными. Теперь, учитывая это, мы попытаемся создать несколько протоколов обмена данными. Мы начнем с наименее надежных и устойчивых, определив состояния, в которых они могут находиться, а затем подумаем о возможных дополнениях к протоколам, которые могут усилить их гарантии.

Канал с допустимыми потерями

Мы можем начать с двух *процессов*, связанных с *каналом*. Процессы могут отправлять сообщения друг другу, как показано на рис. 8.2. Любая среда связи несовершенна, и сообщения могут потеряться или задержаться.

Посмотрим, какие гарантии мы можем получить. После отправки сообщения `M`, с точки зрения отправителя, оно может находиться в одном из следующих состояний:

- *Еще* не доставлено процессу `Б` (но будет в какой-то момент).
- Безвозвратно утеряно во время транспортировки.
- Успешно доставлено удаленному процессу.

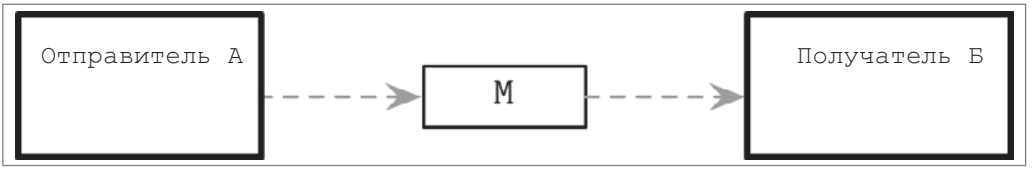

Рис. 8.2. Простейшая, ненадежная форма связи

Обратите внимание, что у отправителя нет возможности узнать, доставлено ли уже сообщение. В терминологии распределенных систем этот вид связи называется каналом с допустимыми потерями (fair-loss). Свойства этого типа канала следующие:

Допустимые потери

Если отправитель и получатель работают корректно и отправитель продолжает повторять передачу сообщения бесконечно много раз, оно в конечном итоге будет доставлено[1].

Ограниченное дублирование (finite duplication)

Отправленные сообщения не будут доставляться бесконечное количество раз.

Без создания (no creation)

Сообщения не возникают в канале; другими словами, канал не доставит сообщение, которое не отправлялось.

Канал с допустимыми потерями является полезной абстракцией и первым строительным блоком для протоколов обмена данными с сильными гарантиями. Мы можем предположить, что в таком канале отсутствуют *систематические* потери сообщений между связывающимися сторонами и не создаются новые сообщения. Но в то же время нельзя полностью полагаться на это предположение. Такой канал может напомнить вам протокол пользовательских датаграмм (UDP, https://databass.

[1] Более точное определение: если корректный процесс `A` отправляет сообщение корректному процессу `Б` бесконечное число раз, то оно будет доставлено бесконечное число раз ([CACHIN11]).

dev/links/52), который позволяет отправлять сообщения от одного процесса другому, но не предлагает надежной семантики доставки на уровне протокола.

Подтверждение сообщения

Чтобы улучшить ситуацию и получить больше ясности в отношении статуса сообщения, мы можем ввести *подтверждения* (acknowledgement): способ для получателя уведомить отправителя о том, что он получил сообщение. Для этого нам нужно использовать двунаправленные каналы связи и добавить некоторые средства, которые позволяют различать сообщения между собой; например, по *порядковым номерам* (sequence number), представляющим собой уникальные монотонно увеличивающиеся идентификаторы сообщений.

Теперь процесс `A` может отправить сообщение `M(n)`, где `n` — монотонно увеличивающийся счетчик сообщений. Как только процесс `Б` получает сообщение, он отправляет подтверждение `ACK(n)` обратно процессу `A`. Такая форма обмена данными показана на рис. 8.3.

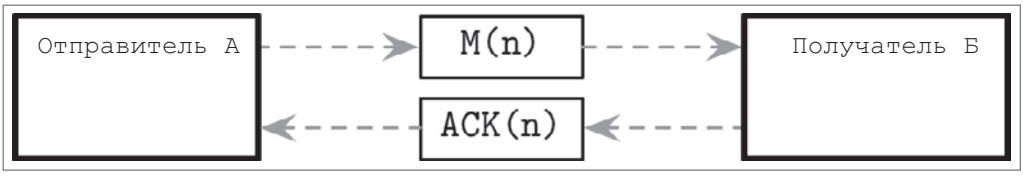

Рис. 8.3. Отправка сообщения с подтверждением

Достаточно присваивать *уникальный* идентификатор каждому сообщению. Порядковые номера являются частным случаем уникального идентификатора, когда мы достигаем уникальности, вырабатывая идентификаторы посредством счетчика. При использовании для уникальной идентификации сообщений алгоритмов хеширования необходимо учесть возможность возникновения конфликтов и проследить за тем, чтобы даже в таком случае можно было однозначно идентифицировать сообщения.

Подтверждение, а также исходное сообщение могут потеряться в пути. Количество состояний, в которых может находиться сообщение, изменяется незначительно. Пока процесс `A` не получит подтверждение, сообщение находится в одном из трех состояний, о которых мы упоминали ранее, но как только процесс `A` получает подтверждение, он может быть уверен, что сообщение доставлено процессу `Б`.

Повторная передача сообщения

Добавления подтверждений все еще недостаточно, чтобы назвать этот протокол обмена данными надежным: отправленное сообщение все равно может потеряться или удаленный процесс может отказать до отправки подтверждения. Для решения этой проблемы и предоставления гарантий доставки вместо использования подтверждений мы можем попробовать повторно отправить сообщение. Повторная пере-

дача — это способ для отправителя повторить потенциально неудачную операцию. Мы говорим «*потенциально* неудачную», потому что отправитель на самом деле не знает, насколько удачно завершилась доставка, поскольку обсуждаемый здесь тип канала не использует подтверждения.

После того как процесс `A` отправляет сообщение `M`, он ждет, пока не истечет время ожидания `T`, и пытается отправить то же сообщение еще раз. Предполагая, что канал между процессами остается неповрежденным, распад сети между процессами не абсолютен и не все пакеты потеряны, мы можем утверждать, что, с точки зрения отправителя, сообщение либо еще не доставлено процессу `Б`, либо успешно доставлено процессу `Б`. Поскольку процесс `A` продолжает попытки отправить сообщение, мы можем сказать, что оно не может быть безвозвратно потеряно в ходе транспортировки.

В терминологии распределенных систем эта абстракция называется *настойчивым каналом* (stubborn channel). Канал называется так по той причине, что отправитель продолжает посылать сообщение снова и снова до бесконечности, но, поскольку такого рода абстракция была бы крайне непрактичной, нам необходимо объединить повторы с подтверждениями.

Проблема с повторными передачами

После отправки сообщения и до получения подтверждения от удаленного процесса мы не знаем, было ли оно уже обработано, будет ли оно обработано в ближайшее время, было ли оно потеряно или произошел отказ удаленного процесса до его получения — возможно любое из этих состояний. Мы можем повторить операцию и снова отправить сообщение, но это может привести к дублированию сообщений. Обработка дублирующих сообщений безопасна, только если операция, которую мы собираемся выполнить, идемпотентна.

Идемпотентная (idempotent) операция — это операция, которая может выполняться несколько раз с одним и тем же результатом без дополнительных побочных эффектов. Например, операция выключения сервера может быть идемпотентной: первый вызов инициирует выключение, а все последующие вызовы не дают никаких дополнительных эффектов.

Если бы каждая операция была идемпотентной, мы могли бы меньше беспокоиться о семантике доставки, больше полагаться на повторные передачи для обеспечения отказоустойчивости и создавать системы с полностью реактивным поведением, когда действие запускается как ответ на некоторый сигнал, не вызывая непреднамеренных побочных эффектов. Однако операции не всегда являются идемпотентными, и, предполагая, что они являются таковыми, мы можем получить побочные эффекты по всему кластеру. Например, операция пополнения кредитной карты клиента не является идемпотентной, и совершенно очевидно, что ее не стоит повторять многократно.

Идемпотентность особенно важна при частичных отказах и распадах сети, поскольку мы не всегда можем узнать точный статус удаленной операции — выполнена ли она успешно, завершилась с ошибкой или будет выполнена в ближайшее время — и нам просто нужно подождать еще некоторое время. Поскольку в реальности невозможно

гарантировать, что каждая выполняемая операция будет идемпотентна, мы должны предоставить гарантии, *эквивалентные* идемпотентности, без изменения семантики базовой операции. Для этого мы можем использовать *дедупликацию* (deduplication) и не допускать обработку сообщения более одного раза.

Порядок сообщений

Ненадежные сети создают нам две проблемы: сообщения могут приходить в случайном порядке и из-за повторной передачи некоторые сообщения могут поступать более одного раза. Мы уже ввели порядковые номера и можем использовать эти идентификаторы сообщений на стороне получателя, чтобы обеспечить упорядочение согласно принципу FIFO (first-in, first-out, «первым пришел, первым вышел»). Поскольку каждое сообщение имеет порядковый номер, получатель может отслеживать следующие переменные:

- $n_{последовательные}$, содержит наибольший порядковый номер, до которого получатель уже видел все сообщения. Сообщения до этого номера можно расположить в исходном порядке.
- $n_{обработаные}$, содержит наибольший порядковый номер, до которого сообщения были расположены получателем в исходном порядке и *обработаны*. Этот номер можно использовать для дедупликации.

Если полученное сообщение имеет непоследовательный порядковый номер, получатель помещает его в буфер переупорядочения. Например, он получает сообщение с порядковым номером `5` после получения сообщения с номером `3` и мы знаем, что `4` все еще отсутствует, поэтому нужно отложить `5` до тех пор, пока не придет `4`, после чего мы сможем восстановить порядок сообщений. Поскольку в основу у нас положен канал с допустимыми потерями, мы предполагаем, что сообщения с номерами между $n_{последовательные}$ и $n_{макс\_увиденный}$ в конечном итоге будут доставлены.

Получатель может безопасно отбрасывать приходящие сообщения, порядковый номер которых меньше $n_{последовательные}$, поскольку они уже гарантированно доставлены.

Дедупликация заключается в проверке, было ли уже *обработано* (передано в стек получателем) сообщение с порядковым номером `n`, и в отбрасывании уже обработанных сообщений.

В терминах распределенных систем этот тип канала называется *совершенным каналом* (perfect link), который предоставляет следующие гарантии [CACHIN11]:

Надежная доставка

Каждое сообщение, отправленное один раз корректным процессом `A` корректному процессу `Б`, в конце концов будет доставлено.

Без дублирования

Ни одно сообщение не будет доставлено более одного раза.

Без создания

Подобно другим типам каналов, данный канал может доставлять только те сообщения, которые действительно были отправлены.

Такой канал может напомнить вам протокол TCP[1] (хотя надежная доставка в TCP гарантируется только в рамках одного сеанса). Конечно, эта модель является лишь упрощенным представлением, которое мы используем только для иллюстрации. Протокол TCP имеет намного более сложную модель для работы с подтверждениями, которая группирует подтверждения и снижает затраты на уровне протокола. Кроме того, в TCP предусмотрены выборочные подтверждения, управление потоком, контроль перегрузки, обнаружение ошибок и многие другие функции, которые выходят за рамки нашего обсуждения.

Строго однократная доставка

> В распределенных системах есть только две серьезные проблемы: 2. Строго однократная доставка. 1. Гарантированный порядок сообщений. 2. Строго однократная доставка.
>
> *Матиас Веррес*

Было много дискуссий о том, возможна ли *строго однократная доставка* (exactly-once delivery). Здесь важны и семантика, и точная формулировка. Поскольку возможен сбой канала связи, из-за которого сообщение не будет доставлено с первой попытки, большинство реальных систем используют *как минимум однократную доставку* (at-least-once delivery), которая гарантирует, что отправитель будет повторять попытку до тех пор, пока не получит подтверждение, иначе сообщение не будет считаться полученным. Еще одним видом семантики является *как максимум однократная доставка* (at-most-once delivery): здесь отправитель посылает сообщение и не ожидает подтверждения доставки.

Протокол TCP работает, разбивая сообщения на пакеты, передавая их один за другим и соединяя их на принимающей стороне. TCP может попытаться повторить передачу некоторых пакетов; при этом удачными могут оказаться несколько попыток передачи. Поскольку TCP помечает каждый пакет порядковым номером, то даже когда некоторые пакеты передаются более одного раза, он может дедуплицировать эти пакеты и гарантировать, что получатель увидит сообщение и обработает его только один раз. В TCP эта гарантия действительна только для одного сеанса: если сообщение подтверждено и обработано, но отправитель не получил подтверждение до прерывания соединения, приложение не узнает об этой доставке и, в зависимости от его логики, может попытаться отправить сообщение еще раз.

Это означает, что главное здесь — обеспечить строго однократную *обработку*, поскольку дублированные доставки (или передачи пакетов) не несут побочных эффектов и являются просто свидетельством производимых каналом усилий. Например, в случае, когда узел базы данных получил запись, но не сохранил ее на персистентном носителе, доставка произошла, но это бесполезно, поскольку мы не можем извлечь запись (как в том случае, если бы она была доставлена и обработана).

Чтобы гарантировать строгую однократность, узлы должны располагать некоторой общей информацией [HALPERN90]: каждый узел должен знать определенный факт,

[1] См. https://databass.dev/links/53.

а также то, что все другие узлы также знают об этом факте. Проще говоря, узлы должны согласовать состояние записи: оба узла должны согласиться с тем, что запись *была* или *не была*, сохранена на персистентном носителе. Как вы увидите позже в этой главе, это теоретически невозможно, но на практике эту идею все же можно использовать, снизив требования к координации.

Любые разногласия относительно того, возможна ли строго однократная доставка, возникают главным образом из-за рассмотрения этой проблемы на разных уровнях протокола и абстракции и использования разных определений термина «доставка». Невозможно создать надежный канал, совсем не используя повторную передачу сообщений, но мы можем создать иллюзию строго однократной доставки, с точки зрения отправителя, производя однократную *обработку* сообщений и игнорируя дубликаты.

Теперь, уже располагая средствами для надежного обмена данными, мы можем двинуться дальше и поискать способы обеспечения единообразия и согласованности между процессами в распределенной системе.

Задача двух генералов

Очень наглядной демонстрацией того, как обеспечивается согласованность в распределенной системе, является мысленный эксперимент, широко известный как «задача двух генералов».

Этот мысленный эксперимент показывает, что при наличии сбоев канала невозможно обеспечить согласованность между двумя сторонами, если связь является *асинхронной*. Хотя TCP обладает свойствами совершенного канала, важно помнить, что совершенные каналы, несмотря на название, не гарантируют идеальную доставку. Они также не могут гарантировать, что участники будут все время находиться в активном состоянии, и касаются лишь вопроса транспортировки данных.

Представьте две армии каждая во главе с генералом, которые готовятся к атаке на укрепленный город. Армии расположены с разных сторон города и могут добиться успеха в осаде, только если нападут одновременно.

Генералы могут общаться, отправляя посланников, и уже разработали план атаки. Им осталось договориться лишь о том, следует ли выполнять этот план. Есть еще два варианта этой задачи: в первом один из генералов имеет более высокий ранг, но должен проследить за тем, чтобы атака была скоординированной, а во втором — генералы должны договориться о точном времени. Эти детали не меняют определение задачи: генералы должны достичь соглашения.

Генералам армий необходимо договориться только о том, что они оба атакуют. В противном случае атака не будет успешной. Генерал `A` отправляет послание `MSG(N)`, сообщая о намерении произвести атаку в указанное время *при условии*, что другая сторона согласится сделать то же самое.

После того как генерал А отправляет посланника, он не знает, прибыл тот или нет: посланника могут перехватить и тогда он не сможет доставить сообщение. Когда генерал Б получает сообщение, он должен отправить подтверждение ACK(MSG(N)). На рис. 8.4 показано, что сообщение отправляется в одну сторону и подтверждается другой стороной.

Посланник, несущий это подтверждение, также может быть перехвачен или просто не справиться с доставкой. У генерала Б нет никакого способа узнать, преуспел ли посланник в доставке сообщения.

Чтобы быть уверенным в этом, Б должен дождаться подтверждения второго порядка ACK(ACK(MSG(N))), сообщающее о том, что А получил подтверждение.

Независимо от того, сколько дополнительных подтверждений генералы отправят друг другу, они всегда будут на расстоянии в одно подтверждение ACK от уверенности в том, что могут безопасно произвести атаку. Генералы обречены задаваться вопросом о том, достигло ли адресата сообщение с этим последним подтверждением.

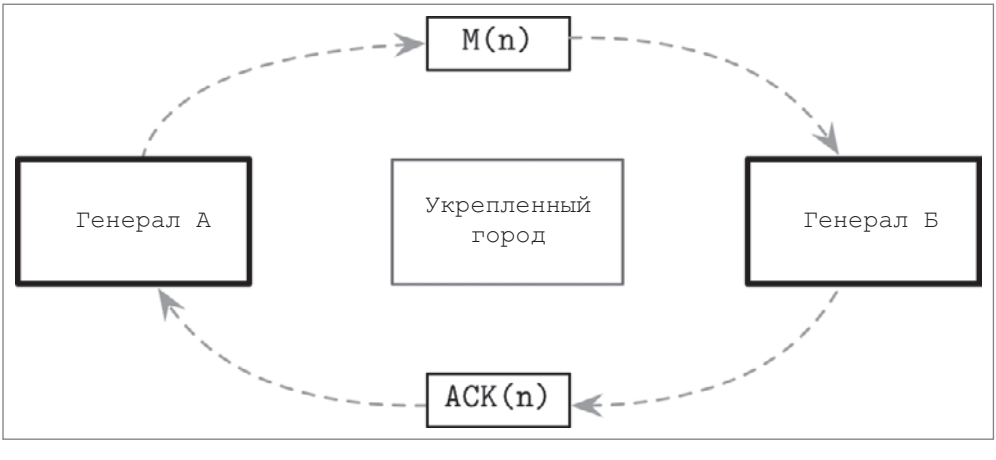

Рис. 8.4. Задача двух генералов

Обратите внимание, что мы не сделали никаких предположений в отношении времени: связь между генералами полностью асинхронна. Не установлено никакого лимита на количество времени, затрачиваемое генералами на выдачу ответа.

Невозможность Фишера–Линча–Патерсона

В статье Фишера, Линча и Патерсона описывается проблема, известная как *проблема невозможности ФЛП* (FLP Impossibility) (название получено из первых букв фамилий авторов) [FISCHER85] или *теорема Фишера–Линча–Патерсона*, в которой

обсуждается вариант консенсуса, когда процессы начинают работу с начальным значением и пытаются согласовать новое значение. После выполнения алгоритма это новое значение должно быть одинаковым для всех исправных процессов.

Обеспечение согласованности в отношении конкретного значения не представляет проблем, когда сеть абсолютно надежна, однако реальные системы подвержены множеству разного рода сбоев, таких как потеря сообщений, дублирование, распад сети и медленная работа или отказ процессов.

Протокол консенсуса описывает систему, которая при наличии нескольких процессов, начинающих работу с ее *начального состояния*, переводит все процессы в *состояние принятия решения*. Чтобы протокол консенсуса был корректным, он должен обеспечивать три параметра:

Согласованность (agreement)

Принимаемое протоколом решение должно быть единодушным: каждый процесс выбирает некоторое значение, которое должно быть одинаковым для всех процессов. В противном случае мы не достигаем консенсуса.

Действительность (validity)

Согласованное значение должно быть *предложено* одним из участников, т. е. это значение не должно быть просто «придумано» системой. Это также подразумевает нетривиальность значения: процессы не всегда выбирают некоторое предопределенное дефолтное значение.

Окончательность (termination)

Согласованность принимает окончательный характер после того, как уже не остается процессов, не достигших состояния принятия решения.

В источнике [FISCHER85] предполагается полностью асинхронная обработка, когда процессы не используют общее время. Алгоритмы в таких системах не могут использовать время ожидания, и у процесса нет никакой возможности узнать, произошел ли отказ другого процесса или он просто работает слишком медленно. В данной статье показано, что с учетом этих допущений не существует протокола, который мог бы гарантировать консенсус в ограниченное время. Для любого полностью асинхронного алгоритма достижения консенсуса будет недопустимым возникновение даже одного отказа удаленного процесса без уведомления о нем.

Если мы не задаем верхний лимит времени для процесса на выполнение шагов алгоритма, то нельзя надежно обнаружить отказы процесса и нет никакого детерминированного алгоритма для достижения консенсуса.

Однако теорема ФЛП совсем не означает, что поскольку достижение консенсуса невозможно, все наши усилия обречены на неудачу. Она лишь означает, что не всегда получается достичь консенсуса в асинхронной системе за ограниченное время. Реальные системы демонстрируют как минимум некоторую степень синхронности, и нам лишь нужно использовать более продуманную модель для решения данной проблемы.

Синхронность системы

Из теоремы ФЛП можно сделать вывод о том, что одной из критических характеристик распределенной системы является предположение в отношении времени. В *асинхронной системе* мы не знаем относительных скоростей процессов и не можем гарантировать доставку сообщений в ограниченное время или в определенном порядке. Процесс может выдать ответ через неопределенно долгий промежуток времени, и часто невозможно обеспечить надежное выявление отказов процесса.

Один из главных аргументов против асинхронных систем заключается в том, что эти допущения не являются реалистичными: процессы не могут иметь *произвольно* разные скорости обработки, а каналы не могут доставлять сообщения *неопределенно долго*. Используя время, мы можем одновременно и упростить логику рассуждений, и предоставить гарантии в отношении верхнего лимита времени.

В асинхронной модели не всегда возможно решить проблему консенсуса [FISCHER85]. Более того, часто невозможно разработать и эффективный синхронный алгоритм, поэтому большинство практических решений некоторых задач является зависимым от времени [ARJOMANDI83].

Мы можем сделать эти предположения менее строгими, считая систему *синхронной*. Для этого мы введем понятие времени. Намного проще рассматривать систему с помощью синхронной модели. Предполагается, что процессы работают с сопоставимыми скоростями, задержки передачи ограничены, а доставка сообщений не может занимать произвольно много времени.

В модель синхронной системы также можно добавить локальные для процесса синхронизированные часы: при этом существует некоторая верхняя граница в разнице во времени между двумя локальными для процессов источниками времени [CACHIN11].

Проектирование систем по синхронной модели позволяет нам использовать время ожидания. Мы можем построить и более сложные абстракции, такие как выбор лидера, консенсус, обнаружение отказов и т. д. Это делает более устойчивыми сценарии по наилучшему варианту, но приводит к сбою, если предположения о времени нарушаются. Например, в алгоритме консенсуса Raft (см. раздел «Raft» на с. 320) мы можем получить несколько процессов-лидеров, что разрешается путем принуждения более медлительного процесса к принятию другого процесса в качестве лидера; алгоритмы обнаружения отказов (см. главу 9) могут ошибочно идентифицировать активный процесс как отказавший или наоборот. При разработке систем мы должны обязательно учитывать эти варианты развития событий.

Свойства асинхронных и синхронных моделей можно объединить, рассматривая систему как *частично синхронную*. Частично синхронная система обладает некоторыми свойствами синхронной системы, но при этом ограничения на время доставки сообщений, уход показаний часов и относительные скорости обработки могут быть приблизительными и действовать лишь *в большинстве случаев* [DWORK88].

Синхронность является важным свойством распределенной системы: она влияет на производительность, масштабируемость и общую разрешимость и подразумевает целый ряд факторов, необходимых для правильного функционирования наших систем. Некоторые из алгоритмов, которые мы обсуждаем в этой книге, подразумевают использование предположений, свойственных синхронным системам.

Модели отказов

Мы все время говорим о *сбоях* и *отказах* (failures), но до сих пор это были довольно широкие и общие понятия, за которыми может скрываться самый разный смысл. Подобно тому как мы можем делать различные предположения в отношении времени, мы можем предполагать и наличие различных типов отказов. *Модель отказов* описывает, как именно могут происходить отказы процессов в распределенной системе, и алгоритмы разрабатываются с использованием этих предположений. Например, мы можем предположить, что процесс может завершиться отказом и не восстановиться, или что он должен восстановиться через некоторое время, или что он может дать сбой, выйдя из-под контроля и предоставляя некорректные значения.

В распределенных системах процессы зависят друг от друга в ходе выполнения алгоритма, поэтому отказы могут привести к некорректному выполнению в рамках всей системы.

Мы обсудим несколько моделей отказов, используемых в распределенных системах, включая *аварийное завершение, пропуски* и *произвольные отказы* (crash, omission, arbitrary faults). Этот список не является исчерпывающим, но охватывает большинство случаев, применимых и важных для реальных систем.

Аварийное завершение

Обычно мы ожидаем, что процесс правильно выполнит все шаги алгоритма. Самым простым вариантом аварийного завершения процесса является *остановка* выполнения всех дальнейших шагов, требуемых алгоритмом, без отправки каких-либо сообщений другим процессам. То есть происходит *аварийное завершение* процесса. В большинстве случаев при этом используется абстракция процесса *«отказ-остановка»* (crash-stop), согласно которой после аварийного завершения процесс остается в этом состоянии.

Эта модель не подразумевает невозможность восстановления процесса и не препятствует восстановлению и не пытается его предотвратить. Она лишь означает, что алгоритм не полагается на восстановление при определении корректности или актуальности. Ничто не мешает процессам восстанавливаться, синхронизироваться с состоянием системы и участвовать в *следующем* экземпляре алгоритма.

Отказавшие процессы не могут продолжать участвовать в текущем раунде согласований, в ходе которого они дали сбой. Присвоение восстанавливающемуся процессу нового идентификатора не делает эту модель эквивалентной модели «отказ-восстановление» (обсуждается далее), так как большинство алгоритмов используют

предварительно определенные списки процессов и четко определяют семантику отказов в плане допустимого количества отказов [CACHIN11].

Абстракция процесса «отказ-восстановление» подразумевает, что процесс прекращает выполнение требуемых алгоритмом шагов, но восстанавливается на более позднем этапе и пытается выполнить дальнейшие шаги. Возможность восстановления требует введения в систему понятий устойчивого состояния и протокола восстановления [SKEEN83]. Алгоритмы, которые допускают восстановление после сбоя, должны учитывать все возможные состояния восстановления, поскольку восстанавливающийся процесс может попытаться продолжить выполнение с последнего известного ему шага.

Алгоритмы, предусматривающие использование восстановления, должны одновременно учитывать и состояние, и идентификатор. Модель «отказ-восстановление» при этом также можно рассматривать как частный случай пропуска, поскольку с точки зрения другого процесса нет никакой разницы между процессом, который был недоступен, и процессом, который аварийно завершился и восстановился.

Пропуск

Еще одной моделью отказа является *пропуск*. Эта модель предполагает, что процесс пропускает некоторые этапы алгоритма, или не может их выполнить, или это выполнение невидимо для других участников, или он не может отправлять или получать сообщения от других участников. Модель пропуска включает в себя распады сети между процессами, вызванные неисправными каналами сети, отказами коммутаторов или перегрузкой сети. Распады сети можно рассматривать как пропуски сообщений между отдельными процессами или группами процессов. Отказ процесса можно сымитировать путем полного пропуска любых сообщений, идущих к процессу или от него.

Когда процесс работает медленнее, чем другие участники, и отправляет ответы намного позже, чем ожидалось, для остальной части системы это может выглядеть как забывание. Вместо полной остановки медленный узел пытается отправить свои результаты без синхронизации с другими узлами.

Пропуски возникают, когда алгоритм, который должен был выполнить определенные шаги, либо пропускает их, либо результаты этого выполнения не видны. Например, это может произойти, если сообщение теряется на пути к получателю и отправителю не удается отправить его снова, но он продолжает работать, как если бы оно было успешно доставлено, несмотря на то что оно было безвозвратно потеряно. Пропуски также могут быть вызваны перемежающимися зависаниями, перегруженными сетями, полными очередями и т. д.

Произвольные ошибки

Труднее всего иметь дело с таким классом отказов, как *произвольные*, или *византийские*, *ошибки* (Byzantine faults): при этом процесс продолжает выполнять шаги

алгоритма, но делает это противоречащим алгоритму образом (так, например, в алгоритме консенсуса процесс может выбрать значение, которое не было предложено одним из других участников).

Такие отказы могут возникать из-за ошибок в программном обеспечении или выполнения процессами разных версий алгоритма; эти проблемы легко поддаются обнаружению и интерпретации. Однако ситуация становится гораздо сложнее, когда при отсутствии контроля над всеми процессами один из процессов намеренно вводит в заблуждение другие процессы.

Возможно, вы слышали о византийской отказоустойчивости из авиационно-космической промышленности: системы самолетов и космических аппаратов не принимают ответы от подкомпонентов за чистую монету и проводят перекрестную проверку их результатов. Еще одной обширной областью применения являются криптовалюты [GILAD17], где нет центральных органов управления, узлы контролируются разными сторонами, и злоумышленники материально мотивированы к тому, чтобы сфальсифицировать значения и попытаться «обхитрить» систему, предоставив ей неправильные ответы.

Обработка отказов

Мы можем *замаскировать* отказ, сформировав группы процессов и введя в алгоритм избыточность: даже если произойдет отказ одного из процессов, пользователь этого не заметит [CHRISTIAN91].

Отказы могут привести к некоторому снижению производительности: нормальное выполнение подразумевает реагирование процессов, а в случае отказа система вынуждена использовать более медленный путь выполнения для обработки и исправления ошибок. Многие отказы можно предотвратить на уровне программного обеспечения с помощью анализа кода, всестороннего тестирования, обеспечения гарантированной доставки сообщений путем введения времени ожидания и повторных попыток и соблюдения последовательности выполнения шагов на локальном уровне.

Большинство алгоритмов, которые мы здесь рассмотрим, подразумевает использование модели отказов и обходят отказы путем введения избыточности. Эти допущения помогают создавать алгоритмы, которые лучше работают и легче поддаются пониманию и реализации.

Итоги

В этой главе мы обсудили некоторые термины и ряд основных концепций распределенных систем. Мы обсудили неотъемлемые проблемы таких систем, связанные с ненадежностью их компонентов: каналы могут не справляться с доставкой сообщений, процессы могут давать сбой, а в сетях может возникать такая проблема, как распад сети.

Этой терминологии должно быть достаточно для продолжения обсуждения. В остальной части книги будет рассказано о *решениях*, широко используемых в распределенных системах: мы еще раз вспомним о том, что может пойти не так, и посмотрим, какие у нас есть варианты.

Дополнительная литература

Если вы хотите узнать больше о концепциях, упомянутых в этой главе, вы можете обратиться к следующим источникам:

Абстракции распределенных систем, модели отказов и предположения в отношении времени

Lynch, Nancy A. 1996. Distributed Algorithms. San Francisco: Morgan Kaufmann.

Tanenbaum, Andrew S. and Maarten van Steen. 2006. Distributed Systems: Principles and Paradigms (2nd Ed). Boston: Pearson[1].

Cachin, Christian, Rachid Guerraoui, and Lus Rodrigues. 2011. Introduction to Reliable and Secure Distributed Programming (2nd Ed.). New York: Springer[2].

[1] *Таненбаум Э., Стеен М. ван.* Распределенные системы. Принципы и парадигмы. СПб.: Питер, 2013. — *Примеч. ред.*

[2] *Качин К., Гуерру Р., Родригес Л.* Введение в надежное и безопасное распределенное программирование. М.: ДМК Пресс, 2016. — *Примеч. ред.*

ГЛАВА 9

Обнаружение отказов

Слышен ли звук падающего дерева в лесу,
если рядом никого нет?

Неизвестный автор

Для того чтобы система надлежащим образом реагировала на отказы, их следует своевременно обнаруживать. Попытки установить связь с неисправным узлом могут продолжаться, даже если он продолжает не отвечать, увеличивая задержки и снижая общую доступность системы.

Обнаружить отказ в асинхронных распределенных системах (т. е. без каких-либо предположений о времени) чрезвычайно сложно, так как невозможно определить, произошло ли аварийное завершение процесса или он работает медленно, и для ответа требуется неопределенно много времени. Мы обсуждали эту проблему в разделе «Невозможность Фишера–Линча–Патерсона» на с. 207.

Такие определения, как «мертвый» (dead), «отказавший» (faulty) и «аварийно завершившийся» (crashed), обычно используются для описания процессов, которые полностью прекратили выполнение своих шагов. Такие определения, как «неотвечающий» (unresponsive), «неисправный» (faulty) и «медленный» (slow), используются для описания подозрительных процессов, которые на самом деле могут быть «мертвыми».

Отказы могут возникать на уровне *канала* (когда сообщения между процессами теряются или доставляются медленно) или на уровне *процесса* (когда процесс аварийно завершается или работает медленно); при этом медленное выполнение иногда невозможно отличить от отказа. Это означает, что всегда приходится находить компромисс между ошибочным отнесением активных процессов к числу отказавших (что ведет к получению *ложноположительных* результатов) и задержкой отнесения неотвечающих процессов к числу отказавших с предоставлением им возможности рано или поздно ответить (что ведет к получению *ложноотрицательных* результатов).

Детектор отказов (failure detector) — это локальная подсистема, отвечающая за выявление отказавших или недоступных процессов с целью исключить их из алгоритма и гарантировать его живучесть с сохранением безопасности.

Живучесть и безопасность — это свойства, которые отражают способность алгоритма решать конкретную проблему и корректность выдаваемых им результатов. Если выражаться более формально, свойство *живучести* (likeness) гарантирует, что *произойдет* конкретное предполагаемое событие. Например, в случае отказа одного из процессов детектор отказов должен обнаружить этот отказ. Свойство *безопасности*

(safety) гарантирует, что *не произойдут* непредвиденные события. Например, если детектор отказов пометит процесс как «мертвый», этот процесс должен на самом деле быть «мертвым» [LAMPORT77] [RAYNAL99] [FREILING11].

С практической точки зрения исключение отказавших процессов помогает избежать выполнения лишней работы и предотвращает распространение отказов и каскадные отказы, одновременно снижая доступность за счет исключения подозрительных активных процессов.

Алгоритмы обнаружения отказов должны обладать несколькими неотъемлемыми свойствами. Прежде всего каждый свободный от неисправностей участник должен в конце концов заметить отказ процесса, а алгоритм должен быть в состоянии продолжить работу и в конечном итоге достичь своего целевого результата. Это свойство называется *завершаемостью* (completeness).

Качество алгоритма можно оценить по его *эффективности*, т. е. по тому, насколько быстро детектор отказов способен идентифицировать отказы процессов. Алгоритм также можно оценить по его *точности*, т. е. по тому, насколько точно выявляется отказ процесса. Другими словами, алгоритм не является точным, если он ошибочно относит активный процесс к числу отказавших или не способен обнаружить существующие отказы.

Мы можем рассматривать отношение между эффективностью и точностью как настраиваемый параметр: более эффективный алгоритм может быть менее точным, а более точный алгоритм обычно менее эффективен. Вероятно, невозможно создать детектор отказов, который будет и точным, и эффективным. В то же время детекторам отказов разрешается выдавать ложноположительные результаты (т. е. ложно идентифицировать активные процессы как отказавшие и наоборот) [CHANDRA96].

Детекторы отказов являются важным элементом и неотъемлемой частью многих алгоритмов достижения консенсуса и алгоритмов атомарной рассылки, которые мы обсудим позже в этой книге.

Многие распределенные системы реализуют детекторы отказов с применением *контрольных пакетов* (heartbeat). Этот подход довольно популярен из-за своей простоты и сильной завершаемости. Алгоритмы, которые мы здесь обсуждаем, предполагают отсутствие византийских ошибок: процессы не пытаются преднамеренно выдавать ложную информацию о своем состоянии или состояниях своих соседей.

Контрольные пакеты и эхо-запросы

Мы можем запрашивать состояние удаленных процессов, используя один из двух периодических процессов:

- Мы можем использовать эхо-запросы (ping), отправляя сообщения удаленным процессам и проверяя их активность путем ожидания ответа в течение определенного времени.
- Мы можем использовать *контрольные пакеты*: при этом процесс активно уведомляет одноранговые процессы о своей активности путем отправки им сообщений.

В следующем примере мы будем использовать эхо-запросы, но эту же задачу можно решить с аналогичными результатами и с помощью контрольных пакетов.

Каждый процесс поддерживает список других процессов (активных, отказавших и подозрительных) и обновляет его данными о времени последнего ответа для каждого процесса. Если процесс не отвечает на сообщение эхо-запроса в течение длительного времени, он помечается как *подозрительный* (suspected).

На рис. 9.1 показано нормальное функционирование системы: процесс P_1 запрашивает состояние соседнего узла P_2, который отвечает подтверждением.

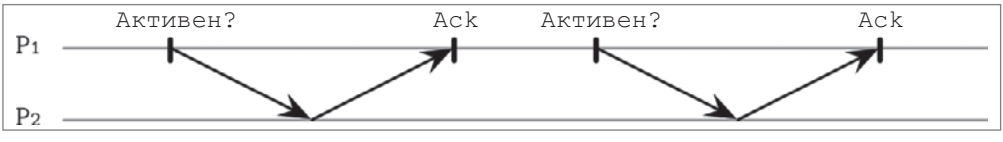

Рис. 9.1. Обнаружение отказов с помощью эхо-запросов (ping): нормальное функционирование, отсутствие задержек сообщений

На рис. 9.2 показан случай, когда подтверждения задерживаются, что может привести к отнесению активного процесса к числу неработающих.

Рис. 9.2. Обнаружение отказов с помощью эхо-запросов (ping): ответы задерживаются, приходя после отправки следующего сообщения

Многие алгоритмы обнаружения отказов используют контрольные пакеты и время ожидания. Например, в популярном фреймворке для создания распределенных систем Akka реализован детектор отказов на основе лимита времени, который использует контрольные пакеты и уведомляет о сбое процесса, если тот не «отметился» в течение фиксированного интервала времени.

У этого подхода есть несколько потенциальных недостатков: его точность зависит от правильного выбора частоты отправки эхо-запросов и величины времени ожидания, и, кроме того, он не учитывает видимость процесса с точки зрения других процессов (см. подраздел «Сторонние контрольные пакеты» на с. 217).

Детектор отказов без времени ожидания

Некоторые алгоритмы не используют время ожидания для обнаружения отказов. Например, не использующий время ожидания детектор отказов на основе контрольных

пакетов [AGUILERA97] представляет собой алгоритм, который просто подсчитывает контрольные пакеты и позволяет приложению выявлять отказы процессов на основе содержимого векторов счетчиков контрольных пакетов. Поскольку этот алгоритм не использует время ожидания, он работает исходя из допущений, свойственных *асинхронной* системе.

Этот алгоритм предполагает, что любые два корректных процесса должны быть связаны друг с другом посредством линии связи с «допустимыми потерями», которая содержит только каналы с допустимыми потерями (т. е. если сообщение передается по этому каналу бесконечное число раз, оно также принимается бесконечное число раз), и что каждый процесс осведомлен о существовании всех других процессов в сети.

Каждый процесс поддерживает список соседей и связанные с ними счетчики. Процессы начинают выполнение с отправки сообщений контрольных пакетов соседям. Каждое сообщение содержит информацию о том, по какому пути уже прошел контрольный пакет. Исходное сообщение содержит данные о первом отправителе в качестве пути и уникальный идентификатор, который можно использовать для исключения многократной отправки одного и того же сообщения.

Когда процесс получает новое сообщение контрольного пакета, он увеличивает счетчики для всех участников, присутствующих в пути, и отправляет контрольный пакет тем участникам, которые не указаны в пути, добавив в путь информацию о себе. Процессы прекращают распространение сообщения после того, как его получают все известные процессы (о чем говорит наличие идентификатора процесса в информации о пути).

Поскольку сообщения распространяются разными процессами и пути контрольных пакетов содержат агрегированную информацию, полученную от соседей, мы можем (корректно) пометить недоступный процесс как активный, даже если прямая связь между двумя процессами не работает.

Счетчики контрольных пакетов представляют собой общее и нормализованное представление системы. Это представление показывает, как контрольные пакеты распространяются относительно друг друга, что позволяет нам сравнивать процессы. Однако одним из недостатков этого подхода является то, что интерпретация счетчиков контрольных пакетов может оказаться довольно сложной: необходимо выбрать порог, способный обеспечить надежные результаты. В противном случае алгоритм будет ошибочно помечать активные процессы как подозрительные.

Сторонние контрольные пакеты

В протоколе SWIM [GUPTA01] используется альтернативный подход, который сводится к тому, чтобы использовать *сторонние контрольные пакеты* для повышения надежности за счет получения информации о жизнеспособности процесса с точки зрения его соседей. Этот подход не требует, чтобы процессы были осведомлены обо всех других процессах в сети, — достаточно иметь информацию лишь о некотором подмножестве подключенных одноранговых узлов.

Как показано на рис. 9.3, процесс P_1 отправляет сообщение эхо-запроса процессу P_2. Процесс P_2 не отвечает на сообщение, поэтому процесс P_1 продолжает, выбрав нескольких случайных участников (P_3 и P_4). Эти случайные участники пытаются отправить сообщения контрольных пакетов процессу P_2 и, если он отвечает, направляют подтверждения обратно процессу P_1.

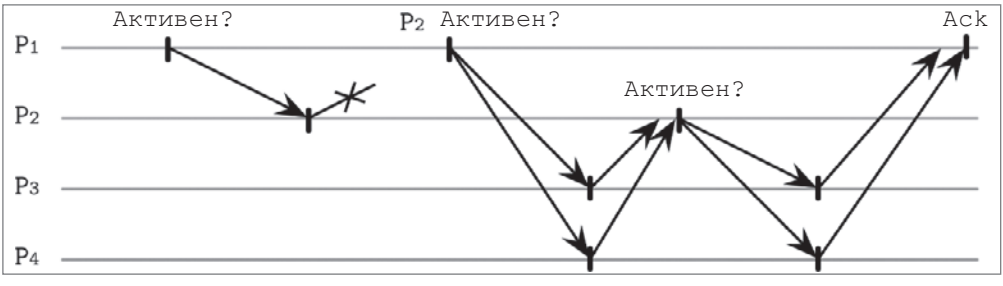

Рис. 9.3. Сторонние контрольные пакеты

Этот подход позволяет учитывать как прямую, так и опосредованную доступность. Например, если у нас есть процессы P_1, P_2 и P_3, мы можем проверить состояние процесса P_3 и со стороны процесса P_1, и со стороны процесса P_2.

Алгоритм сторонних контрольных пакетов позволяет надежно обнаруживать отказы, распределяя ответственность за принятие решений по группе участников. Этот подход не требует широковещательной рассылки сообщений большой группе процессов. Поскольку сторонние контрольные пакеты могут запускаться параллельно, с помощью этого подхода можно быстро собирать больше информации о подозрительных процессах и принимать более взвешенные решения.

Детектор отказа с накопленным уровнем подозрительности

Вместо того чтобы рассматривать отказ узла как бинарную задачу, где процесс может быть только в двух состояниях: работоспособном или неработоспособном, детектор отказа с накопленным фи (φ) (phi accrual failure detector) [HAYASHIBARA04] использует непрерывную шкалу, отражающую вероятность аварийного завершения отслеживаемого процесса. Такой детектор использует скользящее окно, собирая данные о времени поступления самых последних контрольных пакетов от других процессов. Эта информация используется для вычисления приблизительного времени поступления *следующего* контрольного пакета, дальнейшего сравнения этого значения с фактическим временем поступления и вычисления *уровня подозрительности* φ — степени уверенности детектора отказа в наличии отказа с учетом текущего состояния сети.

Данный алгоритм работает путем сбора и выборки данных о времени поступления с созданием представления, позволяющего надежно оценить работоспособность узла. На основе этих выборочных данных вычисляется значение φ, и если оно превышает некоторый порог, узел помечается как неработоспособный. Такой детектор отказов динамически адаптируется к изменяющимся условиям сети путем регулирования шкалы, используемой для выявления подозрительных узлов.

С точки зрения архитектуры детектор отказа с накопленным уровнем подозрительности можно рассматривать как комбинацию трех подсистем:

Мониторинг

Сбор информации о живучести с помощью эхо-запросов (ping), контрольных пакетов или выборочных данных о запросах и ответах.

Интерпретация

Принятие решения о том, должен ли процесс быть помечен как подозрительный.

Действие

Ответный вызов, выполняемый всякий раз, когда процесс помечается как подозрительный.

Процесс мониторинга собирает и сохраняет выборки данных (которые, как предполагается, следуют нормальному распределению) в фиксированного размера окне времен поступления контрольных пакетов. Новые поступления добавляются в окно, а самые старые элементы данных о контрольных пакетах отбрасываются.

Параметры распределения оцениваются на основе окна выборки путем определения среднего значения и дисперсии выборок. Эта информация используется для вычисления вероятности поступления сообщения в течение t единиц времени после поступления предыдущего. На основе этой информации мы вычисляем показатель φ, отражающий вероятность принятия верного решения об активности процесса, или, иначе говоря, вероятность того, что мы допустим ошибку и получим контрольный пакет, не согласующийся с проведенными расчетами.

Этот подход разработан исследователями из Японского передового института науки и техники (JAIST) и в настоящее время используется во многих распределенных системах, например в системах Cassandra и Акка (как и вышеупомянутый детектор отказов на основе лимита времени).

Сплетни и обнаружение отказов

Другим подходом, который при принятии решения не полагается на представление с точки зрения одного узла, является служба обнаружения отказов со сплетнями [VANRENESSE98], которая использует механизм *«сплетен»* (см. раздел «Распространение сплетен» на с. 268) для сбора и распространения информации о состоянии соседних процессов.

Каждый участник поддерживает список других участников с соответствующими *счетчиками контрольных пакетов* и временными метками, отражающими время последнего увеличения счетчика контрольных пакетов. Периодически каждый участник увеличивает свой счетчик контрольных пакетов и отправляет свой список случайному соседу. После получения сообщения соседний узел объединяет принятый список со своим собственным, обновляя счетчики для других соседей.

Кроме того, узлы периодически проверяют список состояний и счетчики контрольных пакетов. Если какой-либо узел не обновлял свой счетчик достаточно долго, он считается отказавшим. Необходимо тщательно подбирать величину этого периода ожидания, чтобы минимизировать вероятность получения ложноположительных результатов. Частота обмена сообщениями между участниками (т. е. пропускная способность в наихудшем случае) не должна быть больше некоторого предела и может возрастать не быстрее линейной зависимости от количества процессов в системе.

На рис. 9.4 показано, как три процесса могут обмениваться данными о своих счетчиках контрольных пакетов:

а) Все три процесса могут обмениваться сообщениями и обновлять свои временные метки.

б) Процесс P_3 не может связаться с процессом P_1, но его временная метка t_6 по-прежнему может распространяться через процесс P_2.

в) Произошло аварийное завершение процесса P_3. Поскольку он больше не отправляет обновления, он распознается другими процессами как отказавший.

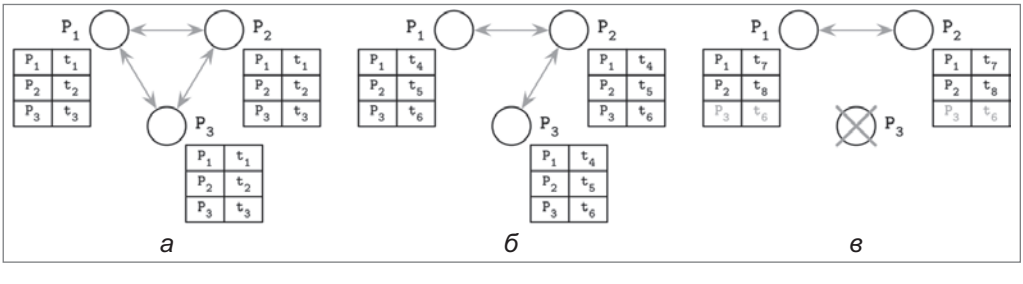

Рис. 9.4. Обнаружение отказов путем репликации таблицы контрольных пакетов

Таким образом, мы можем выявлять отказавшие узлы, а также узлы, недоступные ни одному другому участнику кластера. Принимаемое решение является надежным, поскольку представление кластера представляет собой агрегат данных, получаемых от нескольких узлов. При сбое канала между двумя хостами контрольные пакеты все равно могут распространяться через другие процессы. Использование механизма сплетен для распространения состояний системы увеличивает количество сообщений в системе, но позволяет распространять информацию более надежно.

Обратный взгляд на проблему обнаружения отказов

Поскольку распространение информации об отказах не всегда возможно, а ее распространение путем уведомления каждого участника может оказаться затратным, один из подходов, называемый FUSE (служба уведомления о сбоях, failure notification service) [DUNAGAN04], ставит целью обеспечение надежного и незатратного распространения информации об отказах, которое будет работать даже в случае распада сети.

Для обнаружения отказов процессов этот подход объединяет все активные процессы в группы. Если одна из групп становится недоступной, все участники обнаруживают отказ. Другими словами, каждый раз, когда обнаруживается отказ одного процесса, он преобразуется и распространяется как *групповой отказ*. Это позволяет обнаруживать отказы при наличии любой схемы потери соединения, распадов сети и отказов узлов.

Процессы в группе периодически отправляют сообщения эхо-запросов (ping) другим участникам с целью выяснить, являются ли они по-прежнему активными. Если один из участников не может ответить на это сообщение из-за отказа, распада сети или сбоя соединения, то участник, инициировавший эхо-запрос, в свою очередь, сам перестает отвечать на сообщения эхо-запросов.

На рис. 9.5 показано, как могут обмениваться данными четыре процесса:

а) Исходное состояние: все процессы активны и могут обмениваться сообщениями.

б) Процесс P_2 аварийно завершается и перестает отвечать на сообщения эхо-запросов.

в) Процесс P_4 обнаруживает отказ процесса P_2 и перестает отвечать на сообщения эхо-запросов.

г) В конечном итоге процессы P_1 и P_3 замечают, что процессы P_1 и P_2 не отвечают, и отказ процесса распространяется на всю группу.

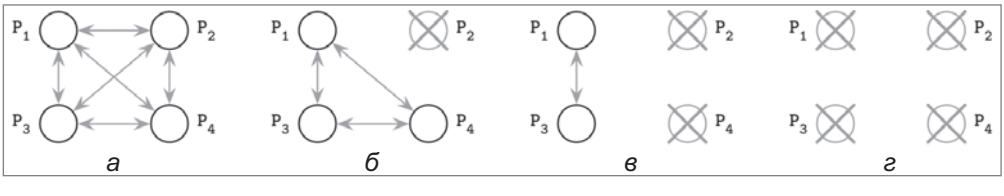

Рис. 9.5. Обнаружение отказа службой FUSE

Все отказы распространяются через систему от источника отказа ко всем остальным участникам. Участники постепенно прекращают отвечать на сообщения эхо-запросов, превращая отказ отдельного узла в групповой отказ.

Здесь мы используем отсутствие связи как средство распространения. Преимущество этого подхода заключается в том, что каждый участник гарантированно узнает о групповом отказе и адекватно отреагирует на него. Одним из недостатков является то, что отказ канала, отделяющий один процесс от других процессов, также может

быть преобразован в групповой отказ, но, в зависимости от сценария использования, это может рассматриваться и как преимущество. Приложения могут принять в расчет такой сценарий, используя собственные определения распространяющихся отказов.

Итоги

Детекторы отказов являются неотъемлемой частью любой распределенной системы. Как видно из теоремы Фишера–Линча–Патерсона, ни один протокол не в состоянии гарантировать консенсус в асинхронной системе. Детекторы отказов помогают дополнить модель, позволяя нам решить проблему консенсуса за счет компромисса между точностью и завершаемостью. Одно из важных открытий в этой области, доказывающее полезность детекторов отказов, было описано в источнике [CHANDRA96], в котором показано, что достижение консенсуса возможно даже с детектором отказов, который совершает бесконечное количество ошибок.

Мы рассмотрели несколько алгоритмов обнаружения отказа, каждый из которых использует свой подход: некоторые фокусируются на обнаружении отказов путем прямого обмена сообщениями, некоторые используют широковещательную рассылку или механизм сплетен для распространения информации, а некоторые, наоборот, используют бездействие (другими словами, отсутствие связи) как средство распространения. Теперь мы знаем, что можем использовать контрольные пакеты или эхо-запросы (ping), жесткие временные рамки или непрерывную шкалу. Каждый из этих подходов имеет свои преимущества: простоту, безошибочность, или точность.

Дополнительная литература

Если вы хотите узнать больше о концепциях, упомянутых в этой главе, вы можете обратиться к следующим источникам:

Обнаружение отказов и алгоритмы

Chandra, Tushar Deepak and Sam Toueg. 1996. “Unreliable failure detectors for reliable distributed systems.” Journal of the ACM 43, no. 2 (March): 225–267. https://doi.org/10.1145/226643.226647.

Freiling, Felix C., Rachid Guerraoui, and Petr Kuznetsov. 2011. “The failure detector abstraction.” ACM Computing Surveys 43, no. 2 (January): Article 9. https://doi.org/10.1145/1883612.1883616.

Phan-Ba, Michael. 2015. “A literature review of failure detection within the context of solving the problem of distributed consensus.” https://www.cs.ubc.ca/~best chai/theses/michael-phan-ba-msc-essay-2015.pdf.

ГЛАВА 10

Выбор лидера

Синхронизация часто требует довольно больших затрат: если на каждом шаге алгоритма требуется контактировать со всеми другими участниками, это может привести к большим затратам на обмен данными. Это особенно актуально в больших и территориально распределенных сетях. Чтобы уменьшить затраты на синхронизацию и количество запросов и подтверждений, необходимых для принятия решения, некоторые алгоритмы предполагают наличие *лидера* (leader) (или *координатора* (coordinator)) — процесса, отвечающего за выполнение или координацию шагов распределенного алгоритма.

Обычно в распределенных системах процессы единообразны, и взять на себя роль лидера может любой из них. Процессы выступают в качестве лидера в течение длительных периодов, однако это не является их постоянной ролью. Обычно процесс остается лидером до своего аварийного завершения. После аварийного завершения любой другой процесс может начать новый тур выбора, взять на себя лидерство, если он будет избран, и продолжить работу аварийного лидера.

Живучесть алгоритма выбора гарантирует, что лидер будет присутствовать *практически всегда* и выбор в конечном итоге будет сделан (т. е. система не должна находиться в состоянии выбора бесконечно долго).

В идеале стоило бы также обеспечить и *безопасность*, гарантируя, что в любой момент будет *не более одного* лидера, и исключая вероятность возникновения ситуации *разделения роли лидера* (когда избираются два лидера, служащие одной цели, которые не знают о существовании друг друга). Однако на практике многие алгоритмы выбора лидера нарушают это соглашение.

Процессы-лидеры могут использоваться, например, для обеспечения общего порядка сообщений в широковещательной рассылке. Лидер собирает и хранит глобальное состояние, получает сообщения и распространяет их среди процессов. Его также можно использовать, чтобы координировать реорганизацию системы после отказа, во время инициализации или когда происходят важные изменения состояния.

Выбор лидера запускается, когда система инициализируется и лидер выбирается впервые или когда предыдущий лидер аварийно завершается или теряется связь с ним. Этот процесс должен быть детерминированным, приводя к появлению ровно одного лидера. Это решение должно иметь силу для всех участников.

Хотя выбор лидера и распределенная блокировка (т. е. эксклюзивное право доступа к общему ресурсу) с теоретической точки зрения, может, и выглядят одинаково, они несколько различаются. Если один процесс удерживает блокировку для выполнения

критического участка кода, другим процессам не так важно знать, кто именно удерживает блокировку прямо сейчас, если сохраняется свойство живучести (т. е. блокировка будет рано или поздно освобождена и станет доступна для установки другими процессами). В отличие от этого, при выборе процесса в качестве лидера он обладает некоторыми особыми свойствами и должен быть известен всем другим участникам, поэтому только что выбранный лидер должен уведомить другие процессы о своей роли.

Если алгоритм распределенной блокировки отдает определенное предпочтение какому-либо процессу или группе процессов, это приведет к тому, что остальные процессы начнут испытывать недостаток в общем ресурсе, что противоречит свойству живучести. В отличие от этого, лидер может оставаться в своей роли до остановки или аварийного завершения; при этом предпочтительна длительная работа лидеров.

Наличие неизменного лидера в системе помогает обойтись без синхронизации состояний между удаленными участниками, сократить количество передаваемых сообщений и управлять выполнением из одного процесса без необходимости в координации одноранговых процессов. Одна из потенциальных проблем систем, использующих концепцию лидерства, заключается в том, что процесс-лидер может стать узким местом. Для решения этой проблемы многие системы разбивают данные на непересекающиеся независимые наборы реплик (см. раздел «Секционирование базы данных» на с. 289). Вместо одного общесистемного лидера у каждого набора реплик есть свой лидер. Этот подход, к примеру, используется в протоколе Spanner (см. раздел «Распределенные транзакции с использованием протокола Spanner» на с. 286).

Поскольку каждый процесс-лидер рано или поздно дает сбой, система должна обнаружить этот сбой, сообщить о нем и отреагировать на него, т. е. выбрать другого лидера взамен отказавшего.

Некоторые алгоритмы, такие как ZAB (см. подраздел «Протокол атомарной рассылки ZooKeeper» на с. 302), Мульти-Паксос (см. подраздел «Мульти-Паксос» на с. 310) или Raft (см. раздел «Raft» на с. 320), используют временных лидеров для сокращения количества сообщений, необходимых для достижения соглашения между участниками. Однако эти алгоритмы используют собственные специфические средства для выбора лидера, обнаружения отказов и разрешения конфликтов между конкурирующими процессами-лидерами.

Алгоритм забияки

Один из алгоритмов выбора лидера, известный как *алгоритм забияки* (bully algorithm), использует для идентификации нового лидера ранги процессов. Каждому процессу присваивается уникальный ранг. В ходе выбора лидером становится процесс с самым высоким рангом [MOLINA82].

Этот алгоритм известен своей простотой. Он назван «алгоритмом забияки» по той причине, что узел с наивысшим рангом «хулиганским образом» навязывает себя другим узлам. Такой подход также известен как *выбор монарха* (monarchical leader

election): одноуровневый элемент с наивысшим рангом становится монархом после того, как прекращает существование предыдущий монарх.

Выбор производится, когда один из процессов замечает, что в системе нет лидера (система еще не проходила инициализацию) или что предыдущий лидер перестал отвечать на запросы. Этот процесс выполняет следующие три шага[1]:

1. Процесс отправляет сообщения о выборе процессам с более высокими идентификаторами.
2. Процесс ожидает, позволяя процессам более высокого ранга ответить. Если ни один процесс с более высоким рангом не отвечает, он переходит к шагу 3. В противном случае процесс отправляет уведомление процессу с наивысшим рангом (из тех, от которых он получил ответ) и позволяет ему перейти к шагу 3.
3. Процесс предполагает отсутствие активных процессов с более высоким рангом и уведомляет все процессы с более низким рангом о новом лидере.

Рисунок 10.1 иллюстрирует алгоритм забияки:

а) Процесс 3 замечает аварийное завершение предыдущего лидера 6 и запускает новый тур выбора, отправляя сообщения «Выбор» процессам с более высокими идентификаторами.

б) Процессы 4 и 5 отвечают «Активен», так как они имеют более высокий ранг, чем 3.

в) Процесс 3 уведомляет процесс 5, который обладает самым высоким рангом (из тех, что ответили в течение этого тура).

г) Процесс 5 выбирается в качестве нового лидера. Он рассылает сообщения «Выбран», уведомляя процессы с более низким рангом о результате выбора.

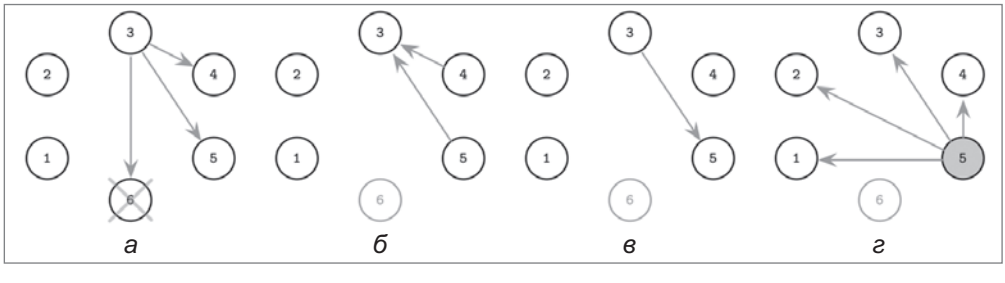

Рис. 10.1. Алгоритм забияки: происходит отказ предыдущего лидера (6) и процесс 3 запускает новый тур выбора

Одна из очевидных проблем этого алгоритма состоит в том, что он не гарантирует безопасность (т. е. что в любой момент будет не более одного лидера) при наличии распада сети. Довольно легко оказаться в ситуации, когда узлы разделяются на два

[1] Здесь описывается *модифицированный* алгоритм забияки [KORDAFSHARI05], поскольку он более компактен и понятен.

или более независимо функционирующих подмножества, каждое из которых выбирает своего лидера. Эта ситуация называется *разделением роли лидера*.

Другой проблемой этого алгоритма является то, что узлам с высокими рангами отдается безусловное предпочтение, что становится проблемой в случае их нестабильности и может привести к постоянному повторению процесса выбора. Нестабильный узел с высоким рангом предлагает себя в качестве лидера, вскоре после этого дает сбой, выигрывает повторный тур выбора, снова дает сбой, и этот процесс повторяется снова. Эту проблему можно решить путем распространения показателей качества узла и принятия их во внимание при выборе лидера.

Аварийное переключение к следующему в очереди

Существует много версий алгоритма забияки, призванных улучшить те или иные его свойства. Например, мы можем использовать несколько следующих в очереди процессов в качестве резервного варианта, чтобы тем самым ускорить выбор лидера [GHOLIPOUR09].

Каждый выбранный лидер предоставляет список резервных узлов. Когда один из процессов обнаруживает отказ лидера, он запускает новый тур выбора, отправляя сообщение альтернативному процессу с наивысшим рангом из списка, предоставленного отказавшим лидером. Если один из предложенных альтернативных процессов работает, он становится новым лидером без необходимости в проведении полного тура выбора.

Если процесс, который обнаружил отказ лидера, сам является процессом с наивысшим рангом в списке, он может немедленно уведомить процессы о том, что он — новый лидер.

Рисунок 10.2 показывает процесс с описанной оптимизацией:

а) Процесс `6`, лидер с обозначенными альтернативами `{5, 4}`, аварийно завершается. Процесс `3` замечает этот отказ и связывается с процессом `5` — альтернативным узлом с самым высоким рангом в списке.

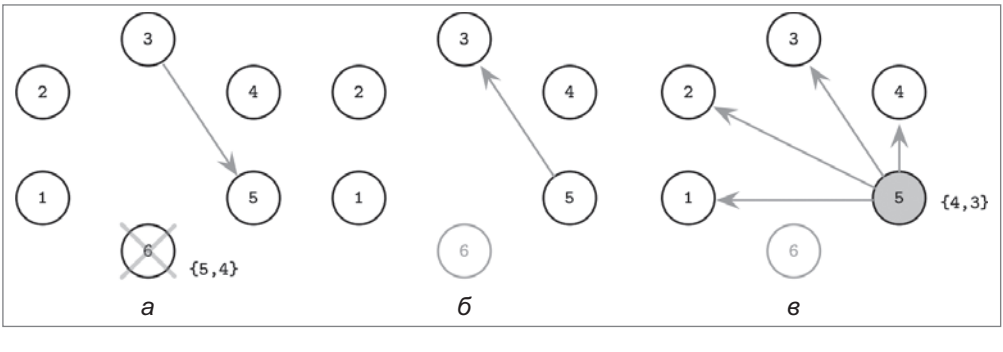

Рис. 10.2. Алгоритм забияки с аварийным переключением: после аварийного завершения работы предыдущего лидера (6) процесс 3 начинает новый тур выбора, связываясь с альтернативным процессом с самым высоким рангом

б) Процесс 5 отвечает процессу 3, что он активен, чтобы процесс 3 не связывался с другими узлами из списка альтернатив.

в) Процесс 5 уведомляет другие узлы, что он — новый лидер.

В результате нам требуется выполнять меньше шагов в ходе выбора, если следующий в очереди процесс активен.

Оптимизация с кандидатами и обычными узлами

Другой алгоритм пытается снизить количество необходимых сообщений за счет разбиения узлов на два подмножества — подмножество *кандидатов* и подмножество *обычных узлов*. При этом стать в итоге лидером может только один из узлов-кандидатов [MURSHED12].

Обычный процесс инициирует выбор, связываясь с узлами-кандидатами, собирая ответы от них, выбирая активного кандидата с наивысшим рангом в качестве нового лидера, а затем уведомляя остальные узлы о результате выбора.

Чтобы исключить вероятность одновременного запуска нескольких процессов выбора, данный алгоритм предусматривает использование переменной разрешения конфликтов δ. Это задержка, которая задается для каждого процесса в отдельности и может сильно варьировать, что позволяет одному из узлов инициировать выбор раньше других. Время разрешения конфликтов обычно больше, чем время прохождения запроса и подтверждения. Узлы с более высокими приоритетами имеют меньшее значение δ, и наоборот.

На рис. 10.3 показаны этапы процесса выбора:

а) Процесс 4 из обычного подмножества замечает отказ процесса-лидера 6. Он начинает новый тур выбора, связываясь со всеми остальными процессами из подмножества кандидатов.

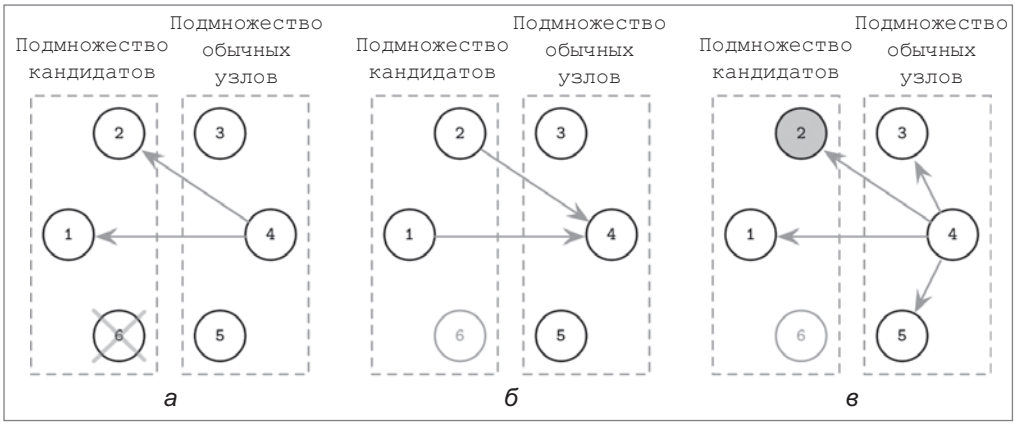

Рис. 10.3. Модификация алгоритма забияки с выделением кандидатов и обычных процессов: после аварийного завершения работы предыдущего лидера (6) процесс 4 начинает новый тур выбора

б) Процессы-кандидаты уведомляют процесс 4 о том, что они по-прежнему активны.

в) Процесс 4 уведомляет все процессы о новом лидере — процессе 2.

Алгоритм с приглашениями

Алгоритм с приглашениями (invitation algorithm) позволяет процессам «приглашать» другие процессы в свою группу, вместо того чтобы соперничать с ними на основе ранга. Этот алгоритм *по определению* допускает наличие нескольких лидеров, поскольку каждая группа имеет своего лидера.

Каждый процесс начинает свою работу как лидер новой группы, в которой единственным участником является он сам. Лидеры групп связываются с одноранговыми процессами, которые не принадлежат к их группе, приглашая их присоединиться. Если одноранговый процесс сам является лидером, две группы объединяются. В противном случае этот процесс отправляет в ответ идентификатор лидера группы, что позволяет двум лидерам групп установить контакт и объединить группы за меньшее количество шагов.

На рис. 10.4 показаны этапы выполнения алгоритма с приглашениями:

а) Четыре процесса начинают свою работу как лидеры групп, содержащих по одному участнику в каждой. Процесс 1 приглашает процесс 2 присоединиться к своей группе, а процесс 3 приглашает присоединиться процесс 4.

б) Процесс 2 присоединяется к группе с процессом 1, а процесс 4 присоединяется к группе с процессом 3. Процесс 1, лидер первой группы, связывается с процессом 3, лидером другой группы. Оставшиеся участники группы (в данном случае — процесс 4) уведомляются о новом лидере группы.

в) Две группы объединяются, и процесс 1 становится лидером расширенной группы.

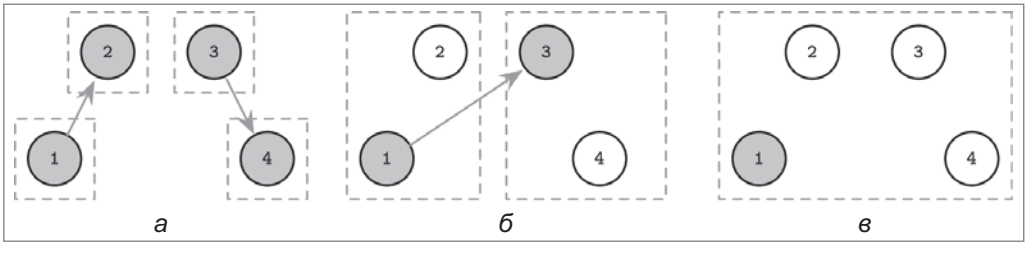

Рис. 10.4. Алгоритм с приглашениями

Поскольку группы объединяются, не имеет значения, становится ли новым лидером процесс, который предложил объединение групп, или другой процесс. Чтобы свести к минимуму количество сообщений, необходимых для объединения групп, лидер большей группы может стать лидером новой группы. Таким образом, о смене лидера должны уведомляться только процессы из меньшей группы.

Подобно другим обсуждаемым алгоритмам, этот алгоритм позволяет процессам объединяться в несколько групп и допускает наличие нескольких лидеров. Алгоритм с приглашениями позволяет создавать группы процессов и объединять их без необходимости в запуске полного нового тура выбора, уменьшая количество сообщений, необходимых для проведения выбора.

Кольцевой алгоритм

В кольцевом алгоритме (ring algorithm) [CHANG79] все узлы системы образуют кольцо и знают о кольцевой топологии (т. е. им известно, какие узлы предшествуют и следуют за ними в кольце). Когда процесс обнаруживает отказ лидера, он начинает новый тур выбора. Сообщение о выборе пересылается по кольцу: каждый процесс связывается со следующим ближайшим узлом в кольце. Если этот узел недоступен, процесс пропускает его и пытается связаться с узлами, расположенными далее по кольцу, пока какой-либо узел не даст ответ.

Узлы связываются с одноуровневыми узлами, следуя по кольцу и собирая множество активных узлов, добавляя себя в это множество перед передачей его следующему узлу, подобно тому как в алгоритме обнаружения отказов, описанном в подразделе «Детектор отказов без времени ожидания» на с. 216, узлы добавляют свои идентификаторы к пути перед передачей этого пути следующему узлу.

Данный алгоритм выполняет полный обход кольца. Когда сообщение доходит по кругу до того узла, который начал процесс выбора, в качестве лидера выбирается узел с наивысшим рангом из множества активных узлов. На рис. 10.5 показан пример такого обхода:

а) В момент аварийного завершения работы предыдущего лидера 6 каждый процесс рассматривает кольцо со своей точки зрения.

б) Процесс 3 инициирует тур выбора, начиная обход. На каждом шаге мы имеем множество узлов, пройденных по пути на данный момент. Процесс 5 не может

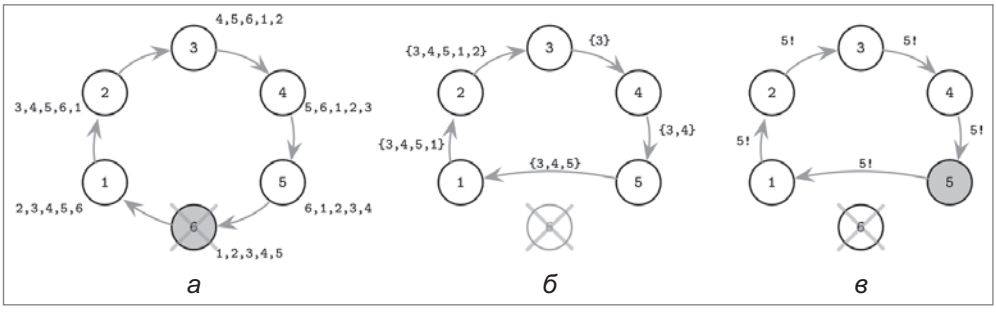

Рис. 10.5. Кольцевой алгоритм: после аварийного завершения работы предыдущего лидера (6) узел 3 начинает процесс выбора

связаться с процессом 6, поэтому он пропускает его и обращается напрямую к процессу 1.

в) Поскольку процесс 5 был узлом с наивысшим рангом, процесс 3 инициирует еще один цикл сообщений, распространяя информацию о новом лидере.

Существует разновидность этого алгоритма, которая для экономии места вместо сбора множества активных узлов предусматривает сохранение одного идентификатора узла с наивысшим рангом: поскольку функция поиска максимума `max` является коммутативной, нам достаточно знать текущий максимум. Когда алгоритм доходит по кругу до того узла, который начал процесс выбора, последний известный наивысший идентификатор еще раз распространяется по кольцу.

Поскольку кольцо может быть разделено на две или более частей таким образом, что в каждой части будет избран свой лидер, этот подход также не обеспечивает свойство безопасности.

Как видите, для корректного функционирования системы с лидером нам необходимо знать статус текущего лидера (активен он или нет), поскольку для поддержания организованности процессов и для продолжения выполнения лидер должен исполнять свои обязанности, будучи активным и доступным. Для обнаружения аварийного завершения работы лидера можно использовать алгоритмы обнаружения отказов (см. главу 9).

Итоги

Выбор лидера является важным аспектом распределенных систем, поскольку использование назначенного лидера помогает снизить затраты на координацию и повысить производительность алгоритма. Туры выбора могут быть затратными, но поскольку они проводятся нечасто, то не оказывают негативного влияния на общую производительность системы. Единственный лидер может стать узким местом, но в большинстве случаев эта проблема решается путем секционирования данных с использованием отдельных лидеров для каждой секции или разных лидеров для различных действий.

К сожалению, все алгоритмы, которые мы обсудили в этой главе, подвержены проблеме разделения роли лидера: мы можем получить двух лидеров в независимых подсетях, которые не будут знать о существовании друг друга. Чтобы избежать разделения роли лидера, нам нужно получить большинство голосов в рамках всего кластера.

Многие алгоритмы достижения консенсуса, в том числе мульти-Паксос и Raft, используют лидера для координации. Однако надо сказать, выбор лидера и достижение консенсуса — это практически одно и то же. Чтобы выбрать лидера, нам нужно прийти к консенсусу относительно его кандидатуры. Если мы сможем достичь консенсуса в отношении кандидатуры лидера, мы сможем использовать те же средства для достижения консенсуса по любому другому вопросу [ABRAHAM13].

Лидер может смениться таким образом, что процессы не узнают об этом, что поднимает вопрос об актуальности локальной информации процесса о лидере. Для решения этой проблемы мы должны объединить выбор лидера с обнаружением отказов. Например, алгоритм *выбора стабильного лидера* использует туры выбора уникального стабильного лидера и обнаружение отказов на основе времени ожидания, чтобы гарантировать сохранение лидером своего статуса до тех пор, пока он будет работоспособным и доступным [AGUILERA01].

Алгоритмы, использующие выбор лидера, часто допускают существование нескольких лидеров, обеспечивая при этом максимально быстрое разрешение конфликтов между лидерами. Например, это верно для алгоритма мульти-Паксос (см. подраздел «Мульти-Паксос» на с. 310), который позволяет продолжить работу только одному из двух конфликтующих лидеров (претендентов); эти конфликты разрешаются путем сбора второго кворума, дающего гарантию в том, что не будут приняты значения двух разных претендентов.

В алгоритме Raft (см. раздел «Raft» на с. 320) лидер может обнаружить, что его срок действия истек (это подразумевает наличие другого лидера в системе), и указать в качестве него более позднее время.

В обоих случаях наличие лидера — это способ обеспечить *живучесть* (когда текущий лидер дает сбой, нам требуется новый лидер), поэтому процессы не должны тратить неограниченно много времени на выяснение того, действительно ли дал сбой текущий процесс лидера. Отсутствие *безопасности* и наличие нескольких лидеров являются средством оптимизации производительности. При этом алгоритм может продолжить работу, выполнив фазу репликации, а безопасность гарантируется путем обнаружения и разрешения конфликтов.

В главе 14 мы еще подробно обсудим консенсус и выбор лидера в контексте достижения консенсуса.

Дополнительная литература

Если вы хотите узнать больше о концепциях, упомянутых в этой главе, вы можете обратиться к следующим источникам:

Алгоритмы выбора лидера

Lynch, Nancy and Boaz Patt-Shamir. 1993. “Distributed algorithms.” Lecture notes for 6.852. Cambridge, MA: MIT.

Attiya, Hagit and Jennifer Welch. 2004. Distributed Computing: Fundamentals, Simulations and Advanced Topics. USA: John Wiley & Sons.

Tanenbaum, Andrew S. and Maarten van Steen. 2006. Distributed Systems: Principles and Paradigms (2nd Ed.). Upper Saddle River, NJ: Prentice-Hall[1].

[1] *Таненбаум Э., Стеен М. ван.* Распределенные системы. Принципы и парадигмы. СПб.: Питер, 2003. — *Примеч. ред.*

ГЛАВА 11

Репликация и согласованность

Прежде чем продолжить обсуждение консенсуса и алгоритмов атомарной фиксации, давайте соберем в единое целое последний элемент, необходимый для их глубокого понимания: *модели согласованности* (consistency models). Модели согласованности важны, поскольку проясняют семантику видимости и поведение системы при наличии нескольких копий данных.

Отказоустойчивость (fault tolerance) — это свойство системы, способной продолжать функционирование в случае отказа некоторых ее частей. Обеспечить отказоустойчивость системы — непростая задача, особенно в случае уже существующей системы. Основная цель при этом сводится к тому, чтобы устранить единую точку отказа в системе и обеспечить избыточность критически важных компонентов. Обычно такая избыточность полностью прозрачна для пользователя.

Система может продолжать корректное функционирование за счет сохранения нескольких копий данных, что позволяет в случае отказа одной из машин произвести аварийное переключение на другую машину. В системе с единым источником истины (например, с первичной базой данных и репликой) аварийное переключение может выполняться явным образом, путем наделения реплики статусом основной базы данных. Другие системы не требуют явного изменения конфигурации, обеспечивая согласованность путем сбора ответов от различных участников при выполнении запросов на чтение и запись.

Репликация (replication) данных — это способ обеспечения избыточности, сводящийся к поддержанию в системе нескольких копий данных. Однако поскольку задача атомарного обновления нескольких копий данных эквивалентна задаче достижения консенсуса [MILOSEVIC11], часто слишком затратно выполнять обновление при выполнении буквально *каждой* операции в базе данных. Нам следует поискать более экономичные и гибкие способы обеспечения того, чтобы данные выглядели для пользователя согласованными, допуская при этом некоторую степень расхождения между участниками.

Репликация особенно важна для систем, развернутых в нескольких центрах данных. В таких случаях георепликация решает сразу несколько задач — она повышает доступность и устойчивость к отказу одного или нескольких центров данных за счет поддержания избыточности и также может помочь в уменьшении задержки за счет размещения копии данных ближе к клиенту.

При модификации записей данных необходимо соответственно обновлять и их копии. В случае репликации нас прежде всего интересуют три события: *запись*, *обновление реплик* и *чтение*. Эти операции запускают некоторую последовательность иниции-

руемых клиентом событий. В некоторых случаях обновление реплик может производиться после завершения операции записи с точки зрения клиента, но при этом клиент все равно должен иметь возможность видеть операции в определенном порядке.

Обеспечение доступности

Мы говорили о проблемах распределенных систем и выделили многое из того, что может пойти не так. В реальности узлы не всегда активны или доступны в сети. Однако случающиеся время от времени отказы не должны сказываться на *доступности* системы: с точки зрения пользователя, система в целом должна продолжать функционирование, как если бы ничего не случилось.

Доступность системы — невероятно важное свойство: в сфере программной разработки мы всегда стремимся обеспечить высокую степень доступности и минимизировать время простоя. Команды разработчиков хвастаются тем, насколько высокие показатели работоспособности им удалось обеспечить. Столь сильная забота о доступности обусловлена целым рядом причин: программное обеспечение стало неотъемлемой частью нашего общества, и без него невозможны многие важные вещи: банковские переводы, связь, путешествия и т. д.

Для компаний недостаток в доступности может привести к потере клиентов и денег: нельзя сделать покупку в неработающем интернет-магазине и выполнить перевод денег, если сайт банка не отвечает.

Чтобы система была высокодоступной, ее необходимо спроектировать так, чтобы отказ или потеря доступности одного или нескольких участников не приводили к потере устойчивости. Этого можно добиться, введя в систему избыточность и репликацию. Однако при добавлении избыточности мы сталкиваемся с проблемой синхронизации нескольких копий данных и должны предусмотреть механизмы восстановления.

Печально известная теорема CAP

Доступность (availability) — свойство, отражающее способность системы успешно отвечать на все поступающие запросы. При этом в теории подразумевается способность давать ответ в конечном итоге, но, разумеется, в реальной системе желательно обходиться без служб, выдающих ответ через неопределенно долгое время.

В идеале все операции должны быть *согласованными* (consistent). Под согласованностью здесь понимается атомарная, или *линеаризуемая*, согласованность (см. подраздел «Линеаризуемость» на с. 241). Линеаризуемая история выполнения может быть представлена в виде последовательности мгновенных операций, отражающей исходный порядок выполнения. Линеаризуемость упрощает рассмотрение возможных состояний системы и помогает представить распределенные системы, как если бы они работали на одной машине.

Нам бы хотелось обеспечить согласованность и доступность одновременно с устойчивостью к распаду сети. В случае распада сети процессы, выполняемые в разных ее

частях, не могут связываться друг с другом: некоторые из сообщений, пересылаемых между разделенными узлами, не будут достигать адресата.

Доступность подразумевает, что любой работающий узел должен выдавать результаты, а согласованность подразумевает, что результаты должны быть линеаризуемыми. В гипотезе CAP, сформулированной Эриком Брюером, рассматриваются компромиссы между согласованностью, доступностью и устойчивостью к распаду сети [BREWER00].

Доступность невозможно обеспечить в асинхронной системе, и мы не сможем реализовать систему, которая одновременно гарантировала бы и *доступность*, и *согласованность* при наличии *распада сети* (network partition) [GILBERT02]. Мы можем создать систему, гарантирующую строгую согласованность с обеспечением *максимально возможной* доступности или доступность с обеспечением *максимально возможной* согласованности [GILBERT12]. Определение «максимально возможная» здесь подразумевает то, что если все работает как следует, система не будет целенаправленно нарушать гарантии, но гарантии могут быть ослаблены или нарушены в случае распада сети.

Другими словами, теорема CAP описывает следующие неразрывно связанные полярные варианты системы:

Согласованная и устойчивая к распаду сети (CP-system)

: СУ-система предпочитает не выполнить запрос, вместо того чтобы выдать потенциально несогласованные данные.

Доступная и устойчивая к распаду сети (AP-System)

: ДУ-система ослабляет требования к согласованности и допускает выдачу в ответ на запрос потенциально несогласованных значений.

Примером СУ-системы может служить реализация алгоритма достижения консенсуса, требующая работоспособности большинства узлов: такая система будет всегда согласованной, но может оказаться недоступной в случае распада сети. База данных, которая всегда принимает запросы на запись и выполняет запросы на чтение, пока функционирует хотя бы одна реплика, является примером ДУ-системы, работа которой может привести к потере данных или выдаче несогласованных результатов.

Согласно гипотезе PACELEC [ABADI12], которая является расширением теоремы CAP, при распаде сети есть выбор между согласованностью и доступностью (partitions, consistency, availability, PAC — распад, согласованность, доступность). Иначе (E — else), даже если система функционирует нормально, нам *все равно* придется выбирать между задержкой (L — latency) и согласованностью (C — consistency).

Осмотрительное использование теоремы CAP

Важно заметить, что теорема CAP рассматривает *распад сети*, а не *аварийное завершение работы узлов* или какие-либо другие варианты отказов (такие, как сбой с последующим восстановлением). Узел, отделенный от остальной части кластера, может

выдать в ответ на запрос несогласованные данные, но аварийно завершившийся узел не ответит вообще. С одной стороны, это означает, что проблемы с согласованностью могут возникнуть, даже если все узлы исправны. С другой стороны, в реальности все происходит иначе: существует много разных сценариев отказов (некоторые из которых можно смоделировать с помощью распада сети).

Теорема CAP подразумевает, что мы можем столкнуться с проблемами согласованности, даже если все узлы работают исправно, но имеются проблемы с обеспечением связи между ними, так как мы рассчитываем на то, что каждый работающий узел будет отвечать корректно вне зависимости от того, сколько узлов могут оказаться неработоспособными.

Гипотеза CAP иногда изображается в виде треугольника, как если бы мы могли «повернуть ручку» и получить в той или иной степени все три параметра. Хотя мы можем и «повернуть ручку», и пожертвовать согласованностью ради доступности, в реальной системе невозможно каким-либо образом настроить или обменять на что-либо свойство устойчивости к распаду сети [HALE10].

В теореме CAP согласованность понимается несколько иначе, чем в ACID-свойствах (см. главу 5). В ACID-свойствах под согласованностью понимается согласованность транзакций: транзакция переводит базу данных из одного допустимого состояния в другое допустимое состояние, сохраняя все инварианты базы данных (такие, как ограничения уникальности и ссылочная целостность). В теореме CAP согласованность означает, что операции являются атомарными (т. е. завершаются успешно или не выполняются как единое целое) и *согласованными* (т. е. никогда не оставляют данные в несогласованном состоянии).

Под доступностью в теореме CAP также понимается нечто иное, нежели ранее упомянутое свойство *высокой доступности* [KLEPPMANN15]. Здесь доступность не подразумевает никаких ограничений на задержку выполнения. Кроме того, в базах данных, в отличие от теоремы CAP, доступность не подразумевает, что *каждый* исправно работающий узел должен отвечать на *каждый* запрос.

Гипотеза CAP используется для описания распределенных систем, рассмотрения сценариев отказа и оценки возможных ситуаций, но важно помнить, что существует тонкая грань между отказом от согласованности и предоставлением непредсказуемых результатов.

При правильном подходе даже те базы данных, которые отдают предпочтение доступности, способны выдавать согласованные результаты на основе реплик при наличии достаточного количества активных реплик. Конечно, существуют и более сложные сценарии отказов; при этом гипотеза CAP является лишь эмпирическим правилом, которое не всегда отражает реальное положение дел[1].

[1] Кворум узлов производит чтение и запись в контексте согласованных в конечном счете хранилищ, которые будут подробно рассмотрены в разделе «Согласованность в конечном счете» на с. 252.

Плоды и урожай

Гипотеза CAP подразумевает согласованность и доступность в их строгой форме, т. е. *линеаризуемость* и способность системы в конечном итоге ответить на каждый запрос.

Это заставляет нас делать непростой выбор между этими свойствами. Однако в случае некоторых применений бывает полезно использовать не столь строгие допущения, рассматривая более слабые формы этих свойств.

Вместо обеспечения строгой согласованности *или* доступности системы могут обеспечивать менее строгие гарантии. Мы можем определить два настраиваемых параметра: *полноту ответа* и *результативность*, при выборе которых будет все равно обеспечиваться корректное поведение [FOX99]:

Полнота ответа (harvest)

Отражает степень выполнения запроса: если в ответ на запрос нужно возвратить 100 строк, но из-за недоступности некоторых узлов мы можем выдать только 99 строк, обычно лучше выдать такой ответ, чем полностью отказаться от выполнения запроса, не возвращая ничего.

Результативность (yield)

Отражает соотношение между количеством успешно выполненных запросов и общим количеством попыток выполнения запросов. Результативность — это не то же самое, что длительность работоспособного состояния, поскольку, например, занятый определенной работой узел остается работоспособным, но может при этом не отвечать на некоторые запросы.

Таким образом, мы можем говорить о компромиссе не в абсолютных, а в относительных категориях. Мы можем найти баланс между полнотой ответа и результативностью, допуская возвращение неполных данных в ответ на запросы. Один из способов повышения результативности при этом состоит в том, чтобы возвращать результаты только из доступных секций (см. раздел «Секционирование базы данных» на с. 289). Например, если подмножество узлов, хранящих записи некоторых пользователей, находится в нерабочем состоянии, мы все равно можем выполнять запросы на выдачу данных о других пользователях. В качестве альтернативы мы можем требовать, чтобы критические данные приложения возвращались только как единое целое, допуская некоторые отклонения для других запросов.

Определение, оценка и осознанная настройка показателей полноты ответа и результативности помогают нам создавать системы с более высокой устойчивостью к отказам.

Общая память

С точки зрения клиента, распределенная система хранения данных работает так, как если бы она имела общее хранилище, подобно системе с одним узлом. Связь и передача сообщений между узлами абстрагированы и скрыты от внешнего наблюдателя, что создает иллюзию использования общей памяти.

Отдельная единица хранения, доступная для операций чтения или записи, обычно называется *регистром*. Мы можем рассматривать *общую память* распределенной базы данных как массив таких регистров.

Мы идентифицируем каждую операцию по событиям ее *вызова* и *завершения*. Операция считается *«сбойной»*, если вызвавший ее процесс дает сбой до своего завершения. Если события вызова и завершения одной операции происходят до вызова второй операции, мы говорим, что первая операция *предшествует* второй операции и эти две операции являются *последовательными*. В противном случае они считаются *параллельными*.

На рис. 11.1 вы можете видеть процессы P_1 и P_2, выполняющие различные операции:

а) Операция, выполняемая процессом P_2, начинается *после завершения* операции, выполняемой процессом P_1, и эти две операции являются *последовательными*.

б) Здесь две операции частично перекрываются и потому являются *параллельными*.

в) Операция, выполняемая процессом P_2, начинается *после* и завершается *до* операции, выполняемой процессом P_1. Эти операции также являются *параллельными*.

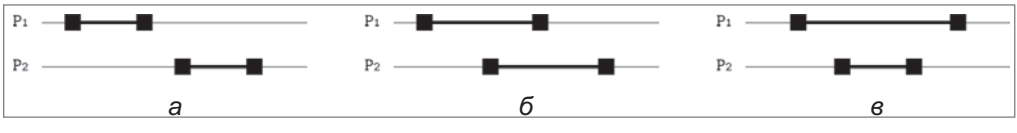

Рис. 11.1. Последовательные и параллельные операции

Несколько операций чтения или записи могут получить доступ к регистру одновременно. Операции чтения и записи в регистры *не являются мгновенными* и занимают некоторое время. Параллельные операции чтения и записи, выполняемые различными процессами, не являются *последовательными*: они могут упорядочиваться различным образом и выдавать разные результаты в зависимости от того, как ведут себя регистры, когда операции перекрываются. В зависимости от поведения регистра при наличии параллельных операций выделяют три типа регистров:

Безопасный (safe)

Операции чтения из безопасных регистров могут возвращать *произвольные* значения в пределах диапазона регистра при наличии параллельной операции записи (что кажется малопригодным на практике, но может описывать семантику асинхронной системы, которая не задает порядок). Для безопасных регистров с двоичными значениями часто характерно «дрожание» (flickering) при выполнении операций чтения параллельно с операциями записи (т. е. попеременное возвращение двух чередующихся значений).

Регулярный (regular)

В случае регулярных регистров мы имеем чуть более строгие гарантии: операция чтения может вернуть только значение, записанное самой последней *завершенной*

операцией записи, или значение, записанное операцией записи, которая накладывается на текущую операцию чтения. В данном случае в системе предусматривается некоторый порядок, но результаты записи не видны всем операциям чтения одновременно (например, так может происходить в случае реплицируемой базы данных, когда основная база данных принимает операции записи и реплицирует их на рабочие копии, выполняющие операции чтения).

Атомарный (atomic)

Атомарные регистры гарантируют линеаризуемость: каждая операция записи подразумевает определенный момент, перед которым каждая операция чтения возвращает прежнее значение и после которого каждая операция чтения возвращает новое. Атомарность — это фундаментальное свойство, упрощающее рассмотрение состояний системы.

Упорядочение

Если мы видим последовательность событий, то нам интуитивно понятен порядок их прохождения. Однако в распределенной системе это не всегда так просто, потому что трудно понять, *когда именно* что-то произошло, и получить эту информацию мгновенно по всему кластеру. У каждого участника может быть свое представление о состоянии, поэтому нам нужно рассмотреть каждую операцию и определить ее в терминах событий *вызова* и *завершения*, тем самым задав ее границы.

Давайте определим систему, в которой процессы могут выполнять операции чтения `read(регистр)` и записи `write(регистр, значение)` над совместно используемыми регистрами. Каждый процесс последовательно выполняет свой собственный набор операций (т. е. каждая вызываемая операция завершается до запуска следующей операции). В совокупности эти последовательно выполняемые наборы каждого процесса образуют глобальную историю, в которой операции могут выполняться параллельно.

Модели согласованности проще всего рассматривать в терминах операций чтения и записи и вариантов их перекрытия: операции чтения не имеют побочных эффектов, а операции записи изменяют состояние регистра. Такой подход помогает понять, когда именно данные становятся читаемыми после записи. Например, рассмотрим историю, в которой два процесса параллельно выполняют следующие события:

```
Процесс 1:     Процесс 2:
write(x, 1)    read(x)
               read(x)
```

При рассмотрении этих событий неясно, каков результат операций `read(x)` в обоих случаях. У нас есть несколько возможных вариантов истории:

- Операция записи завершается до начала обеих операций чтения.
- Операция записи выполняется в промежутке между операциями чтения.
- Обе операции чтения завершаются до начала операции записи.

Нет простого ответа на вопрос, что должно произойти, даже если у нас есть только одна копия данных. В реплицируемой системе у нас больше комбинаций возможных состояний, и ситуация может стать еще сложнее, если у нас есть несколько процессов, читающих и записывающих данные.

Если бы все эти операции выполнялись одним процессом, мы могли бы установить строгий порядок событий, но это сложнее сделать в ситуации с несколькими процессами. Потенциальные трудности можно разделить на две группы:

- Операции могут перекрываться.
- Эффекты неперекрывающихся вызовов могут стать видимыми не сразу.

Чтобы у нас была возможность рассматривать порядок операций и получать однозначные описания возможных результатов, необходимо определить модели согласованности. Мы рассматриваем параллелизм в распределенных системах в терминах общей памяти и параллельных систем, поскольку большинство определений и правил, определяющих согласованность, здесь также применимы. Несмотря на то что значительная часть терминологии параллельных и распределенных систем совпадает, нельзя напрямую применять большинство параллельных алгоритмов из-за различий в моделях связи, производительности и надежности.

Модели согласованности

Поскольку операции над регистрами совместно используемой памяти могут перекрываться, мы должны определить четкую семантику: что происходит, если несколько клиентов читают или изменяют разные копии данных одновременно или в рамках короткого периода. Нет единственно правильного ответа на этот вопрос, поскольку эта семантика изменяется в зависимости от области применения, но она хорошо изучена в контексте моделей согласованности.

Модели согласованности предоставляют разную семантику и гарантии. Их можно рассматривать как договор между участниками: что должна делать каждая реплика для соответствия требуемой семантике и чего ожидать пользователям при выполнении операций чтения и записи.

Модели согласованности описывают ожидания клиентов в терминах возможных возвращаемых значений безотносительно к наличию нескольких копий данных и параллельного доступа к ним. В этом разделе мы обсудим модели согласованности *с одной операцией*.

Каждая модель описывает, насколько далеко поведение системы от ожидаемого или «нормального» поведения. Это помогает нам проводить различие между «всеми возможными вариантами истории» чередующихся операций и «вариантами истории, допустимыми согласно модели X», что значительно упрощает рассмотрение вопроса о видимости изменений состояния.

Мы можем рассматривать согласованность с точки зрения *состояния*, определив, какие инварианты состояния являются приемлемыми и какие отношения допустимы

между копиями данных, находящимися в разных репликах. В качестве альтернативы мы можем рассматривать согласованность *операций*, которая показывает внешний вид хранилища данных, описывает операции и накладывает ограничения на порядок их выполнения [TANENBAUM06] [AGUILERA16].

Без глобальных часов трудно задать распределенным операциям точный и детерминированный порядок. Это своего рода специальная теория относительности для данных: каждый участник обладает своим взглядом на состояние и время.

Теоретически мы можем устанавливать блокировку всей системы каждый раз, когда нам нужно изменить состояние системы, но это было бы крайне непрактично. Вместо этого мы используем набор правил, определений и ограничений, которые лимитируют количество возможных историй и результатов.

Модели согласованности добавляют еще одно измерение к тому, что мы обсуждали в разделе «Печально известная теорема CAP» на с. 233. Теперь нам нужно не только найти баланс между согласованностью и доступностью, но и рассмотреть согласованность с точки зрения затрат на синхронизацию [ATTIYA94]. Затраты на синхронизацию могут включать в себя задержку, дополнительные циклы ЦП, затрачиваемые на выполнение дополнительных операций, дисковый ввод-вывод, используемый для сохранения информации для восстановления, время ожидания, сетевой ввод-вывод и все остальное, чего можно избежать, обойдясь без синхронизации.

Сначала мы сфокусируемся на видимости и распространении результатов операций. Возвращаясь к примеру с параллельными операциями чтения и записи, мы сможем ограничить число возможных вариантов истории либо путем размещения зависимых операций записи друг за другом, либо путем определения точки, в которой должно распространяться новое значение.

Мы обсуждаем модели согласованности в терминах *процессов* (клиентов), генерирующих операции чтения и записи, влияющие на состояние базы данных. Поскольку мы обсуждаем согласованность в контексте реплицируемых данных, мы полагаем, что у базы данных может быть несколько реплик.

Строгая согласованность

Строгая согласованность (strict consistency) является эквивалентом полной прозрачности репликации: результат любой операции записи, выполняемой любым процессом, немедленно становится доступным для последующего чтения любым процессом. Она подразумевает использование глобальных часов и утверждает, что если в момент t_1 была выполнена операция записи `write(x, 1)`, то любая операция чтения `read(x)` вернет вновь записанное значение `1` в любой момент $t_2 > t_1$.

К сожалению, это всего лишь теоретическая модель, которую невозможно реализовать в силу того, что законы физики и характер работы распределенных систем накладывают ограничения на скорость течения событий [SINHA97].

Линеаризуемость

Линеаризуемость (linearizability) — это самая строгая модель согласованности с одним объектом и одной операцией. Согласно этой модели, результаты записи становятся видимыми для всех операций чтения ровно один раз в некоторый момент между ее началом и окончанием и ни один клиент не может видеть переходы состояний или побочные результаты частично выполненной (т. е. незавершенной, текущей) или неполной (т. е. прерванной до завершения) операции записи [LEE15].

Параллельные операции представляются в виде одного из возможных вариантов последовательной истории, для которых сохраняются свойства видимости. Линеаризуемость допускает некоторую степень неопределенности, поскольку может существовать несколько вариантов упорядочения событий [HERLIHY90].

Если две операции накладываются друг на друга, их результаты могут становится видимыми в любом порядке. Все операции чтения, которые выполняются после завершения операции записи, могут наблюдать результаты этой операции. После того как одна операция чтения возвращает определенное значение, все последующие операции чтения возвращают значение, возникшее *не раньше* этого возвращенного [BAILIS14a].

Существует некоторая гибкость в плане порядка параллельных событий в глобальной истории, однако их нельзя переупорядочивать произвольно. Результаты операции не должны проявляться до ее начала, поскольку для этого потребовалось бы предсказывать будущие операции. В то же время результаты должны появляться до завершения операции, так как в противном случае мы не сможем определить точку линеаризации.

Линеаризуемость учитывает как порядок последовательных локальных операций процесса, так и порядок операций, выполняемых параллельно относительно других процессов, и определяет *общий порядок* событий.

Этот порядок должен быть *согласованным*, что означает, что каждая операция чтения общего значения должна возвращать самое последнее значение, записанное в эту общую переменную перед этой операцией чтения, или значение операции записи, накладывающейся на эту операцию чтения. Линеаризуемый доступ на запись к общей переменной также подразумевает взаимное исключение: из двух параллельных операций записи только одна может выполняться первой.

Несмотря на то что операции являются параллельными и в некоторой степени накладываются друг на друга, их результаты становятся видимыми таким образом, что они кажутся последовательными. Ни одна операция не происходит мгновенно, но все равно выглядит как атомарная операция.

Давайте рассмотрим следующую историю выполнения:

```
Процесс 1:      Процесс 2:     Процесс 3:
write(x, 1)     write(x, 2)    read(x)
                               read(x)
                               read(x)
```

На рис. 11.2 у нас есть три процесса, два из которых выполняют операции записи в регистр x с начальным значением ∅. Операции чтения могут видеть эти операции записи согласно одному из указанных ниже вариантов:

а) Первая операция чтения может вернуть 1, 2 или ∅ (начальное значение, состояние перед обеими операциями записи), поскольку обе операции записи еще выполняются. Первая операция чтения может быть поставлена *перед* обеими операциями записи, *между* ними и *после* них.

б) Вторая операция чтения может вернуть только 1 и 2, поскольку первая операция записи уже завершилась, а вторая операция записи еще не вернула результат.

в) Третья операция чтения может вернуть только 2, так как вторая операция записи поставлена после первой.

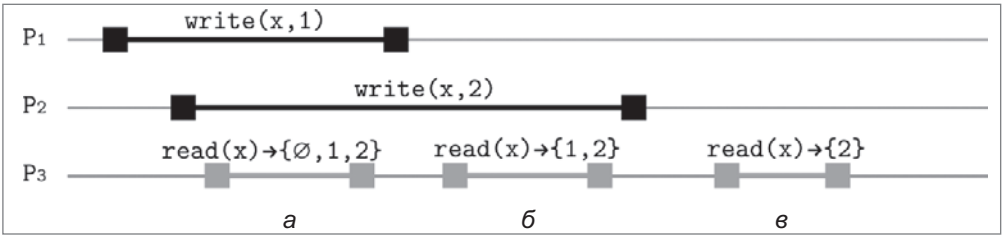

Рис. 11.2. Пример линеаризуемости

Точка линеаризации

Одним из наиболее важных свойств линеаризуемости является видимость: после завершения операции все должны это видеть; при этом система не может «возвращаться в прошлое», откатывая выполнение операции или делая ее невидимой для некоторых участников. Другими словами, линеаризация запрещает чтение устаревших данных и требует, чтобы операции чтения были монотонными.

Эту модель согласованности лучше всего рассматривать в терминах атомарных (т. е. непрерывных, неделимых) операций. При этом операции могут не быть мгновенными (в частности, потому, что таких операций не бывает), но их результаты должны становиться видимыми в определенный момент, создавая иллюзию их мгновенного выполнения. Этот момент называется *точкой линеаризации* (linearization point).

После точки линеаризации операции записи (иными словами, когда значение становится видимым для других процессов) каждый процесс должен видеть либо значение, записанное этой операцией, либо какое-либо более позднее значение, если после нее поставлены некоторые дополнительные операции записи. Видимое значение должно оставаться неизменным до тех пор, пока не станет видимым следующее после него значение, и значение регистра не должно чередоваться между двумя последними состояниями.

Точка линеаризации служит границей, после которой результаты операции становятся видимыми. Мы можем реализовать ее, используя блокировки для защиты

критической секции, атомарные операции чтения и записи или примитивы чтения-модификации-записи.

Большинство языков программирования в наши дни предлагают атомарные примитивы, которые позволяют выполнять атомарные операции записи и операции «сравнение с обменом» (compare-and-swap, CAS). Атомарные операции записи не учитывают текущие значения регистров, в отличие от операций CAS, которые меняют значение, только если предыдущее значение не изменялось [HERLIHY94]. Считывать значение, изменять его, а затем записывать с помощью операции CAS сложнее, чем просто проверить и установить значение, из-за возможной *проблемы ABA* [DECHEV10]: если операция CAS ожидает в регистре значение A, то операция запишет в регистр новое значение, даже если перед этим две другие параллельные операции записи записали значение B, а затем снова записали значение A. Другими словами, наличие значения A само по себе не гарантирует, что значение регистра не менялось с момента последнего чтения.

На рис. 11.3 показано, что линеаризуемость предполагает жесткие временные рамки, где часы отсчитывают *реальное время*, поэтому результаты операции должны становиться видимыми в промежутке между моментом t_1, когда был выдан запрос операции, и моментом t_2, когда процесс получил ответ.

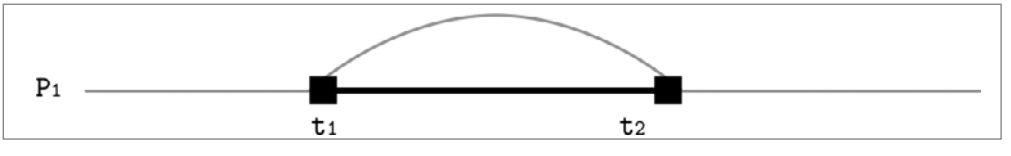

Рис. 11.3. Временные рамки линеаризуемой операции

Рисунок 11.4 показывает, что точка линеаризации делит историю на части, расположенные до и после нее. До точки линеаризации видимо прежнее значение, после нее видимо новое значение.

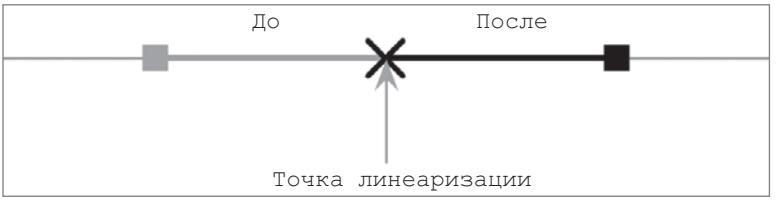

Рис. 11.4. Точка линеаризации

Затраты на обеспечение линеаризуемости

Линеаризуемость не реализуется во многих современных системах. Даже ЦП по умолчанию не обеспечивает линеаризуемость при доступе к оперативной памяти.

Это объясняется тем, что инструкции по синхронизации являются затратными, медленными и включают в себя загрузку ЦП для межузловых действий и аннулирование кэша. Однако линеаризуемость можно реализовать, используя низкоуровневые примитивы [MCKENNEY05a] [MCKENNEY05b].

МНОГОКРАТНО ИСПОЛЬЗУЕМАЯ ИНФРАСТРУКТУРА ДЛЯ ЛИНЕАРИЗАЦИИ

Многократно используемая инфраструктура для линеаризации (RIFL, Reusable infrastructure for linearizability) — это механизм для реализации линеаризуемых удаленных вызовов процедур (RPC) [LEE15]. В RIFL сообщения однозначно идентифицируются с помощью идентификатора клиента и локального монотонно увеличивающегося порядкового номера клиента.

Для присвоения идентификаторов клиентов в RIFL используется *аренда* (lease), выдаваемая общесистемной службой: уникальные идентификаторы, используемые для обеспечения уникальности и установления различия между порядковыми номерами разных клиентов. Если неисправный клиент пытается выполнить операцию с использованием просроченной аренды, его операция не выполняется: клиент должен получить новую аренду и повторить попытку.

Если сервер аварийно завершает работу, не успев подтвердить операцию записи, клиент может попытаться повторить эту операцию, не зная о том, что она уже была применена. В итоге это может привести к следующей ситуации. Клиент `C1` записывает значение `V1` и не получает подтверждения, после чего клиент `C2` записывает значение `V2`. Если `C1` повторит свою операцию и успешно запишет `V1`, результат записи, выполненной клиентом `C2`, будет потерян. Чтобы избежать этого, система должна предотвращать повторное выполнение операций. Когда клиент повторяет операцию, вместо ее повторного применения RIFL возвращает объект завершения, указывающий, что соответствующая операция уже выполнена, и возвращает ее результат.

Объекты завершения хранятся в долговременном хранилище вместе с актуальными записями данных. Однако время их жизни может быть различным: объект завершения должен существовать либо до тех пор, пока клиент-эмитент не пообещает, что не будет повторять связанную с ним операцию, либо до тех пор, пока сервер не обнаружит аварийное завершение клиента, в случае чего можно будет безопасно удалить все связанные с ним объекты завершения. Объект завершения должен создаваться атомарным образом одновременно с изменением той записи данных, с которой он связан.

Клиенты должны периодически обновлять свою аренду, чтобы показать свою активность. Если клиенту не удается возобновить аренду, он помечается как давший сбой и все данные, связанные с его арендой, удаляются процессом сборки мусора. Срок действия аренды ограничивается, чтобы в журнале не хранились бесконечно долго сведения об операциях, относящихся к сбойному процессу. Если давший сбой клиент попытается выполнить операцию, используя просроченную аренду, результаты этой операции не будут зафиксированы и ее потребуется выполнить снова.

Преимущество механизма RIFL состоит в том, что, поскольку удаленный вызов процедур гарантированно не будет выполняться более одного раза, операцию можно сделать линеаризуемой, проследив за тем, чтобы ее результаты делались видимыми атомарным образом и большинство ее деталей реализации не зависело от нижележащей системы хранения.

В сфере параллельного программирования для введения линеаризуемости можно использовать операции «сравнение с обменом» (CAS). Многие алгоритмы сначала *подготавливают* результаты, а затем используют операцию CAS для переключения указателей и их *публикации*. Например, мы можем реализовать параллельную очередь, создавая узел связанного списка и атомарно добавляя его в конец списка [KHANCHANDANI18].

В распределенных системах линеаризуемость требует координации и упорядочения. Линеаризуемость можно реализовать, используя *консенсус*: клиенты взаимодействуют с реплицированным хранилищем посредством сообщений, а модуль консенсуса отвечает за обеспечение согласованности и тождественности применяемых операций в рамках кластера. Результаты каждой операции записи будут появляться мгновенно, ровно один раз в определенный момент между ее вызовом и завершением [HOWARD14].

Любопытным моментом является то, что линеаризуемость в своем традиционном понимании считается *локальным* свойством и подразумевает комбинацию независимо реализованных и проверенных элементов. Комбинация линеаризуемых историй выполнения представляет собой историю, которая также обладает свойством линеаризуемости [HERLIHY90]. Другими словами, система, в которой все объекты линеаризуемы, также является линеаризуемой. Это очень полезное свойство, но нужно помнить, что его область действия ограничивается одним объектом, и даже если операции над двумя независимыми объектами являются линеаризуемыми, операции, которые включают оба объекта, должны использовать дополнительные средства синхронизации.

Последовательная согласованность

Достижение линеаризуемости может оказаться слишком затратным, однако можно ослабить требования, в то же время обеспечивая довольно сильные гарантии согласованности. *Последовательная согласованность* (sequential consistency) позволяет упорядочивать операции, как если бы они выполнялись в некотором последовательном порядке, при этом требуя выполнения операций каждого отдельного процесса в том же порядке, в котором они были выполнены процессом.

Процессы могут видеть операции, выполняемые другими участниками в порядке, соответствующем их собственной истории, хотя это представление может быть произвольно устаревшим с общесистемной точки зрения [KINGSBURY18a]. Порядок выполнения между процессами не определен, так как нет общего понятия времени.

Последовательная согласованность была первоначально введена в контексте конкурентности и описывалась как способ корректного выполнения многопроцессорных программ. В исходном определении требовалось, чтобы запросы памяти к одной и той же ячейке были упорядочены в очереди (FIFO, порядок поступления), не устанавливалось глобальное упорядочение для перекрывающихся операций записи

в независимые ячейки памяти, а операциям чтения позволялось извлекать значение из ячейки памяти или последнее значение из очереди, если очередь была непустой [LAMPORT79]. Этот пример помогает понять семантику последовательной согласованности. Операции могут быть упорядочены разными способами (в зависимости от порядка поступления или даже произвольно, если две записи поступают одновременно), но все процессы видят операции в одном и том же порядке.

Каждый процесс может выдавать запросы на чтение и запись в порядке, указанном его собственной программой, что легко интуитивно понять. Любая однопоточная программа без конкурентности выполняет свои шаги последовательно. Все операции записи, распространяющиеся из одного и того же процесса, становятся видимыми в том порядке, в котором они передаются этим процессом. Операции, распространяющиеся из разных источников, могут располагаться *в произвольном порядке*, но этот порядок будет согласованным с точки зрения операций чтения.

Последовательную согласованность часто путают с линеаризуемостью, поскольку у них схожая семантика. Последовательная согласованность, так же как и линеаризуемость, требует, чтобы операции были глобально упорядочены, но линеаризуемость требует, чтобы локальный порядок каждого процесса и глобальный порядок были согласованными. Другими словами, в линеаризуемости соблюдается порядок операций реального времени. При последовательной согласованности порядок сохраняется только для операций записи одного происхождения [VIOTTI16]. Другое важное отличие относится к комбинированию: мы можем комбинировать линеаризуемые истории, при этом результаты будут также линеаризуемыми, в то время как последовательно согласованные планы выполнения не пригодны для комбинирования [ATTIYA94].

На рис. 11.5 показано, каким образом операции записи `write(x, 1)` и `write(x, 2)` могут стать видимыми для процессов P_3 и P_4. Хотя с точки зрения системных часов значение `1` было записано *раньше* значения `2`, оно может расположиться после значения `2`. В то же время процесс P_3 может считать значение `1` еще до того, как процесс P_4 считает значение `2`. Однако оба порядка, `1 → 2` и `2 → 1`, правомерны, если для них сохраняется согласованность с точки зрения разных операций чтения. Главное, чтобы и процесс P_3, и процесс P_4 видели значения в одном и том же порядке: сначала `2`, а затем `1` [TANENBAUM14].

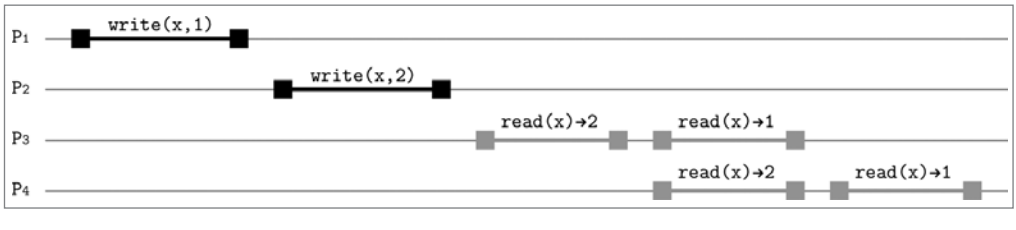

Рис. 11.5. Порядок при последовательной согласованности

Чтение устаревших значений может объясняться, например, расхождением реплик: даже если результаты записи распространяются на разные реплики в одном и том же порядке, они могут поступить туда в разное время.

Основное отличие от линеаризуемости заключается в отсутствии глобально установленных временных рамок. При линеаризуемости результаты операции должны проявляться в пределах ее временных рамок. К моменту завершения операции записи W_1 ее результаты должны быть применены и каждая операция чтения должна увидеть либо значение, записанное операцией W_1, либо более позднее. Аналогичным образом, после того как операция чтения R_1 возвращает считанное значение, любая операция чтения, которая выполняется после нее, должна возвращать значение, которое увидела операция R_1, или более позднее значение (в отношении которого действует то же правило).

Последовательная согласованность ослабляет это требование: результаты операции могут становиться видимыми после ее завершения, если порядок является согласованным с точки зрения отдельных процессоров. Операции записи одного происхождения не могут «перепрыгивать» друг через друга: должен сохраняться их программный порядок относительно их собственного исполняемого процесса. Другое ограничение заключается в том, что порядок проявления операций должен быть согласованным для всех операций чтения.

Подобно линеаризуемости, современные процессоры не гарантируют последовательную согласованность по умолчанию, и поскольку процессор может переупорядочивать инструкции, мы должны использовать барьеры памяти (или, иначе говоря, «заграждения»), чтобы гарантировать, что записи будут становиться видимыми для конкурентно работающих потоков в корректном порядке [DREPPER07] [GEORGOPOULOS16].

Причинная согласованность

> Видите ли, есть только одна постоянная, одна универсальная величина, и это единственная настоящая истина: причинность. Действие. Реакция. Причина и следствие.
>
> *Меровинген, «Матрица: перезагрузка»*

Хотя в наличии глобального порядка операций часто нет необходимости, иногда требуется установить порядок в рамках *нескольких* операций. Согласно модели *причинной согласованности* (causal consistency), все процессы должны видеть *причинно-связанные* операции в одном и том же порядке. *Конкурентные операции записи* без причинной связи могут наблюдаться разными процессорами в разном порядке.

Сначала давайте посмотрим, зачем нам нужна причинная связь и как могут распространяться результаты операций записи без причинной связи. На рис. 11.6 процессы P_1 и P_2 выполняют записи, которые не являются причинно-упорядоченными. Результаты этих операций могут распространяться до операций чтения в разное

время и без сохранения порядка. Процесс P_3 увидит значение 1 раньше значения 2, а процесс P_4 сначала увидит значение 2, а затем — значение 1.

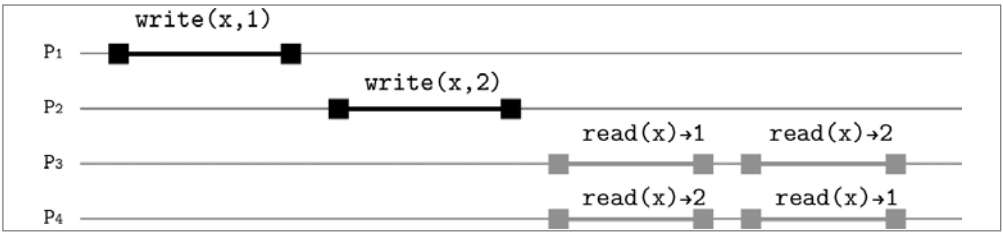

Рис. 11.6. Операции записи без причинной связи

На рис. 11.7 показан пример причинно-связанных записей. В дополнение к записанному значению теперь мы должны указать показание логических часов, которое будет определять причинный порядок в рамках нескольких операций. Процесс P_1 начинается с операции записи `write(x, ∅, 1) → t1`, которая начинает выполнение с начального значения ∅. Процесс P_2 выполняет еще одну операцию записи `write(x, t1, 2)` и указывает, что она логически упорядочена после момента t_1, требуя, чтобы результаты операции распространялись только в порядке, установленном логическими часами.

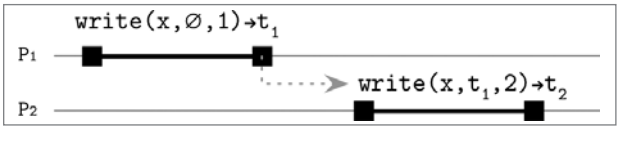

Рис. 11.7. Причинно-связанные операции записи

Таким образом устанавливается *причинный порядок* в рамках данных операций. Даже если результат последней операции записи распространяется быстрее, чем результат предыдущей, он не будет виден до тех пор, пока не будут получены все ее зависимости и восстановлен порядок событий согласно их логическим временным меткам. Другими словами, логически устанавливается отношение «произошло раньше» (happens-before relationship) без использования физических часов, и все процессы достигают согласия в отношении этого порядка.

На рис. 11.8 показаны процессы P_1 и P_2, выполняющие причинно-связанные операции записи, которые распространяются до процессов P_3 и P_4 в их логическом порядке. Такой подход избавляет нас от ситуации, показанной на рис. 11.6; вы можете сравнить истории процессов P_3 и P_4 на обоих рисунках.

Этот процесс во многом напоминает процесс общения на интернет-форуме: вы публикуете что-то в интернете; другой пользователь видит ваше сообщение и отвечает на него; третий пользователь видит этот ответ и продолжает цепочку беседы.

Потоки сообщений могут расходиться: вы можете ответить на одну из цепочек в потоке и продолжить цепочку событий, но некоторые потоки будут иметь только несколько общих сообщений, поэтому мы не всегда будем иметь единую историю для всех сообщений.

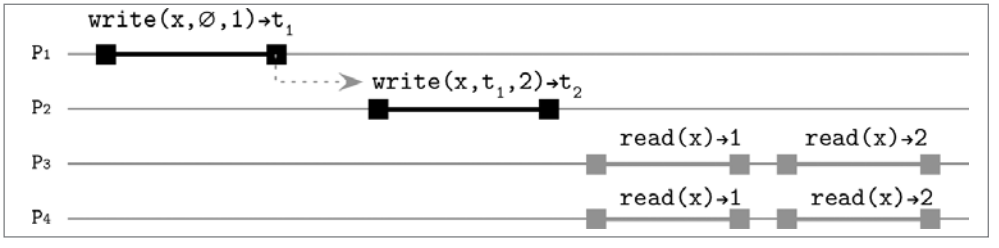

Рис. 11.8. Причинно-связанные операции записи

В причинно-согласованной системе для приложения гарантируется согласованность в рамках сеанса: обеспечивается согласованность представления базы данных с ее собственными действиями, даже если она выполняет запросы на чтение и запись для разных, потенциально не согласованных серверов [TERRY94]. В частности, обеспечиваются гарантии в отношении монотонного чтения, монотонной записи, чтения собственных записей и записи после чтения. Более подробную информацию о моделях сеансов можно найти в разделе «Модели сеансов» на с. 251.

Причинную согласованность можно реализовать с помощью логических часов [LAMPORT78] и отправки с каждым сообщением метаданных контекста, показывающих, какие операции логически предшествуют текущей. При этом получаемое с сервера обновление должно содержать самую последнюю версию контекста. Любая операция может быть обработана, только если результаты всех предшествующих ей операций уже применены. Сообщения, для которых контексты не совпадают, буферизируются на сервере, так как их еще слишком рано доставлять.

Двумя известными и часто упоминаемыми проектами, реализующими причинную согласованность, являются системы COPS (Clusters of Order-Preserving Servers — кластеры сохраняющих порядок серверов) [LLOYD11] и Eiger [LLOYD13]. Оба проекта реализуют причинную связь посредством библиотеки (реализованной как фронтальный сервер, к которому подключаются пользователи) и обеспечивают согласованность путем отслеживания зависимостей. COPS отслеживает зависимости посредством версий ключей, в то время как Eiger вместо этого устанавливает порядок операций (операции в Eiger могут зависеть от операций, выполняемых на других узлах; например, в случае распределенных многосоставных транзакций). В отличие от хранилищ, согласованных в конечном счете, обе системы не показывают неупорядоченные операции. Вместо этого они обнаруживают и обрабатывают конфликты: в COPS это делается путем проверки порядка ключей и использования функций, специфичных для приложения, а в Eiger всегда отдается предпочтение последней операции записи.

Векторные часы

Установление причинной связи позволяет системе восстанавливать последовательность событий, даже если сообщения доставляются без сохранения порядка, заполнять пробелы между сообщениями и избегать публикации результатов операции в случае, если некоторые сообщения все еще отсутствуют. Например, если сообщения `{M1(∅, t1), M2(M1, t2), M3(M2, t3)}`, каждое из которых указывает свои зависимости, причинно связаны и распространяются без сохранения порядка, процесс буферизует их на время до тех пор, пока не сможет собрать все зависимости операции и восстановить их причинный порядок [KINGSBURY18b]. Во многих базах данных, например в Dynamo [DECANDIA07] и Riak [SHEEHY10a], для установления причинного порядка используются *векторные часы* (vector clock) [LAMPORT78] [MATTERN88].

Векторные часы — это структура для установления *частичного порядка* (partial order) между событиями, обнаружения и устранения расхождений между цепочками событий. С помощью векторных часов мы можем моделировать общее время, глобальное состояние и представлять асинхронные события как синхронные. Процессы поддерживают векторы *логических часов* (logical clock). У каждого процесса есть свои логические часы. Каждые часы начинают отсчет с начального значения и увеличивают свои показания на единицу при поступлении каждого нового события (такого, как операция записи). При получении векторов часов от других процессов процесс обновляет свой локальный вектор до самых высоких показаний часов для каждого процесса, присутствующих в получаемых векторах (т. е. максимальных значений, которые когда-либо видел передающий узел).

Чтобы использовать векторные часы для разрешения конфликтов, всякий раз, когда мы делаем запись в базу данных, мы сначала проверяем, существует ли локально значение для записываемого ключа. Если предыдущее значение уже существует, мы добавляем новую версию в вектор версий и устанавливаем причинную связь между двумя операциями записи. В противном случае мы запускаем новую цепочку событий и инициализируем значение одной версией.

Мы говорили о согласованности с точки зрения доступа к регистрам общей памяти и упорядочения операций согласно системным часам и впервые упомянули потенциальное расхождение реплик, когда говорили о последовательной непротиворечивости. Поскольку необходимо упорядочить только операции записи в одну и ту же ячейку памяти, мы не можем оказаться в ситуации, когда у нас будет конфликт записи, если значения независимы [LAMPORT79].

Поскольку мы ищем модель согласованности, которая позволила бы улучшить доступность и производительность, мы должны допустить расхождение реплик не только из-за обслуживания устаревших операций чтения, но и из-за принятия потенциально конфликтующих записей, в силу чего в системе допускается создание двух независимых цепочек событий. На рис. 11.9 показано такое расхождение: с точки зрения одной реплики история выглядит как `1`, `5`, `7`, `8`, а с точки зрения другой реплики — как `1`, `5`, `3`. Система Riak позволяет пользователям видеть расхождение историй и разрешать такие конфликты [DAILY13].

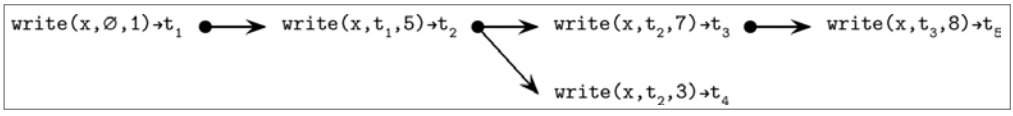

Рис. 11.9. Расхождение историй выполнения при причинной согласованности

Чтобы реализовать причинную согласованность, мы должны хранить историю причинных связей, добавить сборку мусора и попросить пользователя синхронизовать расходящиеся истории в случае конфликта. Векторные часы позволяют выявить конфликт, но не предлагают конкретного способа его разрешения, поскольку семантика разрешения часто зависит от приложения. Из-за этого некоторые согласованные в конечном счете базы данных, например Apache Cassandra, не поддерживают причинный порядок операций и вместо этого для разрешения конфликтов отдают приоритет последней операции записи [ELLIS13].

Модели сеансов

Рассмотрение согласованности с точки зрения распространения значений полезно для разработчиков баз данных в силу того, что это помогает понять и задать необходимые инварианты данных, но некоторые вещи легче понять и объяснить с точки зрения клиента. Мы можем взглянуть на нашу распределенную систему с точки зрения одного клиента, а не нескольких клиентов.

Модели сеансов (session models) [VIOTTI16] (или, как их еще называют, модели согласованности, ориентированные на клиента) [TANENBAUM06]) помогают рассматривать согласованность распределенной системы с точки зрения клиента: каким образом каждый клиент видит состояние системы при выполнении операций чтения и записи.

Если другие модели согласованности, которые мы обсуждали до сих пор, фокусируются на объяснении порядка операций при наличии конкурентных клиентов, рассмотрение согласованности с точки зрения клиента помогает увидеть, как взаимодействует с системой отдельный клиент. Мы по-прежнему подразумеваем, что операции каждого клиента являются последовательными: он должен завершить одну операцию перед тем, как начать выполнение следующей. Если клиент аварийно завершается или теряет соединение с сервером до завершения операции, мы не делаем никаких предположений о состоянии незавершенных операций.

В распределенной системе клиенты зачастую могут подключаться к любой доступной реплике, и если результаты недавней записи в одну реплику не распространяются на другую реплику, клиент может не увидеть изменение состояния, произведенное этой репликой.

При этом можно естественным образом рассчитывать на то, что клиенту будет видна каждая отправленная им операция записи. Это предположение выполняется для модели согласованности с гарантией *чтения собственных записей* (read-own-writes), согласно которой каждая операция чтения, следующая за операцией записи в той же или другой

реплике, должна видеть обновленное значение. Например, операция чтения `read(x)`, которая была выполнена сразу после операции записи `write(x, V)`, вернет значение `V`.

Модель с гарантией *монотонного чтения* (monotonic reads) ограничивает видимость значения и предполагает, что если операция чтения `read(x)` видела значение `V`, то следующие операции чтения должны видеть это же значение `V` или более позднее значение.

Модель с гарантией *монотонной записи* (monotonic writes) предполагает, что значения, исходящие от одного и того же клиента, становятся видимыми в порядке их внесения клиентом. Если в соответствии с порядком сеанса клиента операция записи `write(x, V2)` была сделана после операции записи `write(x, V1)`, их результаты должны становиться видимыми для всех других процессов в том же порядке (т. е. сначала `V1`, а затем `V2`). Без этого предположения старые данные могут быть «воскрешены», что приведет к потере данных.

Принцип «запись после чтения» (writes follow reads) (который иногда называют «причинной связью сеанса» (session causality)) гарантирует, что операции записи будут расположены после операций записи, которые были видны предыдущим операциям чтения. Например, если операция записи `write(x, V2)` расположена после операции чтения `read(x)`, которая вернула значение `V1`, то операция `write(x, V2)` будет располагаться после операции `write(x, V1)`.

В моделях сеансов не делается предположений об операциях, выполняемых *различными* процессами (клиентами) или из другого логического сеанса [TANENBAUM14]. Эти модели описывают порядок операций с точки зрения одного процесса. Однако одни и те же гарантии должны предоставляться для *каждого* процесса в системе. Другими словами, если процесс P_1 может читать свои собственные записи, то процесс P_2 также должен иметь возможность читать свои собственные записи.

Гарантии монотонного чтения, монотонной записи и чтения собственных записей в совокупности обеспечивают конвейерную согласованность ОЗУ (pipelined RAM — PRAM) [LIPTON88] [BRZEZINSKI03], также известную как FIFO-согласованность. Конвейерная согласованность ОЗУ гарантирует, что операции записи, исходящие от одного процесса, будут распространяться в том порядке, в котором они были выполнены этим процессом. В отличие от последовательной согласованности, здесь операции записи, исходящие от разных процессов, могут быть видны в разном порядке.

Свойства, описанные моделями согласованности, ориентированными на клиента, являются желательными и в большинстве случаев используются разработчиками распределенных систем для проверки систем и упрощения их использования.

Согласованность в конечном счете

Синхронизация затратна как в многопроцессорном программировании, так и в распределенных системах. Как обсуждалось в разделе «Модели согласованности» на с. 239, мы можем ослабить гарантии согласованности и использовать модели, которые

допускают некоторое расхождение между узлами. Например, последовательная согласованность позволяет распространять результаты чтения с разными скоростями.

При *согласованности в конечном счете* (eventual consistency) обновления распространяются по системе асинхронно. Формально этот тип согласованности подразумевает, что если для элемента данных не выполняются *дополнительные* обновления, то *в конечном счете* все операции доступа должны возвращать последнее записанное значение [VOGELS09]. В случае конфликта представление о том, что является «последним значением», может варьировать, так как значения расходящихся реплик синхронизируются с использованием такой стратегии разрешения конфликтов, как принцип приоритетности последней операции записи, или с использованием векторных часов (см. подраздел «Векторные часы» на с. 250).

Применительно к распространению значений определение «в конечном счете» интересно тем, что не задает жестких временных рамок для процесса распространения. Если служба доставки предоставит лишь гарантию того, что доставка будет произведена «в конечном счете», то вы вряд ли посчитаете ее очень надежной. Однако на практике это хорошо работает, и многие базы данных в настоящее время определяются как *согласованные в конечном счете*.

Настраиваемая согласованность

Системы, согласованные в конечном счете, иногда описываются в терминах теоремы CAP: вы можете пожертвовать доступностью ради согласованности, и наоборот (см. раздел «Печально известная теорема CAP» на с. 233). С точки зрения сервера системы, согласованные в конечном счете, обычно реализуют настраиваемую согласованность, при которой данные реплицируются, считываются и записываются с использованием трех переменных:

Коэффициент репликации N

Количество узлов, которые будут хранить копию данных.

Согласованность записи W

Количество узлов, которые должны подтвердить операцию записи для ее успешного завершения.

Согласованность чтения R

Количество узлов, которые должны ответить на операцию чтения для ее успешного завершения.

При уровнях согласованности, удовлетворяющих неравенству `R + W > N`, система может гарантировать возврат самого последнего записанного значения, потому что между наборами чтения и записи всегда есть перекрытие. Например, если `N = 3`, `W = 2`, а `R = 2`, система может допустить отказ только одного узла. Два узла из трех должны подтвердить запись. В идеальном случае система также асинхронно реплицирует операцию записи на третий узел. Если третий узел не работает, механизмы антиэнтропии (см. главу 12) в конечном итоге распространят операцию записи.

Во время чтения должны быть доступны две реплики из трех, чтобы мы могли отвечать на запросы с согласованными результатами. Любая комбинация узлов даст нам по крайней мере один узел, содержащий самую последнюю запись для заданного ключа.

При выполнении операции записи координатор должен передать ее `N` узлам, но может дождаться подтверждения только от `W` узлов перед тем, как продолжить ее выполнение (или `W – 1`, если координатор также является репликой). Остальные операции записи могут выполняться асинхронно или завершиться с ошибкой. Аналогичным образом при выполнении чтения координатор должен собрать как минимум `R` ответов. Некоторые базы данных используют спекулятивное выполнение и отправляют дополнительные запросы на чтение, чтобы уменьшить задержку ответа координатора. Это означает, что если один из первоначально отправленных запросов на чтение завершается с ошибкой или медленно распространяется, то вместо `R` ответов можно засчитать спекулятивные запросы.

Для системы с интенсивной записью можно иногда выбирать `W = 1` и `R = N`, что позволяет подтверждать успешность записи только одним узлом, но для этого потребуется, чтобы все реплики (даже потенциально отказавшие) были доступны для чтения. То же самое верно и для комбинации `W = N`, `R = 1`: последнее значение может быть прочитано с любого узла, если запись успешно применена ко всем репликам.

Повышение уровней согласованности операций чтения или записи увеличивает задержки и повышает требования к доступности узлов во время запросов. Снижение уровней улучшает доступность системы при ослаблении согласованности.

КВОРУМЫ

Уровень согласованности, состоящий из `⌊N/2⌋ + 1` узлов, называется кворумом (quorum), большинством узлов. В случае распада сети или отказа узлов в системе с `2f + 1` узлами активные узлы могут продолжить прием операций записи или чтения, если недоступно до `f` узлов, пока остальная часть кластера снова не станет доступной. Другими словами, такие системы могут выдержать отказ не более `f` узлов.

При выполнении операций чтения и записи с использованием кворумов система не сможет выдержать отказ большинства узлов. Например, если есть всего три реплики и две из них не работают, то для операций чтения и записи не хватит узлов, чтобы достичь согласованности, поскольку только один узел из трех сможет ответить на запрос.

Чтение и запись с использованием кворумов не гарантируют монотонность в случаях неполной записи. Если какая-либо операция записи завершилась с ошибкой после записи значения в одной реплике из трех, то в зависимости от того, с какими репликами была установлена связь, чтение с применением кворума может вернуть либо результат незавершенной операции, либо старое значение. Поскольку последовательным операциям чтения одних и тех же значений не нужно связываться с одними и теми же репликами, возвращаемые ими значения могут чередоваться. Для обеспечения монотонного чтения (за счет снижения доступности) мы должны использовать блокирующий механизм исправления при чтении (см. раздел «Исправление при чтении» на с. 262).

Реплики-свидетели

Использование кворумов для достижения согласованности чтения помогает повысить доступность: даже если некоторые узлы не работают, СУБД может принимать запросы на чтение и выполнять операции записи. Требование относительно большинства гарантирует, что, поскольку в любом большинстве есть перекрытие хотя бы одного узла, при любом чтении кворумом будет видна самая последняя завершенная запись кворума. Однако использование репликации и большинства повышает затраты на хранение: мы должны хранить копию данных в каждой реплике. Если наш коэффициент репликации равен пяти, мы должны хранить пять копий.

Мы можем оптимизировать затраты на хранение, используя концепцию *«реплик-свидетелей»* (witness replicas). Вместо того чтобы хранить копию записи в каждой реплике, мы можем разделить реплики на подмножества *«копий»* (copy, «c» далее по тексту) и *«свидетелей»* (witness, «w» далее по тексту). Реплики-копии будут попрежнему содержать записи данных. При нормальной работе реплики-свидетели будут просто хранить записи, свидетельствующие о том, что операция записи была выполнена. Однако может возникнуть ситуация, когда число реплик-копий будет слишком малым. Например, если при наличии трех реплик-копий и двух реплик-свидетелей перестанут работать две реплики-копии, мы получим кворум из одной копии и двух реплик-свидетелей.

В случае превышения времени ожидания операции записи или отказа реплик-копий можно *повышать статус* реплик-свидетелей, чтобы они временно хранили запись вместо тех реплик-копий, которые дали сбой или превысили время ожидания. После восстановления работоспособности исходных реплик-копий можно вернуть в прежнее состояние те реплики, которые были повышены в статусе, или сделать свидетелями восстановленные реплики.

Давайте рассмотрим реплицируемую систему с тремя узлами, два из которых содержат копии данных, а третий выступает в роли свидетеля: `[1c, 2c, 3w]`. Мы пытаемся произвести запись, но узел `2c` временно недоступен и не может завершить операцию. В этом случае мы временно сохраняем запись в реплике-свидетеле `3w`. После того как узел `2c` снова заработает, механизмы исправления могут привести его в актуальное состояние и удалить лишние копии из свидетелей.

Давайте представим другой сценарий: мы пытаемся выполнить чтение и запись присутствует в узлах `1c` и `3w`, но не в узле `2c`. Поскольку любых двух реплик достаточно, чтобы составить кворум, то мы можем гарантировать согласованные результаты, если имеется любое подмножество из двух узлов, будь то две копии — `[1c, 2c]` или одна копия копии и один свидетель — `[1c, 3w]` или `[2c, 3w]`. Если мы читаем из узлов `[1c, 2c]`, то мы выбираем самую последнюю запись из узла `1c` и можем реплицировать ее в узел `2c`, поскольку там значение отсутствует. В случае, когда доступны только узлы `[2c, 3w]`, самую последнюю запись можно извлечь из узла `3w`. Для восстановления исходной конфигурации и приведения в актуальное состояние узла `2c` запись может быть реплицирована в этот узел и удалена из свидетеля.

В общем случае наличие `n` реплик-копий и `m` реплик-свидетелей обеспечивает те же гарантии доступности, что и наличие `n + m` копий, если соблюдаются следующие два правила:

- Операции чтения и записи выполняются с использованием большинства (т. е. `N/2 + 1` участников).
- По крайней мере одна из реплик в этом кворуме *обязательно* является копией.

Это работает, поскольку данные гарантированно содержатся либо в репликах-копиях, либо в репликах-свидетелях. В случае отказа реплики-копии приводятся в актуальное состояние механизмом исправления с временным сохранением данных в репликах-свидетелях.

Использование реплик-свидетелей помогает снизить затраты на хранение при сохранении инвариантов согласованности. Есть несколько реализаций этого подхода, например Spanner [CORBETT12] и Apache Cassandra.

Строгая согласованность в конечном счете и структуры CRDT

Мы обсудили несколько моделей строгой согласованности, таких как линеаризуемость и сериализуемость, а также форму слабой согласованности: согласованность в конечном счете. Возможной золотой серединой, предлагающей некоторые преимущества обеих моделей, является *строгая согласованность в конечном счете*. Согласно этой модели, обновления могут распространяться на серверы с опозданием или без сохранения порядка, но после того, как все обновления в конечном итоге распространятся на целевые узлы, можно будет разрешить конфликты между ними и объединить их с получением единого действительного состояния [GOMES17].

При соблюдении некоторых условий мы можем ослабить наши требования к согласованности, позволяя операциям сохранять дополнительное состояние, позволяющее состояниям с расхождениями синхронизоваться (другими словами, объединяться) после выполнения. Одним из наиболее ярких примеров такого подхода являются *бесконфликтные реплицируемые типы данных* (Conflict-Free Replicated Data Types, CRDT [SHAPIRO11a]), реализованные, например, в СУБД Redis [BIYIKOGLU13].

CRDT — это специализированные структуры данных, которые исключают конфликт и позволяют применять операции над этими типами данных в любом порядке без изменения результата. Такое свойство может быть чрезвычайно полезным в распределенной системе. Например, в многоузловой системе, в которой используются бесконфликтные реплицируемые счетчики, мы можем увеличивать значения счетчиков на каждом узле независимо, даже если они не могут связываться друг с другом из-за распада сети. После восстановления связи результаты всех узлов можно синхронизовать, и ни одна из операций, примененных во время распада, не будет потеряна.

Такая особенность делает структуры CRDT полезными в системах, согласованных в конечном счете, поскольку состояния реплик в таких системах могут временно расходиться. Реплики могут выполнять операции локально, без предварительной синхронизации с другими узлами, а операции в конечном итоге распространяются на все другие реплики, потенциально без соблюдения порядка. Структуры CRDT позволяют нам реконструировать полное состояние системы из локальных отдельных состояний или последовательностей операций.

Простейшим примером структур CRDT являются коммутативные реплицируемые типы данных на основе операций (CmRDT). Для применения структур CmRDT нужно, чтобы разрешенные операции обладали следующими свойствами:

Свободные от побочных эффектов

Их применение не меняет состояния системы.

Коммутативные

Порядок аргументов не имеет значения: `x • y = y • x`. Другими словами, не имеет значения, объединяется ли `x` с `y` или `y` с `x`.

Причинно-упорядоченные

Их успешная доставка зависит от предварительного условия, которое гарантирует, что система достигла состояния, к которому может быть применена операция.

Например, мы могли бы реализовать *только растущий счетчик* (grow-only counter). Каждый сервер может хранить вектор состояний, содержащий последние известные обновления счетчика от всех других участников, инициализированный нулями. Каждому серверу разрешено изменять только свое значение в векторе. При распространении обновлений функция слияния `merge(state1, state2)` объединяет состояния двух серверов.

Например, у нас есть три сервера с инициализированными векторами начального состояния:

```
Узел 1:      Узел 2:       Узел 3:
[0, 0, 0]    [0, 0, 0]     [0, 0, 0]
```

Если мы обновим счетчики на первом и третьем узлах, их состояния изменятся следующим образом:

```
Узел 1:      Узел 2:       Узел 3:
[1, 0, 0]    [0, 0, 0]     [0, 0, 1]
```

После распространения обновлений мы используем функцию слияния для объединения результатов путем выбора максимального значения для каждого слота:

```
Узел 1 (распространяется вектор состояния узла 3):
merge([1, 0, 0], [0, 0, 1]) = [1, 0, 1]

Узел 2 (распространяется вектор состояния узла 1):
merge([0, 0, 0], [1, 0, 0]) = [1, 0, 0]
```

```
Узел 2 (распространяется вектор состояния узла 3):
merge([1, 0, 0], [0, 0, 1]) = [1, 0, 1]

Узел 3 (распространяется вектор состояния узла 1):
merge([0, 0, 1], [1, 0, 0]) = [1, 0, 1]
```

Чтобы определить текущее векторное состояние, вычисляется сумма значений во всех слотах: `sum([1, 0, 1]) = 2`. Функция слияния коммутативна. Поскольку серверам разрешено обновлять только свои собственные значения, а эти значения являются независимыми, дополнительной координации не требуется.

Можно создать *положительно-отрицательный счетчик* (PN-счетчик), поддерживающий как увеличение, так и уменьшение путем использования полезных данных, состоящих из двух векторов: вектора `P`, в узлах которого хранятся положительные приращения, и вектора `N`, в котором хранятся отрицательные приращения. В крупных системах, чтобы избежать распространения огромных векторов, мы можем использовать *суперузлы*. Суперузлы реплицируют состояния счетчиков и помогают избежать постоянного обмена данными между одноранговыми узлами [SHAPIRO11b].

Для сохранения и репликации значений мы можем использовать *регистры*. Простейшей версией регистра является регистр с принципом «побеждает последняя запись» (last-write-wins, LWW), в котором хранится уникальная глобально упорядоченная временная метка, прикрепляемая к каждому значению для разрешения конфликтов. В случае конфликта записей мы сохраняем только ту, которая имеет большую временную метку. Операция слияния (выбор значения с наибольшей отметкой времени) здесь также является коммутативной, поскольку использует временную метку. Если мы не можем допустить, чтобы значения отбрасывались, мы можем предоставить логику слияния, специфичную для конкретного приложения, и использовать *многозначный регистр*, который хранит все записанные значения и позволяет приложению выбрать из них нужное значение.

Другим примером структуры CRDT является неупорядоченное *только растущее множество* (G-Set). Каждый узел поддерживает свое локальное состояние и может добавлять к нему элементы. Добавление элементов создает действительный набор. Объединение двух множеств также является коммутативной операцией. Подобно счетчикам, мы можем использовать два множества для поддержки как добавления, так и удаления. В таком случае мы должны сохранять следующий инвариант: только значения, содержащиеся в множестве добавления, могут быть добавлены в множество удаления. Чтобы реконструировать текущее состояние множества, все элементы, содержащиеся в множестве удаления, вычитаются из множества добавления [SHAPIRO11b].

Примером бесконфликтного типа, комбинирующего более сложные структуры, является бесконфликтный реплицируемый тип данных для JSON, позволяющий производить такие модификации, как вставка, удаление и присвоение, в глубоко вложенных документах JSON с типами преобразования и списковыми типами. Этот

алгоритм выполняет операции слияния на стороне клиента и не требует распространения операций в каком-либо определенном порядке [KLEPPMANN14].

Структуры CRDT предоставляют нам достаточно обширные возможности, поэтому существует множество хранилищ данных, использующих эту концепцию для обеспечения строгой согласованности в конечном счете (Strong Eventual Consistency, SEC). Это мощная концепция, которую можно добавить в наш арсенал инструментов для построения отказоустойчивых распределенных систем.

Итоги

Отказоустойчивые системы используют репликацию для повышения доступности: даже если некоторые процессы не работают или не отвечают, система в целом может продолжать корректно работать. Однако синхронизация нескольких копий требует дополнительной координации.

Мы обсудили несколько моделей согласованности с одной операцией, начиная с моделей с самыми строгими гарантиями и заканчивая моделями с наименьшими гарантиями[1]:

Линеаризуемость

С точки зрения внешнего наблюдателя, операции применяются мгновенно; поддерживается порядок операций реального времени.

Последовательная согласованность

Результаты операций распространяются с соблюдением некоторого общего порядка, и этот порядок согласован с порядком, в котором они выполнялись отдельными процессами.

Причинная согласованность

Результаты причинно-связанных операций становятся видимыми в одном и том же порядке для всех процессов.

Конвейерная согласованность ОЗУ (PRAM/FIFO)

Результаты операций становятся видимыми в том же порядке, в котором они выполнялись отдельными процессами. Операции записи из разных процессов могут наблюдаться в разном порядке.

Далее мы обсудили несколько моделей сеансов:

Чтение собственных записей

Операции чтения отражают предшествующие операции записи. Операции записи распространяются по системе и становятся доступными для последующего чтения тем же клиентом.

[1] Эти короткие определения даны только для повторения материала, читателю рекомендуется обратиться к полным определениям с контекстом.

Монотонное чтение

Любая операция чтения, которая увидела некое значение, не может увидеть значение, появившееся раньше увиденного.

Монотонная запись

Результаты операций записи, поступающих от одного клиента, распространяются на других клиентов в том порядке, в котором они были выполнены этим клиентом.

Запись после чтения

Операции записи ставятся после операций записи, результаты которых были видимыми при предыдущих операциях чтения, выполненных тем же клиентом.

Знание и понимание этих концепций могут помочь вам в понимании гарантий, обеспечиваемых нижележащими системами, и использовании их для разработки приложений. Модели согласованности описывают правила, которым должны следовать операции над данными, но их область действия ограничена конкретной системой. Размещение систем с более слабыми гарантиями поверх систем с более сильными гарантиями или игнорирование влияния уровня согласованности нижележащих систем может привести к неисправимому рассогласованию и потере данных.

Мы также обсудили концепции *согласованности в конечном счете* и *настраиваемой согласованности*. Системы на основе кворума используют большинство для выдачи согласованных данных. Для снижения затрат на хранение могут использоваться *реплики-свидетели*.

Дополнительная литература

Если вы хотите узнать больше о концепциях, упомянутых в этой главе, вы можете обратиться к следующим источникам:

Модели согласованности

Perrin, Matthieu. 2017. Distributed Systems: Concurrency and Consistency (1st Ed.). Elsevier, UK: ISTE Press.

Viotti, Paolo and Marko Vukolić. 2016. “Consistency in Non-Transactional Distributed Storage Systems.” ACM Computing Surveys 49, no. 1 (July): Article 19. https://doi.org/0.1145/2926965.

Bailis, Peter, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. “Highly available transactions: virtues and limitations.” Proceedings of the VLDB Endowment 7, no. 3 (November): 181–192. https://doi.org/ 10.14778 / 2732232.2732237.

Aguilera, M. K., and D. B. Terry. 2016. “The Many Faces of Consistency.” Bulletin of the Technical Committee on Data Engineering 39, no. 1 (March): 3–13.

ГЛАВА 12

Антиэнтропия и распространение

Большинство моделей связи, которые мы до сих пор обсуждали, представляли собой либо модели одноранговой типа, либо модели типа один ко многим (с координатором и репликами). Для надежного распространения записей данных по всей системе нам нужно, чтобы распространяющий узел был доступен и мог связаться с другими узлами, но даже в таком случае пропускная способность будет ограничена одной машиной.

Быстрое и надежное распространение часто в меньшей степени применимо к записям данных, будучи более актуальным для таких общекластерных метаданных, как информация о членстве (добавлении и удалении узлов), состоянии узлов, отказах, изменениях схемы и т. д. Сообщения с такой информацией обычно являются небольшими и распространяются нечасто, однако их необходимо распространять максимально быстро и надежно.

Такие обновления обычно могут распространяться на все узлы кластера с использованием одной из трех широких групп подходов [DEMERS87], схематическое изображение которых показано на рис. 12.1:

а) Рассылка уведомлений от одного процесса всем остальным.

б) Периодический одноранговый обмен информацией. Одноранговые узлы соединяются попарно и обмениваются сообщениями.

в) Совместная рассылка, когда получатели сообщений пересылают их дальше и помогают быстрее и надежнее распространять информацию.

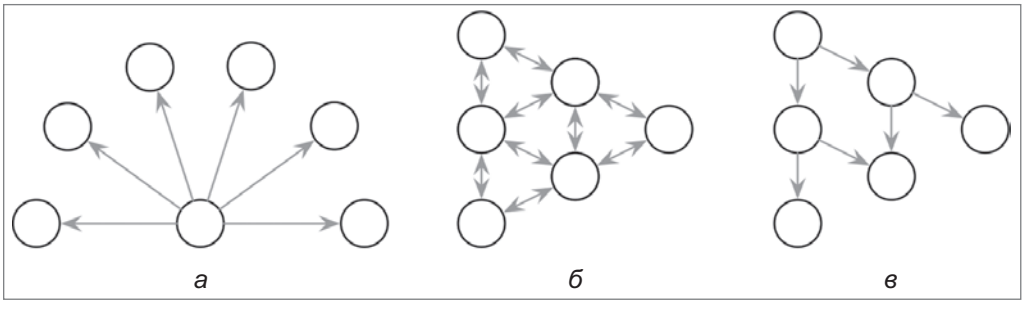

Рис. 12.1. Рассылка (*а*), антиэнтропия (*б*) и сплетни (*в*)

Рассылка сообщения всем остальным процессам — наиболее простой подход, который хорошо работает, когда число узлов в кластере невелико, но в крупных кластерах это может быть слишком *затратно* из-за большого количества узлов и *ненадежно* из-за чрезмерной зависимости от одного процесса. Отдельные процессы не всегда знают о существовании всех других процессов в сети. Кроме того, необходимо некоторое пересечение во времени, в течение которого одновременно работают и рассылающий процесс, и каждый из его получателей, что в некоторых случаях бывает трудно обеспечить.

Чтобы ослабить эти ограничения, мы можем допустить, что *некоторые* обновления не смогут достичь адресата. Координатор приложит все усилия и доставит сообщения всем доступным участникам, а затем механизмы антиэнтропии восстановят синхронизацию узлов в случае каких-либо отказов. Таким образом, ответственность за доставку сообщений распределяется между всеми узлами системы и делится на два этапа: первичная доставка и периодическая синхронизация.

Энтропия — это свойство, отражающее степень неупорядоченности системы. В распределенной системе энтропия отражает степень расхождения состояний между узлами. Поскольку энтропия является нежелательным свойством, которое необходимо свести к минимуму, есть множество методов ее уменьшения.

Антиэнтропия обычно используется для приведения узлов в актуальное состояние в случае отказа основного механизма доставки. Система может продолжать корректно работать, даже если в какой-то момент произойдет сбой координатора, так как другие узлы будут продолжать распространять информацию. Другими словами, механизм антиэнтропии призван уменьшить время конвергенции в системах, согласованных в конечном счете.

Чтобы синхронизировать узлы, механизм антиэнтропии запускает фоновый процесс или процесс переднего плана, который сравнивает и согласовывает недостающие или конфликтующие записи. Фоновые процессы антиэнтропии используют такие вспомогательные структуры, как деревья Меркла, и обновляют журналы для выявления расхождений. Процессы антиэнтропии переднего плана совмещаются с запросами на чтение или запись путем передачи подсказки, исправления при чтении и т. д.

Если в реплицируемой системе расходятся реплики, то для восстановления согласованности и синхронизации реплик мы должны найти и исправить отсутствующие записи, попарно сравнив состояния реплик. В случае больших наборов данных это может быть очень затратно: мы должны прочитать весь набор данных на обоих узлах и уведомить реплики о последних изменениях состояния, которые еще не были распространены. Чтобы снизить затраты, мы можем рассмотреть варианты потери репликами актуальности и схемы доступа к данным.

Исправление при чтении

Проще всего обнаружить расхождение между репликами во время чтения, поскольку в этот момент мы можем связаться с репликами, получить запрашиваемое состояние

у каждой из них и посмотреть, совпадают ли их ответы. Обратите внимание, что в этом случае мы не запрашиваем весь набор данных, хранящийся в каждой реплике, и ограничиваемся только данными, запрошенными клиентом.

Координатор выполняет распределенное чтение, оптимистично предполагая, что реплики синхронизированы и содержат одинаковую информацию. Если реплики выдают разные ответы, координатор отправляет недостающие обновления в те реплики, в которых они отсутствуют.

Этот механизм называется *исправлением при чтении* (read repair). Он часто используется для обнаружения и устранения несоответствий. В ходе исправления при чтении узел-координатор отправляет запрос репликам, ожидает ответы и сравнивает их. Если некоторые реплики пропустили последние обновления и их ответы отличаются, координатор выявляет несоответствия и отправляет этим репликам обновления [DECANDIA07].

Некоторые СУБД, наподобие Dynamo, не требуют, чтобы обеспечивалась связь со *всеми* репликами, используя вместо этого настраиваемые уровни согласованности. Чтобы вернуть согласованные результаты, не нужно связываться со всеми репликами и исправлять их; достаточно будет того количества узлов, которое соответствует уровню согласованности. Если мы выполняем чтение и запись с использованием кворума, мы все равно получаем согласованные результаты, при том что некоторые из реплик могут содержать неполный набор записей.

Исправление при чтении можно реализовать либо как *блокирующую*, либо как *асинхронную* операцию. В ходе блокирующего исправления при чтении исходный клиентский запрос должен ждать, пока координатор не «исправит» реплики. При асинхронном исправлении при чтении просто планируется задача, которая может быть выполнена после возврата результатов пользователю.

Блокирующее исправление обеспечивает монотонность (https://databass.dev/links/1, см. раздел «Модели сеансов» на с. 251) для чтения с применением кворума: после считывания клиентом определенного значения последующие операции чтения возвращают то же значение, которое он видел, или более позднее, поскольку состояния реплики были исправлены. Если мы не используем кворумы для чтения, мы теряем гарантию монотонности, поскольку данные могут не распространиться на целевой узел к моменту последующего чтения. В то же время блокирующее исправление при чтении снижает доступность, поскольку целевые реплики должны подтвердить исправления, а результат чтения не возвращается, пока реплики не ответят.

Чтобы определить, какие именно записи различаются в ответах реплик, некоторые базы данных (например, Apache Cassandra) используют специализированные итераторы с обработчиками слияния, которые находят различия между объединенным результатом и отдельными входными данными. Их выходные данные затем используются координатором для уведомления реплик об отсутствующих данных.

При исправлении при чтении предполагается, что реплики *в основном* синхронизированы и мы не ожидаем, что каждый запрос приведет к блокирующему исправлению.

Из-за монотонности чтения блокирующих исправлений мы также можем ожидать, что последовательные запросы будут возвращать одинаковые согласованные результаты, если в промежутке между ними не будет выполнено ни одной операции записи.

Чтение с запросом хэш-суммы

Вместо отправки полного запроса на чтение каждому узлу координатор может отправить полный запрос на чтение только одному узлу, а остальным репликам — запросы *хэш-суммы*. Запрос хэш-суммы считывает локальные данные реплики и вместо возврата полного моментального снимка запрошенных данных вычисляет их хэш. После этого координатор может вычислить хэш-сумму единожды прочитанного полного снимка и сравнить его с хэш-суммами всех других узлов. Если все хэш-суммы совпадают, то можно быть уверенными, что реплики синхронизированы.

Если хэш-суммы не совпадают, то в этом случае координатору не известно, какие реплики ушли вперед, а какие отстали. Чтобы синхронизировать отставшие реплики с остальными узлами, координатор должен выполнить полное чтение всех реплик, вернувших несовпадающие хэш-суммы, сравнить их ответы, согласовать данные и отправить обновления отставшим репликам.

Хэш-суммы обычно вычисляются с использованием некриптографической хэш-функции, такой как MD5, поскольку это вычисление должно выполняться быстро, чтобы обеспечить эффективность «удачного пути». При этом возможны *коллизии хэш-функций*, но в большинстве реальных систем вероятность их возникновения крайне низка. Поскольку базы данных часто используют более одного механизма антиэнтропии, можно ожидать, что даже в маловероятном случае коллизии хэш-функций данные будут синхронизированы другой подсистемой.

Передача подсказки

Другим механизмом антиэнтропии является *передача подсказки* (Hinted Handoff) [DECANDIA07], представляющая собой механизм исправления при записи. Если целевой узел не подтвердил запись, координатор записи или одна из реплик сохраняет у себя специальную запись, называемую *подсказкой*, которая воспроизводится на целевом узле после восстановления его работоспособности.

В СУБД Apache Cassandra, если не задан уровень согласованности `ANY` [ELLIS11], то операции записи, выполняемые по подсказке, не учитываются при расчете коэффициента репликации (см. раздел «Настраиваемая согласованность» на с. 253), поскольку данные в журнале подсказок недоступны для операций чтения и используются только для синхронизации отставших участников.

Некоторые базы данных, например Riak, используют механизм передачи подсказки в сочетании с *нестрогими кворумами*. При использовании нестрогих кворумов в случае отказов реплик операции записи могут использовать дополнительные исправные узлы

из списка узлов; причем эти узлы необязательно должны быть целевыми репликами для выполняемых операций.

Например, допустим, что у нас есть кластер из пяти узлов {A, B, C, D, E} и узлы {A, B, C} являются целевыми репликами для выполняемой операции записи, а узел B не работает. Узел A, являясь координатором запроса, выбирает узел D для организации нестрогого кворума и поддержания требуемых гарантий доступности и долговечности. При этом данные реплицируются в узлы {A, D, C}. Однако в метаданных записи, вносимой в узел D, будет содержаться пометка о том, что это подсказка, поскольку эта операция записи изначально предназначалась для узла B. После восстановления работоспособности узла B узел D попытается переслать ему подсказку. После того как подсказка будет воспроизведена на узле B, ее можно будет безопасно удалить без сокращения общего количества реплик [DECANDIA07].

Если при такой же конфигурации из-за распада сети узлы {B, C} будут ненадолго отделены от остальной части кластера и будет выполнена операция записи в узлы {A, D, E} с использованием нестрогого кворума, то операция чтения из узлов {B, C}, выполненная сразу после этой операции записи, не увидит самые последние данные [DOWNEY12]. Другими словами, нестрогие кворумы повышают доступность за счет уменьшения степени согласованности.

Деревья Меркла

Поскольку механизм исправления при чтении может исправить несоответствия только в данных, запрашиваемых в текущий момент, мы должны использовать различные механизмы для поиска и исправления рассогласования в данных, которые не запрашиваются активно.

Как мы уже обсуждали, для выяснения того, какие строки расходятся в репликах, они должны попарно обмениваться записями данных и проводить их сравнение. Это очень непрактично и затратно. Во многих базах данных для снижения затрат на синхронизацию используют *деревья Меркла* (Merkle trees) [MERKLE87].

Деревья Меркла организуют компактное хэшированное представление локальных данных в виде дерева хэш-сумм. Самый низкий уровень этого хэш-дерева создается путем сканирования всей таблицы, содержащей записи данных, и вычисления хэш-сумм для диапазонов записей. Более высокие уровни дерева содержат хэш-суммы, вычисленные на основе хэш-сумм нижележащего уровня, создавая иерархическое представление, позволяющее быстро выявлять несоответствия путем сравнения хэш-сумм, рекурсивно следуя узлам хэш-дерева, чтобы сузить искомые несогласованные диапазоны. Проверка может осуществляться путем пересылки и сопоставления информации в рамках поддеревьев каждого уровня или в рамках всего дерева.

На рис. 12.2 показана структура дерева Меркла. На самом нижнем уровне находятся хэш-суммы диапазонов записей данных. Хэш-суммы каждого более высокого уровня вычисляются путем хэширования хэш-сумм нижележащих уровней с рекурсивным повтором этого процесса до корня дерева.

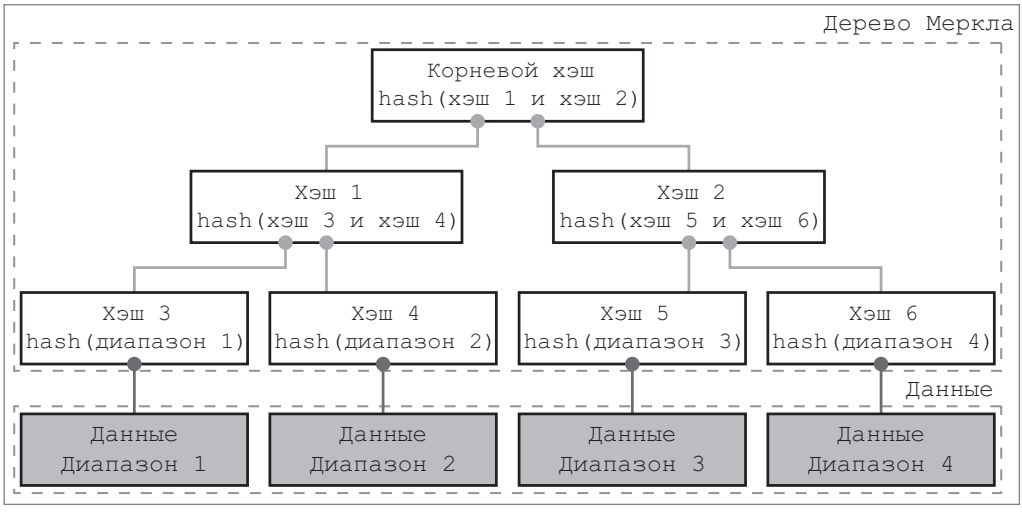

Рис. 12.2. Дерево Меркла. Серые прямоугольники представляют диапазоны записей данных. Белые прямоугольники представляют иерархию хэш-дерева

Чтобы определить, имеется ли несоответствие между двумя репликами, нам достаточно сравнить хэш-суммы корневого уровня из их деревьев Меркла. Путем попарного сравнения хэш-сумм сверху вниз можно найти диапазоны, содержащие различия между узлами, и исправить содержащиеся в них записи данных.

Поскольку деревья Меркла вычисляются рекурсивно снизу вверх, изменение данных вызывает пересчет всего поддерева. Кроме того, существует компромисс между размером дерева (от которого также зависят размеры пересылаемых сообщений) и его точностью (тем, насколько малы и точны диапазоны данных).

Битовая карта векторов версий

В последних исследованиях по этой теме вводится понятие *битовой карты векторов версий* (bitmap version vector) [GONÇALVES15], которую можно использовать для разрешения конфликтов данных на основе новизны: каждый узел ведет одноранговый журнал операций, которые выполнялись локально или были реплицированы. Механизм антиэнтропии сравнивает журналы, и отсутствующие данные реплицируются на целевой узел.

Каждая координируемая узлом операция записи представляется в виде *точки* `(i, n)`: событие с локальным для узла порядковым номером `i`, координируемое узлом `n`. Порядковый номер `i` начинается с `1` и увеличивается каждый раз, когда узел выполняет операцию записи.

Для отслеживания состояний реплик мы используем локальные для узла логические часы. Каждые часы — набор точек, представляющих операции записи, которые данный узел видел *напрямую* (координируемые самим узлом) или *транзитивно* (координируемые и реплицируемые с других узлов).

Согласно логическим часам узла между событиями, координируемыми самим узлом, не будет пробелов. Пробелы будут в том случае, если некоторые операции записи не будут реплицированы с других узлов. Для синхронизации два узла могут обменяться показаниями логических часов, выявить пробелы, представленные отсутствующими точками, а затем реплицировать соответствующие записи данных. Для этого нам нужно воссоздать записи данных, на которые ссылается каждая точка. Эта информация хранится в *контейнере точек и причинных связей*, который сопоставляет точки с информацией о причинных связях для заданного ключа. Таким образом, разрешение конфликтов сводится к определению причинных связей между операциями записи.

На рис. 12.3 (адаптированном из источника [GONÇALVES15]) показан пример представления состояния трех узлов системы, P_1, P_2 и P_3 с точки зрения узла P_2, отслеживающего, какие значения он видел. Каждый раз, когда узел P_2 выполняет операцию записи или получает реплицированное значение, он обновляет эту таблицу.

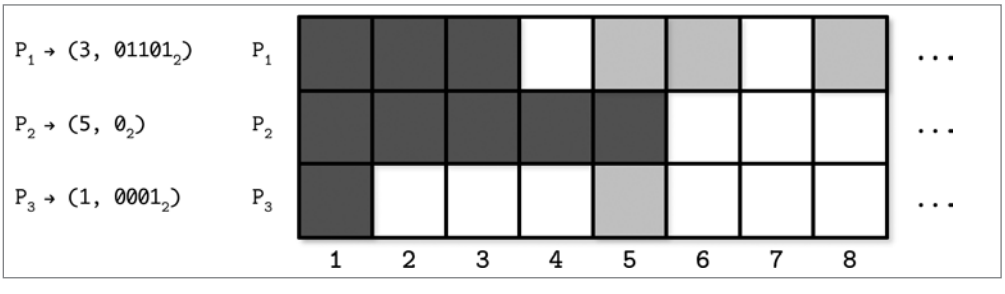

Рис. 12.3. Пример битовой карты векторов версий

Во время репликации узел P_2 создает компактное представление этого состояния и создает карту, начиная с идентификатора узла и заканчивая парой последних значений, до которых он видел последовательные операции записи, и битовую карту, где другие видимые операции записи кодируются как 1. $(3, 01101_2)$ здесь означает, что узел P_2 видел последовательные обновления вплоть до третьего значения и значения во второй, третьей и пятой позициях относительно 3 (т. е. он видел значения с порядковыми номерами 5, 6 и 8).

Во время обмена с другими узлами он получит отсутствующие обновления, которые видел другой узел. После того как все узлы системы увидят последовательные значения вплоть до индекса `i`, вектор версии можно будет усечь до этого индекса.

Преимущество этого подхода заключается в том, что он определяет причинную связь между операциями записи значений и позволяет узлам точно идентифицировать элементы данных, отсутствующие в других узлах. Возможным недостатком является то, что когда узел не работает в течение длительного времени, другие узлы не могут усекать журнал, поскольку им еще нужно реплицировать данные на отстающий узел после восстановления его работоспособности.

Распространение сплетен

Массы всегда являются рассадником психических эпидемий.

Карл Юнг

Чтобы задействовать другие узлы и распространять обновления с охватом рассылки и надежностью антиэнтропии, мы можем использовать протоколы сплетен.

Протоколы сплетен (gossip protocols) — это вероятностные процедуры связи, в основе которых лежит механизм распространения слухов в человеческом обществе или болезней среди населения. Слухи и эпидемии наглядно показывают, как работают эти протоколы: слухи распространяются до тех пор, пока люди готовы их слушать, а болезни — до тех пор, пока среди населения еще имеются предрасположенные к заболеванию люди.

Основной целью протоколов сплетен является совместное распространение информации от одного процесса на остальную часть кластера. Подобно тому как вирус распространяется среди людей, передаваясь от одного человека к другому, потенциально увеличивая охват с каждым шагом, информация передается через систему, охватывая все больше процессов.

Процесс, хранящий записи, которые нужно распространить, называют *зараженным*. Любой процесс, который еще не получил обновление, соответственно является *восприимчивым*. Зараженные процессы, не желающие распространять новое состояние по прошествии периода активного распространения, считаются *удаленными* [DEMERS87]. Все процессы изначально являются восприимчивыми. Всякий раз, когда приходит обновление для некоторой записи данных, получивший это обновление процесс становится зараженным и начинает распространять его среди других *случайно выбранных* соседних процессов, заражая их. После того как у зараженных процессов появляется уверенность в том, что обновление распространено, они переходят в состояние удаленных.

Чтобы избежать явной координации и ведения глобального списка получателей и необходимости распространения одним координатором сообщения всем участникам в системе, этот класс алгоритмов моделирует завершаемость, используя функцию *ослабления интереса*. При этом эффективность протокола определяется тем, насколько быстро он может заразить максимально возможное количество узлов с минимально возможными затратами на избыточные сообщения.

Механизм сплетен можно использовать для асинхронной доставки сообщений в однородных децентрализованных системах, где у узлов может не быть долгосрочного членства и какой-либо топологии организации. Поскольку протоколы сплетен обычно не требуют явной координации, они могут быть полезны в системах с гибким членством (где узлы часто присоединяются к группе и покидают ее) и в ячеистых сетях.

Протоколы сплетен очень устойчивы и помогают достичь высокой надежности при наличии отказов, присущих распределенным системам. Поскольку сообщения ретранслируются случайным образом, они могут быть доставлены, даже если некоторые ком-

поненты связи между узлами выходят из строя, — в таком случае сообщения достигнут адресатов другими путями. Можно сказать, что система адаптируется к отказам.

Механизмы сплетен

Процессы периодически выбирают другие процессы случайным образом (где `f` — настраиваемый параметр, называемый *степенью ветвления* (fa out)) и обмениваются с ними «горячей» информацией. Всякий раз, когда процесс получает новую информацию от других узлов, он пытается передать ее дальше. Поскольку узлы выбираются вероятностным образом, всегда будет в какой-то мере присутствовать перекрытие, повторная доставка сообщений или распространение сообщений по кругу. *Избыточность сообщений* (message redundancy) — это показатель, который отражает издержки, связанные с повторной доставкой. Избыточность — это важное свойство, имеющее решающее значение для работы механизма сплетен.

Время, необходимое системе для достижения конвергенции, называется *задержкой*. Существует небольшая разница между достижением конвергенции (остановкой процесса сплетен) и доставкой сообщения всем узлам, поскольку может быть короткий период, в течение которого все узлы уже получили информацию, но сплетни продолжают доставляться. Степень ветвления и задержка зависят от размера системы: в более крупной системе мы должны либо увеличить степень ветвления, чтобы сохранить стабильность, либо разрешить более высокую задержку.

Со временем, когда узлы заметят, что они снова и снова получают одну и ту же информацию, сообщение начнет терять свою значимость и узлам в конечном итоге нужно будет прекратить его передачу. Ослабление интереса можно рассчитать либо вероятностным образом (путем расчета вероятности остановки распространения для каждого процесса на каждом шаге), либо с использованием порогового значения (путем подсчета количества полученных дубликатов и остановки распространения после того, как это число станет слишком большим). Оба подхода должны принимать во внимание размер кластера и степень ветвления. Подсчет дубликатов для оценки конвергенции может привести к снижению задержки и степени избыточности [DEMERS87].

С точки зрения согласованности протоколы сплетен предлагают *конвергентную согласованность* (convergent consistency) [BIRMAN07]: вероятность того, что узлы будут иметь одинаковое представление о событиях тем выше, чем дальше эти события удалены в прошлое.

Наложенные сети

Несмотря на то что протоколы сплетен важны и полезны, они обычно применяются для узкого круга задач. С помощью неэпидемических подходов можно распространять послание с невероятностной достоверностью, меньшей избыточностью и, как правило, более оптимальным образом [BIRMAN07]. Алгоритмы сплетен часто хвалят за их масштабируемость и за возможность распространять сообщение в пределах `logN` раундов сообщений (где `N` — размер кластера) [KREMARREC07], но также важно

помнить и о количестве избыточных сообщений, генерируемых во время работы механизма сплетен. Для достижения надежности протоколы на основе сплетен производят некоторое количество дублирующих сообщений.

Случайный выбор узлов значительно повышает *надежность* системы: при распаде сети сообщения будут доставлены в конечном счете, если есть каналы, косвенно связывающие два процесса. Очевидным недостатком этого подхода является то, что он не оптимален с точки зрения отправки сообщений: чтобы гарантировать надежность, мы должны поддерживать избыточные соединения между узлами и отправлять избыточные сообщения.

Компромиссный вариант между двумя подходами заключается в создании *временной* фиксированной топологии в системе сплетен. Для этого можно создать *наложенную сеть* (overlay network) одноранговых узлов: узлы могут тестировать другие узлы и выбирать лучшие точки контакта на основе близости (обычно оцениваемой по величине задержке).

Узлы могут образовывать *остовные деревья* (spanning trees): неориентированные графы без петель с различными ребрами, охватывающие всю сеть. Имея такой граф, можно распределять сообщения за фиксированное число шагов.

На рис. 12.4 показан пример остовного дерева[1]:

а) Мы обеспечиваем связь между всеми точками без использования всех ребер.

б) Мы можем потерять связь со всем поддеревом, если будет разорван только один канал.

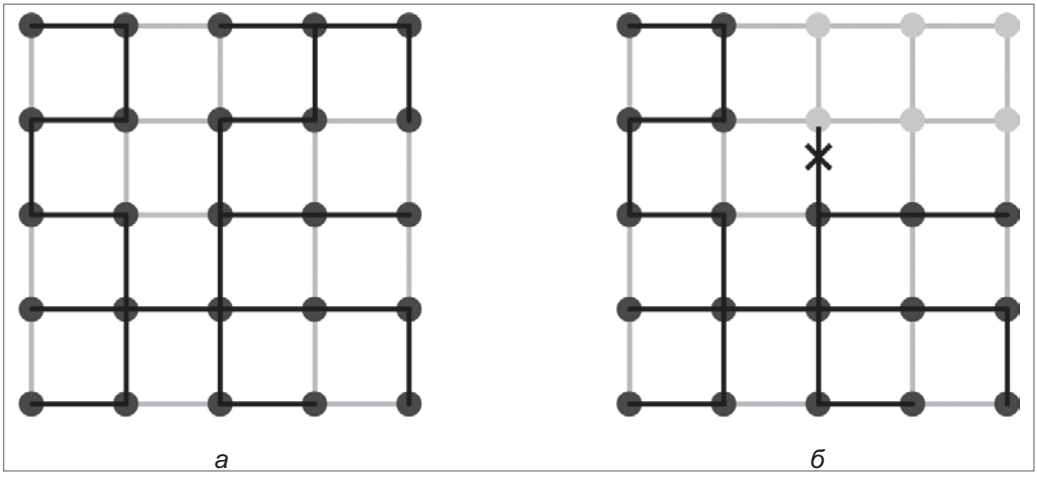

Рис. 12.4. Остовное дерево. Черные точки представляют узлы. Черные линии представляют наложенную сеть. Серые линии представляют другие возможные соединения между узлами

[1] Этот пример используется только для иллюстрации: узлы в сети обычно не располагаются в виде решетки.

Одним из потенциальных недостатков этого подхода является то, что он может привести к образованию взаимосвязанных «островков» узлов, имеющих строгие предпочтения по отношению друг к другу.

Чтобы обеспечить низкое количество сообщений с возможностью быстрого восстановления в случае потери связи, мы можем объединить оба подхода, используя фиксированные топологии и рассылку с помощью дерева, когда система находится в *стабильном* состоянии, и прибегая к механизму сплетен для преодоления отказа и восстановления системы.

Гибридные протоколы сплетен

Многоадресные активные/лениво-активные деревья (push/lazy-push multicast trees) (Plumtrees) [LEITAO07] представляют компромисс между эпидемическими примитивами и примитивами рассылок, основанными на деревьях. Деревья Plumtrees создают наложенную сеть узлов остовного дерева для *активного* распространения сообщений с наименьшими издержками. В нормальных условиях узлы отправляют полные сообщения только небольшому подмножеству узлов, предоставляемых службой тестирования узлов.

Каждый узел отправляет полное сообщение небольшому подмножеству узлов, а для остальных узлов *лениво* пересылает только идентификатор сообщения. Если узел получает идентификатор сообщения, которое он никогда не видел, он может запросить его у других узлов. Использование таких лениво-активных шагов гарантирует высокую надежность и позволяет быстро восстанавливать дерево рассылки. В случае отказов протокол возвращается к механизму сплетен посредством лениво-активных шагов, рассылая сообщения и восстанавливая наложенную сеть.

Для распределенных систем характерно то, что любой узел или связь между узлами может в любой момент выйти из строя, что делает невозможным обход дерева в том случае, когда сегмент становится недоступным. Сеть ленивых сплетен позволяет уведомлять узлы об увиденных сообщениях с целью построения и восстановления дерева.

На рис. 12.5 показана такая двойная связь: узлы связаны оптимальным остовным деревом (сплошные линии) и сетью ленивых сплетен (пунктирные линии). Эта иллюстрация не представляет какую-либо конкретную топологию сети, а лишь показывает *соединения* между узлами.

Частичные представления

Рассылка сообщений всем известным узлам и поддержание полного представления кластера могут стать затратными и непрактичными, особенно при большом показателе *текучести* (churn) (который показывает, насколько часто узлы присоединяются к системе и покидают ее). Чтобы избежать этого, протоколы сплетен часто используют *службу отбора узлов* (peer sampling service). Эта служба поддерживает

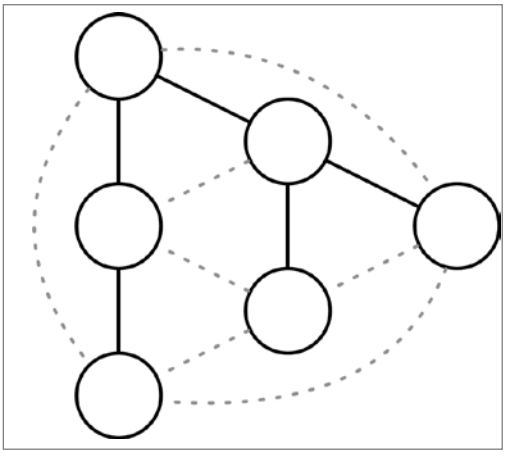

Рис. 12.5. Ленивые и активные сети. Сплошные линии представляют дерево рассылки. Пунктирные линии обозначают связи для ленивых сплетен

частичное представление (partial view) кластера, которое периодически обновляется с помощью сплетен. Частичные представления перекрываются, поскольку для протоколов сплетен требуется некоторая степень избыточности; однако слишком большая степень избыточности будет означать выполнение лишней работы.

Например, протокол гибридного частичного преставления (HyParView) [LEITAO07] поддерживает небольшое *активное* и более крупное *пассивное* представления кластера. Узлы из активного представления образуют наложенную сеть, которую можно использовать для распространения. Пассивное представление используется для ведения списка узлов, которые можно использовать для замены сбойных узлов активного представления.

Периодически узлы выполняют операцию перетасовки, во время которой они обмениваются между собой активными и пассивными представлениями. Во время этого обмена узлы добавляют участников из пассивных и активных представлений других узлов в свои пассивные представления, исключая при этом самые старые значения, чтобы ограничить размер списка.

Активное представление обновляется в зависимости от состояния узлов в этом представлении и в зависимости от запросов от других узлов. Если процесс P_1 предполагает, что процесс P_2 (содержащийся в его активном представлении) дал сбой, то процесс P_1 удаляет процесс P_2 из своего активного представления и пытается установить соединение с замещающим процессом P_3 из пассивного представления. Если соединение установить не удается, то процесс P_3 удаляется из пассивного представления процесса P_1.

В зависимости от количества процессов в активном представлении процесса P_1 процесс P_3 может отклонить соединение, если активное представление процесса P_3 уже заполнено. Если представление процесса P_1 пусто, то процесс P_3 должен заменить

в своем активном представлении один из узлов узлом P_1. Это ускоряет первый запуск или восстановление узлов, позволяя им быстрее стать эффективными участниками кластера, и платой за это является то, что некоторые соединения могут зацикливаться.

Этот подход помогает уменьшить количество сообщений в системе за счет использования для распространения только узлов активного представления, сохраняя при этом высокую надежность за счет использования пассивных представлений в качестве механизма восстановления. Одним из показателей производительности и качества является то, насколько быстро служба отбора узлов достигает стабильной наложенной сети в случаях реорганизации топологии [JELASITY04]. В этом отношении протокол HyParView получает довольно высокие оценки за счет используемого способа поддержки представлений и за счет того, что он отдает приоритет впервые запускаемым процессам.

Протокол HyParView и дерево Plumtree используют метод *гибридных сплетен*, сводящийся к использованию небольшого подмножества узлов для рассылки сообщений с переходом к более широкой сети в случае отказов или распада сети. Обе системы не используют глобальное представление, включающее все узлы, что может быть полезным не только в силу большого количества узлов в системе (что случается достаточно редко), но и с точки зрения затрат на поддержание актуального списка участников в каждом узле. Частичные представления позволяют узлам активно взаимодействовать только с небольшим подмножеством соседних узлов.

Итоги

Системы, согласованные в конечном счете, допускают расхождение состояний реплик. Настраиваемая согласованность позволяет повышать степень согласованности за счет уменьшения степени доступности, и наоборот. Расхождение реплик можно устранить с помощью одного из механизмов антиэнтропии:

Передача подсказки

Временно сохраняет записи в соседних узлах в случае, если целевой узел выходит из строя, и воспроизводит их на целевом узле после восстановления его работоспособности.

Исправление при чтении

Сверяет запрашиваемые при чтении диапазоны данных путем сравнения ответов, выявления отсутствующих записей и отправки их отстающим репликам.

Деревья Меркла

Определяют, какие диапазоны данных требуют восстановления, путем построения иерархических деревьев хэш-сумм и обмена ими.

Битовая карта векторов версий

Выявляет в репликах недостающие операции записи посредством ведения компактных записей с информацией о самых последних операциях записи.

Эти подходы антиэнтропии могут настраиваться для обеспечения оптимального охвата, новизны или завершенности. Мы можем уменьшить область действия антиэнтропии, синхронизируя только активно запрашиваемые данные (исправление при чтении) или только отдельные недостающие операции записи (передача подсказки). Предположив, что большинство отказов являются временными и восстановление участников происходит максимально быстро, мы можем хранить журнал самых последних отклонившихся событий и точно знать благодаря этому, что именно необходимо синхронизировать в случае отказа (битовая карта векторов версий). Если нужно попарно сравнивать целые наборы данных на нескольких узлах и эффективно находить различия между ними, мы можем хэшировать данные и сравнивать хэш-суммы (деревья Меркла).

Для надежного распространения информации в крупномасштабной системе можно использовать протоколы сплетен. Протоколы гибридных сплетен сокращают количество пересылаемых сообщений с сохранением максимально возможной устойчивости к распадам сети.

Многие современные системы используют механизм сплетен для обнаружения отказов и распространения информации о членстве [DECANDIA07]. Протокол HyParView используется в высокопроизводительной и масштабируемой среде распределенных вычислений Partisan. Деревья Plumtree использовались в ядре хранилища Riak для общекластерной информации.

Дополнительная литература

Если вы хотите узнать больше о концепциях, упомянутых в этой главе, вы можете обратиться к следующим источникам:

Протоколы сплетен

Shah, Devavrat. 2009. “Gossip Algorithms.” Foundations and Trends in Networking 3, no. 1 (January): 1–125. https://doi.org/10.1561/1300000014.

Jelasity, Márk. 2003. “Gossip-based Protocols for Large-scale Distributed Systems.” Dissertation. http://www.inf.u-szeged.hu/~jelasity/dr/doktori-mu.pdf.

Demers, Alan, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. “Epidemic algorithms for replicated database maintenance.” In Proceedings of the sixth annual ACM Symposium on Principles of distributed computing (PODC ’87), 1-12. New York: Association for Computing Machinery. https://doi.org/10.1145/41840.41841.

ГЛАВА 13

Распределенные транзакции

Для поддержания порядка в распределенной системе мы должны гарантировать хотя бы некоторую согласованность. В разделе «Модели согласованности» на с. 239 мы говорили о моделях согласованности с одним объектом и одной операцией, которые помогают нам рассматривать ход отдельных операций. Однако в базах данных нам часто нужно атомарно выполнять *несколько* операций.

Атомарные операции описываются в терминах переходов между состояниями: база данных находилась в состоянии `A` до того, как была запущена конкретная транзакция; к тому времени, когда она закончилась, состояние изменилось с `A` на `B`. Это легко понять в терминах операций, поскольку у транзакций нет заранее определенного присоединенного состояния. Вместо этого они применяют операции к записям данных, начиная с *некоторого* момента времени. Это дает нам некоторую гибкость в плане планирования и выполнения: транзакции можно переупорядочивать и даже выполнять повторно.

Основное внимание при обработке транзакций уделяется определению допустимых *историй выполнения*, моделированию и представлению возможных сценариев перемежающегося выполнения. В данном случае история представляет собой граф зависимостей, показывающий, какие транзакции выполнялись до выполнения текущей транзакции. История считается *сериализуемой*, если она эквивалентна (т. е. имеет такой же граф зависимостей) некоторой истории, которая выполняет эти транзакции последовательно. Об историях выполнения, их эквивалентности, сериализуемости и других концепциях мы говорили в подразделе «Сериализуемость» на с. 112. Данная глава в основном повторяет применительно к распределенным системам материал главы 5, где мы обсуждали локальную обработку транзакций на уровне узла.

Односекционные транзакции (single partition transactions) подразумевают использование пессимистичных (с блокировкой или отслеживанием) или оптимистичных (с выполнением проверки после попытки выполнения) схем управления конкурентностью, которые мы обсуждали в главе 5, но ни первый, ни второй подход не решает проблему многосекционных транзакций, которые требуют координации между различными серверами, распределенной фиксацией и протоколами отката.

По большому счету, когда мы переводим деньги с одного счета на другой, желательно, чтобы снятие денег с первого счета и их зачисление на второй счет производились *одновременно*. Однако если транзакция разбивается на отдельные этапы, то даже снятие или зачисление не будет выглядеть как атомарная операция: нам нужно прочитать прежний баланс, добавить или вычесть необходимую сумму и сохранить

этот результат. Каждый из этих подэтапов включает в себя несколько операций: узел получает запрос, анализирует его, находит данные на диске, выполняет операцию записи и, наконец, подтверждает ее. Причем даже такое описание является достаточно высокоуровневым представлением: чтобы выполнить простую операцию записи, мы должны выполнить сотни более мелких шагов.

Это означает, что мы должны сначала выполнить транзакцию и только затем сделать ее результаты *видимыми*. Но давайте сначала определим, что представляет собой транзакция. *Транзакция* — это набор операций, атомарная единица выполнения. Атомарность транзакции подразумевает, что видимыми становятся либо все ее результаты, либо ни один из них. Например, если транзакция изменяет сразу несколько строк или даже таблиц, то будут применены либо все, либо ни одна из этих модификаций.

Для обеспечения атомарности транзакции должны быть *восстановимыми*. То есть в случае невозможности завершения, прерывания или истечения времени ожидания транзакции должен производиться полный откат ее результатов. Частично выполненная невосстановимая транзакция может оставить базу данных в несогласованном состоянии. Таким образом, в случае неудачного выполнения транзакции базу данных необходимо вернуть в ее прежнее состояние, как если бы мы даже не пытались ее выполнять.

Другим важным аспектом являются распад сети и отказы узлов: хотя узлы системы могут давать сбой и восстанавливать свою работоспособность независимо, их состояния должны при этом оставаться согласованными. Это означает, что требование по обеспечению атомарности должно распространяться не только на локальные операции, но и на операции, выполняемые на других узлах: изменения должны быть либо распространены на все участвующие в транзакции узлы, либо не распространены ни на один из них [LAMPSON79].

Обеспечение атомарности операций

Чтобы несколько операций выглядели атомарными, особенно если некоторые из них являются удаленными, нам нужно использовать алгоритмы *атомарной фиксации* (atomic commitment). Атомарная фиксация не допускает разногласий между участниками: транзакция не будет зафиксирована, если хотя бы один из участников проголосует против нее. В то же время это означает, что давшие сбой процессы должны прийти к тому же выводу, что и остальная часть когорты. Другим важным следствием является то, что алгоритмы атомарной фиксации не работают при наличии византийских ошибок, когда процесс преднамеренно выдает ложную информацию о своем состоянии или выбирает произвольное значение, поскольку это нарушает единогласие [HADZILACOS05].

Атомарная фиксация призвана обеспечить согласие в отношении того, следует ли выполнять предложенную транзакцию. Когорты не могут выбирать предложенную транзакцию, влиять на нее, модифицировать ее или предлагать вместо нее какую-

либо альтернативу: они могут лишь отдать свой голос за или против ее выполнения [ROBINSON08].

Алгоритмы атомарной фиксации не предъявляют строгих требований к семантике операций *подготовки* (prepare), *фиксации* (commit) или *отката* (rollback) транзакции. Разработчикам базы данных необходимо решить следующее:

- В какой момент данные считаются готовыми к фиксации, когда остается лишь переключить указатель, чтобы обнародовать изменения.
- Как следует выполнять фиксацию, чтобы результаты транзакции становились видимыми максимально быстро.
- Как следует производить откат внесенных транзакцией изменений в том случае, если алгоритм решит не фиксировать ее.

В главе 5 мы уже рассмотрели локальные по отношению к узлу реализации этих процессов.

Алгоритмы атомарной фиксации используют многие распределенные системы, например MySQL (для выполнения распределенных транзакций) и Kafka (для обеспечения взаимодействия производителя и потребителя [МЕНТА17]).

В базах данных распределенные транзакции выполняются компонентом, широко известным как *диспетчер транзакций* (transaction manager). Диспетчер транзакций — это подсистема, отвечающая за планирование, координацию, выполнение и отслеживание транзакций. В распределенной среде диспетчер транзакций отвечает за то, чтобы гарантии видимости на локальном узле согласовывались с видимостью, обеспечиваемой распределенными атомарными операциями. Другими словами, транзакции фиксируются во всех секциях и для всех реплик.

Мы обсудим два алгоритма атомарной фиксации: алгоритм двухфазной фиксации, который решает задачу фиксации, но не допускает выхода из строя координатора, и алгоритм трехфазной фиксации [SKEEN83], который решает задачу *неблокирующей атомарной фиксации*[1] и позволяет участникам продолжать выполнение даже в случае отказов координатора [BABAOGLU93].

Двухфазная фиксация

Давайте начнем с самого простого протокола распределенной фиксации, который допускает многосекционные *атомарные* обновления. (Для получения дополнительной информации о секционировании вы можете обратиться к разделу «Секционирование базы данных» на с. 289.) *Двухфазная фиксация* (2-phase commit, 2PC) обычно обсуждается в контексте транзакций базы данных. Алгоритм 2PC выполняется в два этапа. На первом этапе распределяется значение, по которому принимается решение, и со-

[1] Мелким шрифтом добавляется: «предполагая высоконадежную сеть». Другими словами, сеть, которая исключает распады [ALHOUMAILY10]. Последствия этого предположения обсуждаются в разделе указанной статьи, посвященном описанию алгоритма.

бираются голоса. В ходе второй фазы узлы просто «поворачивают переключатель», делая видимыми результаты первой фазы.

Алгоритм 2PC предполагает наличие *лидера* (или *координатора*), который хранит состояние, собирает голоса и является основным ориентиром для раунда достижения согласия. Остальные узлы называются *когортами* (cohorts). В данном случае когорты обычно представляют собой секции, оперирующие непересекающимися наборами данных, над которыми выполняются транзакции. Координатор и каждая когорта ведут локальные журналы операций для каждого выполненного шага. Участники голосуют за принятие или отклонение некоторого *значения*, предложенного координатором. Чаще всего этим значением является идентификатор той распределенной транзакции, которую необходимо выполнить, но алгоритм 2PC может применяться и в других контекстах.

Координатором может быть узел, получивший запрос на выполнение транзакции, выбранный случайным образом с помощью алгоритма выбора лидера, или назначенный вручную, или даже установленный на весь срок службы системы. Данный протокол не накладывает ограничений на роль координатора, и эта роль может быть передана другому участнику для обеспечения надежности или производительности.

Как следует из названия, двухфазная фиксация выполняется в два этапа:

Подготовка (prepare)

Координатор уведомляет когорты о новой транзакции, отправляя сообщение *Предложение*. Когорты принимают решение о том, могут ли они зафиксировать относящуюся к ним часть транзакции. Если когорта решает, что она может выполнить фиксацию, она уведомляет об этом решении координатора. В противном случае она отправляет координатору запрос на прерывание транзакции. Все принимаемые когортой решения персистентным образом сохраняются в журнале координатора, а каждая когорта хранит копию своего решения локально.

Фиксация/прерывание (commit/abort)

Операции внутри транзакции могут изменять состояние разных секций (каждая из которых представлена когортой). Если хотя бы одна из когорт голосует за прерывание транзакции, координатор отправляет всем когортам сообщение *Прерывание*. Только если все когорты проголосовали положительно, координатор отправляет им финальное сообщение *Фиксация*.

Этот процесс показан на рис. 13.1.

На этапе *подготовки* координатор распространяет предлагаемое значение и собирает голоса участников в отношении того, следует ли фиксировать это значение. Когорты могут отклонить предложение координатора, если, например, другая конфликтующая транзакция уже зафиксировала другое значение.

После сбора голосов координатор может перейти к принятию решения о том, следует ли *зафиксировать* или *прервать* транзакцию. Если все когорты проголосовали положительно, он принимает решение о фиксации и уведомляет об этом когорты,

отправляя сообщение *Фиксация*. В противном случае координатор отправляет всем когортам сообщение *Прерывание* и производится откат транзакции. То есть если один узел отклоняет предложение, весь раунд прерывается.

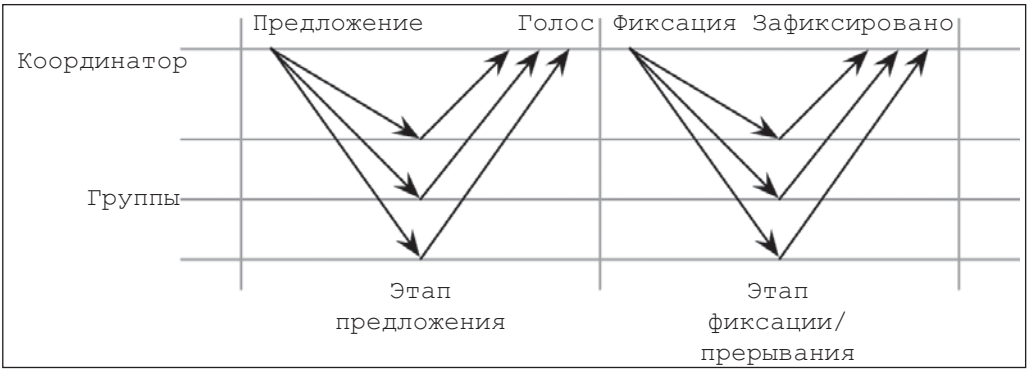

Рис. 13.1. Протокол двухфазной фиксации. В ходе первой фазы когорты уведомляются о новой транзакции. В ходе второй фазы транзакция фиксируется или прерывается

На каждом этапе координатор и когорты должны записывать результаты каждой операции в долговременное хранилище, чтобы можно было воспроизвести состояние и восстановить работоспособность в случае локальных отказов, а также переслать и воспроизвести результаты для других участников.

В контексте СУБД каждый раунд двухфазной фиксации обычно отвечает за одну транзакцию. В ходе фазы подготовки содержимое транзакции (операции, идентификаторы и другие метаданные) передается от координатора в когорты. Транзакция выполняется когортами локально и остается в состоянии *частичной фиксации* (или *предварительной фиксации*), после чего координатор может завершить выполнение в ходе следующей фазы, зафиксировав или прервав транзакцию. К моменту выполнения фиксации содержимое транзакции уже оказывается долговременно сохраненным во всех остальных узлах [BERNSTEIN09].

Отказы когорт в ходе двухфазной фиксации

Давайте рассмотрим несколько сценариев отказов. Например, как показано на рис. 13.2, если одна из когорт дает сбой во время фазы *предложения*, координатор не может произвести фиксацию, так как для этого необходимо, чтобы все голоса были положительными. Если одна из когорт будет недоступна, координатор прервет транзакцию. Это требование отрицательно влияет на доступность: выполнению транзакций может помешать отказ одного узла. Некоторые системы, например Spanner (см. раздел «Распределенные транзакции с использованием протокола Spanner» на с. 286), чтобы повысить обеспечиваемую протоколом степень доступности, выполняют двухфазную фиксацию над группами Паксос, а не отдельными узлами.

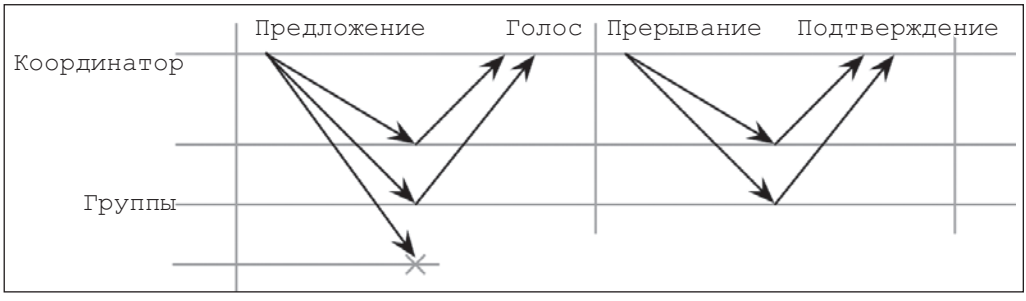

Рис. 13.2. Отказ когорты во время фазы предложения

Главным принципом двухфазной фиксации является *обещание* со стороны когорты того, что после положительного ответа на предложение она не изменит своего решения, в силу чего прервать транзакцию может только координатор.

Если одна из когорт дает сбой *после* принятия предложения, она должна сначала узнать результат голосования, прежде чем сможет корректно обработать значения, поскольку координатор мог прервать фиксацию из-за решений других когорт. После восстановления работоспособности узла когорты она должна выяснять окончательное решение координатора. Обычно это делается путем персистентного хранения журнала решений на стороне координатора и репликации значений решений на давших сбой участников. До этого момента когорта не может обслуживать запросы, так как находится в несогласованном состоянии.

Поскольку в протоколе есть несколько точек, где процессы ожидают других участников (когда координатор собирает голоса или когда когорта ожидает фазы фиксации/прерывания), сбои каналов связи могут привести к потере сообщения, и тогда ожидание будет продолжаться бесконечно. Если координатор не получает ответ от реплики во время фазы предложения, он может запустить механизм времени ожидания и прервать транзакцию.

Отказы координатора в ходе двухфазной фиксации

Если одна из когорт не получает от координатора команду фиксации или прерывания во время второй фазы, как показано на рис. 13.3, она должна попытаться выяснить, какое решение было принято координатором. Координатор, возможно, определился со значением, но не смог сообщить его конкретной реплике. В таких случаях информация о решении может быть реплицирована из журналов транзакций других узлов или из резервного координатора. Репликация решения о фиксации не представляет опасности, поскольку это решение всегда единогласно: весь смысл алгоритма 2PC заключается в том, чтобы либо фиксировать, либо прерывать работу на всех сторонах, а фиксация в одной когорте подразумевает, что все другие когорты также должны произвести фиксацию.

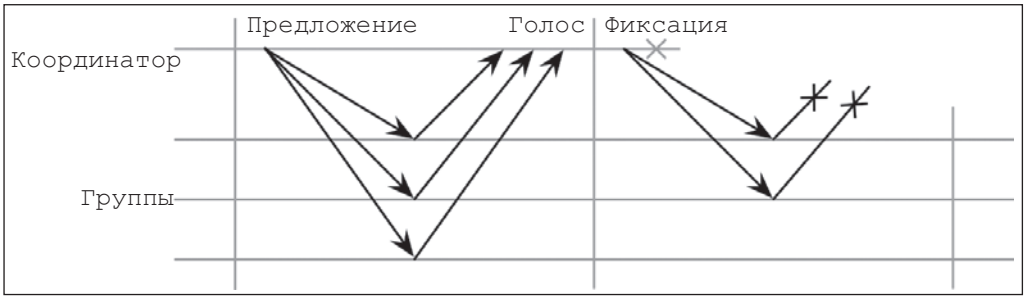

Рис. 13.3. Отказ координатора после фазы предложения

В ходе первой фазы координатор собирает голоса и, следовательно, обещания со стороны когорт о том, что они будут ждать от него явную команду на фиксацию или прерывание. Если происходит сбой координатора после сбора голосов, но еще до рассылки результатов голосования, когорты оказываются в состоянии неопределенности. Такая ситуация показана на рис. 13.4. Когорты не знают, что именно решил координатор и были ли какие-либо участники (потенциально также недоступные) уведомлены о результатах транзакции [BERNSTEIN87].

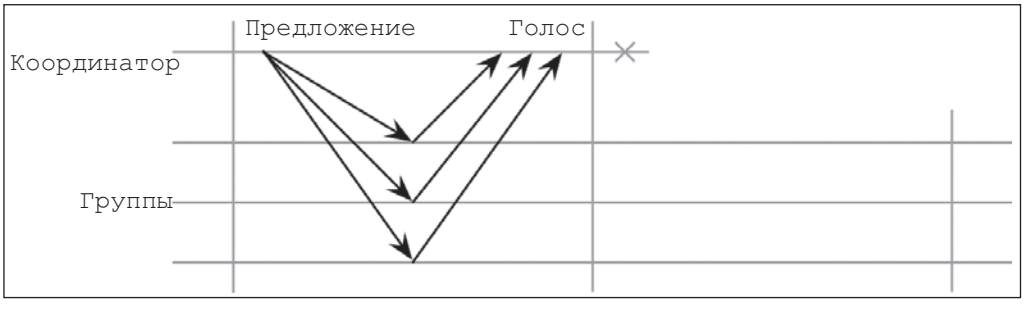

Рис. 13.4. Отказ координатора, прежде чем он смог связаться с какой-либо когортой

Неспособность координатора выполнить фиксацию или прерывание оставляет кластер в неопределенном состоянии. Это означает, что когорты не смогут узнать об окончательном решении в случае непрекращающегося отказа координатора. В силу этой особенности можно сказать, что двухфазная фиксация является алгоритмом *блокирующей* атомарной фиксации. Если координатор не вернется в работоспособное состояние, заменивший его узел должен снова собрать голоса в отношении текущей транзакции и принять окончательное решение.

Многие базы данных используют двухфазную фиксацию: MySQL, PostgreSQL (https://databass.dev/links/6), MongoDB[1] и другие. Двухфазная фиксация часто используется

[1] Однако в документации говорится, что с версии v3.6 протокол 2PC предоставляет только семантику, подобную транзакции: https://databass.dev/links/7.

для реализации распределенных транзакций из-за ее простоты (ее легко обдумывать, реализовывать и отлаживать) и низких затрат (сложность сообщения и количество необходимых запросов и подтверждений невелики). Чтобы уменьшить вероятность только что описанных отказов, важно реализовать надлежащие механизмы восстановления и предусмотреть резервные узлы-координаторы.

Трехфазная фиксация

Чтобы сделать протокол атомарной фиксации устойчивым к отказам координатора и избежать неопределенных состояний, протокол трехфазной фиксации (three phase commit, 3PC) добавляет еще один шаг и время ожидания на обеих сторонах. Эти дополнения позволяют когортам выполнять либо фиксацию, либо прерывание в случае отказа координатора в зависимости от состояния системы. Алгоритм 3PC предполагает синхронную модель и невозможность сбоев связи [BABAOGLU93].

В алгоритме трехфазной фиксации добавляется фаза подготовки перед фазой фиксации/прерывания, в ходе которой координатор рассылает состояния когорт, выявленные в ходе фазы предложения, позволяя протоколу продолжить работу даже в случае отказа координатора. Во всем остальном алгоритм 3PC аналогичен алгоритму двухфазной фиксации и также требует наличия координатора для проведения каждого раунда. Еще одно полезное дополнение в 3PC — время ожидания на стороне когорт. В зависимости от текущего шага решение о фиксации или прерывании принимается по истечении времени ожидания.

Как показано на рис. 13.5, трехфазная фиксация состоит из трех этапов:

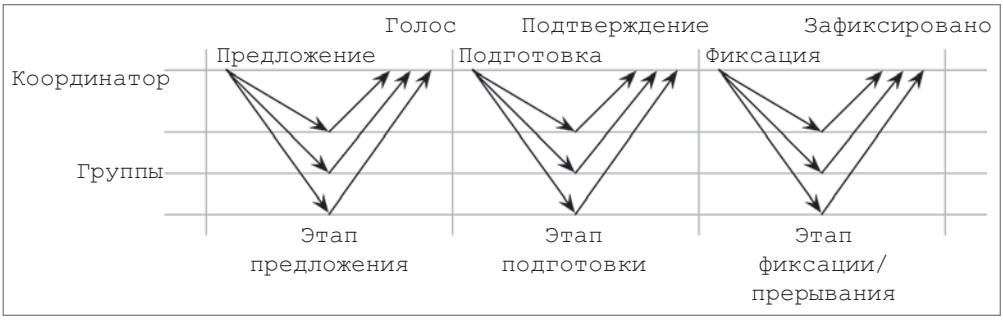

Рис. 13.5. Трехфазная фиксация

Предложение

Координатор рассылает предлагаемое значение и собирает голоса.

Подготовка

Координатор уведомляет когорты о результатах голосования. Если голосование выполнено и все когорты приняли решение зафиксировать транзакцию, то координ-

натор отправляет сообщение *Подготовка*, чтобы они подготовились к фиксации. В противном случае отправляется сообщение *Прерывание* и раунд завершается.

Фиксация

Координатор уведомляет когорты о необходимости фиксации транзакции.

На этапе предложения, так же как в алгоритме 2PC, координатор распределяет предлагаемое значение и собирает голоса от когорт, как показано на рис. 13.5. Если во время этой фазы произойдет отказ координатора и истечет время ожидания операции или если одна из когорт проголосует отрицательно, транзакция будет прервана.

После сбора голосов координатор принимает решение. Если координатор решает продолжить выполнение транзакции, он отправляет команду *Подготовка*. Может случиться так, что координатор не сможет разослать сообщения о подготовке всем когортам или не сможет получить от них подтверждения. В этом случае когорты могут прервать транзакцию после истечения времени ожидания, так как алгоритм не перешел до конца в состояние готовности.

После того как все когорты успешно перейдут в состояние готовности и координатор получит от них подтверждения готовности, транзакция будет зафиксирована даже в случае отказа любой из сторон. Фиксация будет выполнена, поскольку все участники на этом этапе имеют одинаковое представление о состоянии.

Во время фиксации координатор сообщает о результатах фазы подготовки всем участникам, сбрасывая их счетчики времени ожидания и тем самым завершая транзакцию.

Отказы координатора в ходе трехфазной фиксации

Все переходы между состояниями координируются, и когорты не могут перейти к следующему этапу, пока каждая из них не завершит предыдущий этап: прежде чем продолжить, координатор должен ждать реплики. Когорты могут в конечном итоге прервать транзакцию, если до истечения времени ожидания не узнают от координатора, пройден ли этап подготовки.

Как уже говорилось ранее, алгоритм двухфазной фиксации не может восстановиться в случае отказа координатора; при этом когорты могут оставаться в неопределенном состоянии, пока снова не заработает координатор. Алгоритм трехфазной фиксации в той же ситуации не допускает блокирования процессов и позволяет когортам принимать детерминированное решение.

Наихудший сценарий для 3PC — это распад сети, показанный на рис. 13.6. Некоторые узлы при этом успешно переходят в состояние готовности и могут продолжить фиксацию после истечения времени ожидания. Некоторые узлы не могут связаться с координатором и прервут транзакцию после истечения времени ожидания. Такая ситуация приводит к разделению роли лидера: некоторые узлы выполняют фиксацию, а некоторые — прерывание (все это согласно протоколу), оставляя участников в несогласованном и противоречивом состоянии.

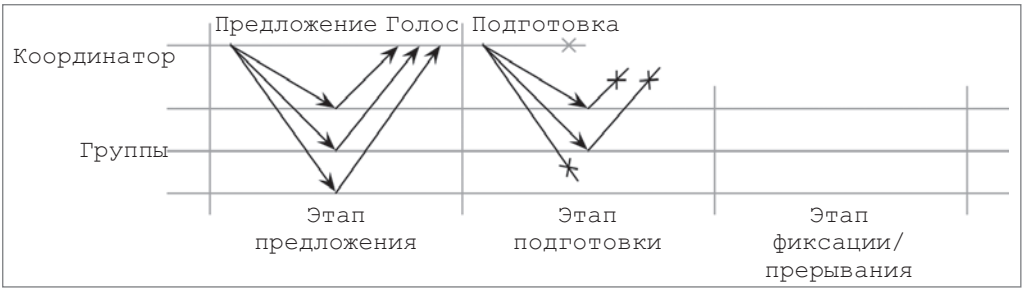

Рис. 13.6. Отказ координатора на втором этапе

Хотя теоретически алгоритм трехфазной фиксации в некоторой степени решает проблему с блокировкой, свойственную двухфазной фиксации, он подразумевает бóльшие затраты на сообщения, добавляет потенциальные противоречия и не работает должным образом при наличии распадов сети. Этим, вероятно, и объясняется то, что он не получил широкого распространения на практике.

Распределенные транзакции с использованием протокола Calvin

Мы уже обсуждали тему затрат на синхронизацию и различные способы решения этой проблемы. Но есть и другие способы уменьшить количество конфликтов и общее количество времени, в течение которого транзакциям нужно удерживать блокировку. Один из этих способов состоит в том, чтобы дать репликам возможность согласовать порядок выполнения и границы транзакций до установки блокировок и дальнейшего выполнения. В таком случае отказы узлов не будут вызывать прерывание транзакции, поскольку узлы могут восстанавливать состояние посредством других участников, которые выполняют ту же транзакцию параллельно.

Традиционные СУБД выполняют транзакции с использованием двухфазной блокировки или оптимистичного управления конкурентностью и не предполагают детерминированный порядок транзакций. Это означает, что для сохранения порядка узлам необходима координация. Детерминированный порядок транзакций устраняет затраты на координацию на этапе выполнения, и поскольку все реплики получают одинаковые входные данные, они также выдают одинаковые результаты. Этот подход широко известен как протокол быстрых распределенных транзакций Calvin [THOMSON12]. Одним из ярких примеров реализации распределенных транзакций с использованием Calvin является система FaunaDB (https://databass.dev/links/8).

Для обеспечения детерминированного порядка в Calvin используется *задатчик* (sequencer) — точка входа для всех транзакций. Задатчик определяет порядок выполнения транзакций и устанавливает глобальную входную последовательность транзакций. Чтобы свести к минимуму конфликтные ситуации и число решений, касающихся серий, временная шкала делится на *эпохи*. Задатчик собирает транзакции

и группирует их в короткие временные окна (в оригинальной статье упоминаются серии по 10 миллисекунд), которые также становятся единицами репликации, в силу чего транзакции не требуется передавать по отдельности.

После успешной репликации серии транзакций задатчик направляет ее *планировщику* (scheduler), который организует выполнение транзакции. Планировщик использует детерминированный протокол планирования, который выполняет части транзакции параллельно, сохраняя при этом указанный задатчиком последовательный порядок выполнения. Поскольку применение транзакции к определенному состоянию гарантирует внесение только тех изменений, которые указаны в транзакции, и порядок транзакций предопределен, репликам не нужно дополнительно связываться с задатчиком.

Каждая транзакция в протоколе Calvin имеет *набор чтения* (зависимости, представляющие собой набор записей данных из текущего состояния базы данных, необходимых для выполнения транзакции) и *набор записи* (результаты выполнения транзакции; другими словами, ее побочные эффекты). Протокол Calvin изначально не поддерживает транзакции, использующие дополнительные операции чтения для определения наборов чтения и записи.

Управляемый планировщиком рабочий поток производит выполнение в четыре этапа:

1. Он анализирует наборы чтения и записи транзакции, определяет локальные для каждого узла записи данных на основе набора чтения и создает список *активных* участников (т. е. тех, которые содержат элементы набора записи и будут модифицировать данные).
2. Он собирает *локальные* данные, необходимые для выполнения транзакции, т. е. записи набора чтения, которые находятся на этом узле. Собранные записи данных направляются соответствующим *активным* участникам.
3. Если этот рабочий поток выполняется на узле активного участника, он получает записи данных, передаваемые от других участников, в качестве партнера других узлов, выполняющих шаг 2.
4. Наконец, он выполняет серию транзакций, сохраняя результаты в локальном хранилище. Ему не нужно пересылать результаты выполнения другим узлам, поскольку они получают те же входные данные для транзакций и сами выполняют операции и сохраняют результаты локально.

В типичных реализациях протокола Calvin подсистемы задатчика и планировщика, подсистемы выполнения и хранения взаимодействуют, как показано на рис. 13.7. Чтобы обеспечить согласованность задатчиков в отношении того, какие именно транзакции должны попасть в текущую эпоху/серию, протокол Calvin использует алгоритм консенсуса Паксос (см. раздел «Паксос» на с. 304) или асинхронную репликацию, при которой в роли лидера выступает выделенная реплика. Хотя использование лидера может снизить задержку, такой подход повышает затраты на восстановление, поскольку узлам необходимо воспроизвести состояние вышедшего из строя лидера, чтобы продолжить выполнение.

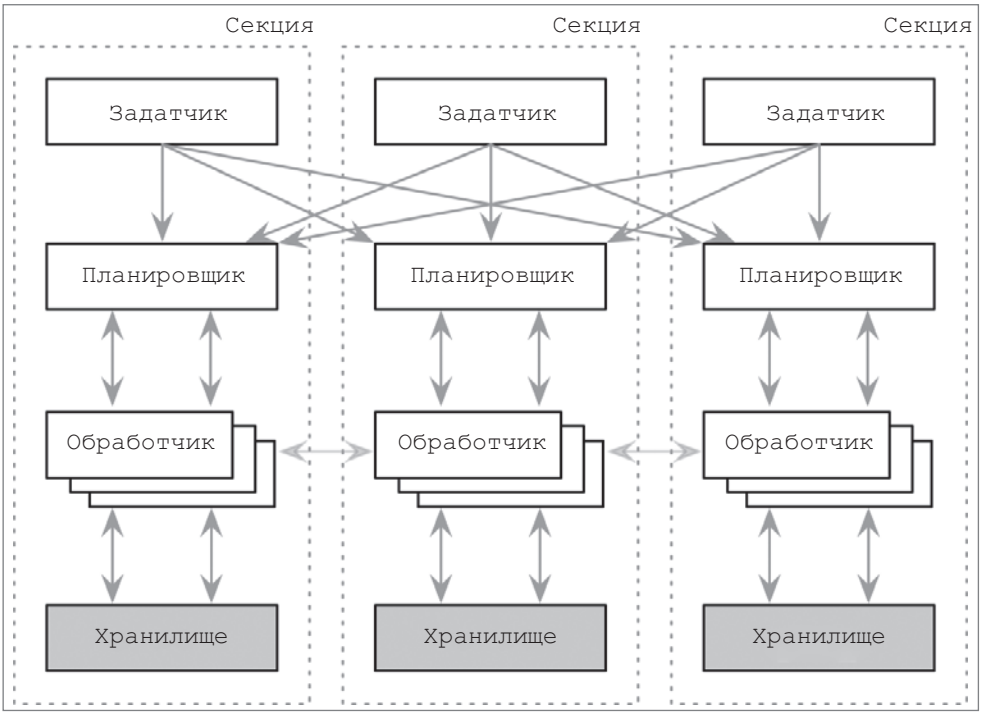

Рис. 13.7. Архитектура протокола Calvin

Распределенные транзакции с использованием протокола Spanner

Протоколу Calvin часто противопоставляется другой подход к управлению распределенными транзакциями, называемый протоколом Spanner [CORBETT12]. Его реализации (или производные) включают в себя несколько баз данных с открытым исходным кодом — в первую очередь это CockroachDB (https://databass.dev/links/9) и YugaByteDB (https://databass.dev/links/10). В то время как протокол Calvin задает глобальный порядок выполнения транзакций путем достижения консенсуса между задатчиками, в протоколе Spanner используется двухфазная фиксация в рамках групп консенсуса, создаваемых для каждой секции (или, иначе говоря, «сегмента»). У протокола Spanner довольно сложная структура, которую мы рассмотрим здесь лишь в общих чертах.

Для достижения согласованности и соблюдения порядка транзакций в протоколе Spanner используется API высокоточных системных часов TrueTime, который также экспонирует временные рамки неопределенности, что позволяет локальным операциям вводить искусственные замедления для выхода за эти временные рамки.

Протокол Spanner предлагает три основных типа операций: *транзакции чтения-записи*, *только чтения* и *чтения моментальных снимков*. Транзакции чтения-записи

требуют блокировок, пессимистичного управления конкурентностью и наличия реплики-лидера. Транзакции только чтения не используют блокировки и могут выполняться в любой реплике. Лидер требуется только для операций чтения *с самой последней* временной меткой, которые извлекают последнее зафиксированное значение из группы Паксос. Операции чтения по определенным временным меткам являются согласованными, так как значения версионируются и содержимое снимка не может измениться после записи. Каждой записи данных присваивается временная метка, которая содержит время фиксации транзакции. Это также подразумевает возможность сохранения нескольких версий записи с разными временными метками.

На рис. 13.8 показана архитектура протокола Spanner. Каждый сервер *Spanserver* (реплика, экземпляр сервера, предоставляющий данные клиентам) содержит несколько так называемых *таблетов* (tablet) с подключенными к ним конечными автоматами Паксос (см. раздел «Паксос» на с. 304). Реплики сгруппированы в наборы, называемые группами Паксос — это единица размещения и репликации данных. У каждой группы Паксос есть долговременный лидер (см. подраздел «Мульти-Паксос» на с. 310). Лидеры общаются друг с другом во время выполнения многосегментных (многосекционных) транзакций.

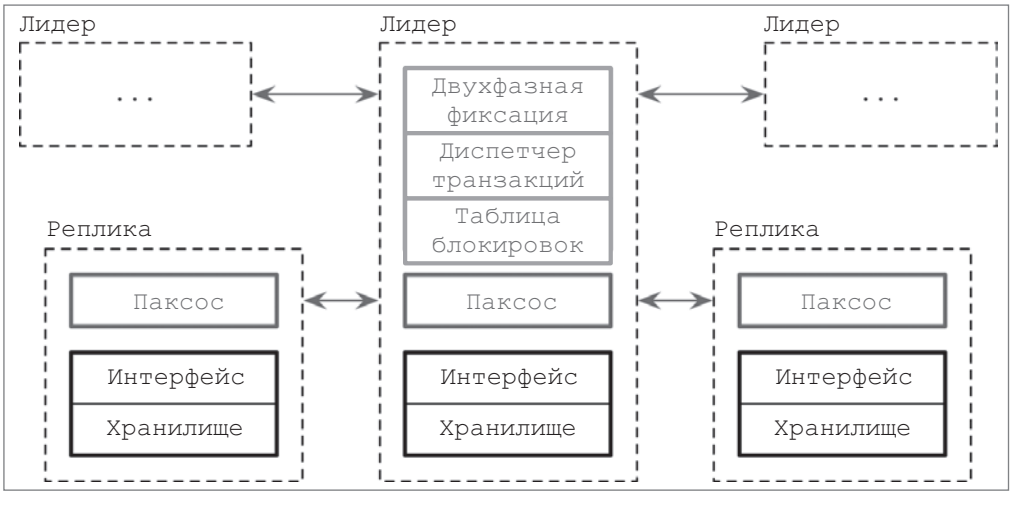

Рис. 13.8. Архитектура протокола Spanner

Каждая операция записи должна выполняться через лидера группы Паксос, в то время как результаты операций чтения могут извлекаться непосредственно из таблета в репликах с актуальными данными. Лидер ведет *таблицу блокировок*, которая используется для реализации управления конкурентностью с использованием механизма двухфазной блокировки (см. подраздел «Управление параллелизмом на основе блокировки» на с. 118) и *диспетчера транзакций*, который отвечает за многосегментные распределенные транзакции. Операции, которым требуется синхронизация

(например, операции записи и чтения в рамках транзакции), должны использовать блокировки из таблицы блокировок, в то время как другие операции (операции чтения моментальных снимков) могут производить прямой доступ к данным.

В случае многосегментных транзакций лидеры групп должны координировать и выполнять двухфазную фиксацию для обеспечения согласованности и использовать двухфазную блокировку для обеспечения изолированности. Поскольку алгоритм двухфазной фиксации для успешной фиксации требует присутствия всех участников, это снижает степень доступности. В протоколе Spanner эта проблема решается путем использования в качестве когорт групп Паксос, а не отдельных узлов. Это означает, что алгоритм 2PC не перестанет работать, даже если некоторые члены группы выйдут из строя. В группе Паксос алгоритм 2PC связывается только с узлом, который выступает в роли лидера.

Группы Паксос используются для согласованной репликации состояний диспетчера транзакций в рамках нескольких узлов. Лидер группы Паксос сначала получает блокировки записи и выбирает временную метку записи, которая гарантированно превышает временную метку любой предыдущей транзакции, после чего вносит в Паксос-группе запись «Подготовка» алгоритма 2PC. Координатор транзакций собирает временные метки, генерирует временную метку фиксации, которая превышает любую временную метку подготовки, и протоколирует в Паксос-группе запись «Фиксация». Затем он ожидает до временной метки, выбранной для фиксации, поскольку он должен гарантировать, что клиенты будут видеть только результаты транзакции с прошедшими временными метками. После этого он отправляет эту временную метку клиенту и лидерам, которые протоколируют запись «Фиксация» с новой временной меткой в своей локальной группе Паксос и наконец могут свободно снимать блокировки.

Односегментным транзакциям не нужно обращаться к диспетчеру транзакций (и следовательно, не нужно выполнять межсегментную двухфазную фиксацию), поскольку для того, чтобы гарантировать порядок и согласованность транзакций внутри сегмента, достаточно свериться с группой Паксос и таблицей блокировок.

Транзакции чтения-записи протокола Spanner обеспечивают порядок сериализации, называемый *внешней согласованностью*: временные метки транзакций отражают порядок сериализации даже в случае распределенных транзакций. Внешняя согласованность имеет свойства реального времени, эквивалентные линеаризуемости: если транзакция T_1 фиксируется до начала транзакции T_2, то временная метка транзакции T_1 меньше, чем временная метка транзакции T_2.

Подводя итог, можно сказать, что в протоколе Spanner используются алгоритм Паксос для согласованной репликации журнала транзакций, двухфазная фиксация для межсегментных транзакций и API TrueTime для обеспечения детерминированного порядка транзакций. Это означает, что многосекционные транзакции требуют более высоких затрат из-за дополнительного раунда двухфазной фиксации по сравнению с протоколом Calvin [ABADI17]. Важно понимать оба подхода, поскольку они позво-

ляют нам выполнять транзакции в распределенных хранилищах данных, разбитых на секции.

Секционирование базы данных

Обсуждая протоколы Spanner и Calvin, мы использовали термины «секционирование» (partitioning) и «секции» («сегменты»). Поговорим о них более подробно. Поскольку хранить все записи на одном узле для большинства современных приложений нереалистично, многие базы данных используют секционирование — логическое разделение данных на более мелкие управляемые сегменты.

Самый простой способ секционирования данных сводится к тому, чтобы разделить их на диапазоны и разрешить *наборам реплик* (replica sets) управлять только определенными диапазонами (секциями). При выполнении запросов клиенты (или координаторы запросов) должны направлять запросы на основе *ключа маршрутизации* (partition key) к нужному набору реплик и для чтения, и для записи. Эта схема секционирования обычно называется *сегментированием* (sharding): каждый набор реплик действует как единый источник для подмножества данных.

Чтобы использовать секции наиболее эффективно, необходимо рассчитывать их размеры с учетом распределения нагрузки и значений. Это означает, что часто используемые диапазоны, с которыми производится большой объем операций чтения/записи, можно разделить на более мелкие секции, чтобы распределить нагрузку между ними. В то же время будет полезно разделить на более мелкие секции и диапазоны значений, отличающиеся повышенной плотностью. Например, если в качестве ключа маршрутизации мы будем использовать почтовый индекс, то поскольку население страны распределено неравномерно, на некоторые диапазоны почтовых индексов будет приходиться больше данных (например, данных о получателях и заказах).

Когда узлы добавляются или удаляются из кластера, база данных должна переразбить данные для поддержания баланса. Для согласованного перемещения мы должны переместить данные до того, как обновим метаданные кластера и начнем направлять запросы к новым целям. Некоторые базы данных выполняют *автоматическое сегментирование* и перемещают данные с помощью алгоритмов размещения, определяющих оптимальный вариант секционирования. Эти алгоритмы используют информацию о нагрузках чтения и записи и количестве данных в каждом сегменте.

Чтобы найти целевой узел по ключу маршрутизации, некоторые СУБД вычисляют *хэш-сумму* ключа и используют некоторую форму сопоставления значения хэш-суммы с идентификатором узла. Одним из преимуществ использования хэш-функций для определения новой позиции реплик является то, что это часто позволяет сократить количество «горячих точек» в диапазоне, поскольку значения хэш-функции сортируются не так, как исходные значения. В то время как два лексикографически близких ключа маршрутизации были бы помещены в один и тот же набор реплик, при использовании хэшированных значений они расположатся в разных наборах.

Самый простой способ сопоставить значения хэш-функции с идентификаторами узлов сводится к тому, чтобы взять остаток от деления значения хэш-суммы на размер кластера. Если в системе имеется `N` узлов, идентификатор целевого узла выбирается путем вычисления остатка от целочисленного деления хэш-суммы `hash(v)` на `N`. Основная проблема этого подхода заключается в том, что при каждом добавлении или удалении узлов с изменением размера кластера с `N` на `N'` будет изменяться и множество значений, получаемых путем вычисления остатка от деления `hash(v)` на `N'`. Это означает, что нам нужно будет перемещать значительную часть данных.

Согласованное хэширование

Чтобы смягчить эту проблему, некоторые базы данных, такие как Apache Cassandra и Riak, используют другую схему секционирования, которую называют *согласованным хэшированием* (consistent hashing). Как упоминалось ранее, значения ключей маршрутизации хэшируются. Значения, возвращаемые хэш-функцией, выстраиваются в *кольцо* таким образом, чтобы после максимально возможного значения следовало наименьшее. Каждый узел получает свою позицию в кольце и отвечает за диапазон значений между позицией предшественника и своей собственной.

Использование согласованного хэширования позволяет сократить количество перемещений, необходимых для поддержания баланса: изменение в кольце затрагивает только *непосредственных соседей* удаляемого или добавляемого узла, а не весь кластер. Определение *согласованное* здесь подразумевает, что при изменении размера хэш-таблицы при наличии `K` возможных ключей хэш-функции и `n` узлов нам в среднем придется переместить только `K/n` ключей. То есть изменение диапазона согласованной хэш-функции ведет к минимальным изменениям в ее результатах [KARGER97].

Распределенные транзакции с использованием библиотеки Percolator

Вернемся к теме распределенных транзакций. Уровни изолированности трудно применять из-за допускаемых ими аномалий чтения и записи. Если приложению не требуется сериализуемость, одним из способов избежать аномалий записи, описанных в стандарте SQL-92, является использование модели транзакций, называемой *изоляцией моментального снимка* (snapshot isolation, SI).

Изоляция моментального снимка гарантирует согласованность всех операций чтения, выполняемых в рамках транзакции, с моментальным снимком базы данных, содержащим все значения, которые *были зафиксированы до* временной метки, соответствующей началу транзакции. В случае *конфликта операций записи* (когда две выполняемые параллельно транзакции пытаются произвести запись в одну и ту же ячейку) будет зафиксирована только одна из этих операций. Обычно это выражается в виде формулировки «побеждает первый фиксирующий».

Изоляция моментального снимка исключает возможность аномалии *искажения чтения*, появление которой допускается при уровне изолированности «чтение фиксированных данных». Например, допустим, что сумма `x` и `y` должна быть равна `100`. Транзакция `T1` выполняет операцию чтения `read(x)` и считывает значение `70`. Транзакция `T2` обновляет два значения с помощью операций `write(x, 50)` и `write(y, 50)` и фиксируется. Если транзакция `T1` попытается выполнить операцию `read(y)` и продолжить выполнение, используя значение `y`, вновь зафиксированное транзакцией `T2, (50)`, это приведет к несогласованности. Значение `x`, которое транзакция `T1` считала перед фиксацией транзакции `T2`, и новое значение `y` не соответствуют друг другу. Поскольку изоляция моментального снимка делает видимыми для транзакций только значения, расположенные во времени до определенной временной метки, новое значение `y`, `50`, не будет видимо для транзакции `T1` [BERENSON95].

Изоляция моментального снимка обладает несколькими удобными свойствами:

- Она позволяет выполнять *только* повторяемые операции чтения зафиксированных данных.
- Значения являются согласованными, поскольку они считываются из моментального снимка с определенной временной меткой.
- Во избежание рассогласованности конфликтующие операции записи прерываются и выполняются повторно.

Несмотря на это, истории при использовании изоляции моментального снимка не сериализуемы. Поскольку прерываются только конфликтующие операции записи в одни и те же ячейки, мы все равно можем получить *искажение записи* (см. раздел «Аномалии чтения и записи» на с. 113). Искажение записи происходит, когда две транзакции изменяют непересекающиеся наборы значений с сохранением инвариантов для записываемых данных. Обеим транзакциям разрешено зафиксироваться, но комбинация операций записи, выполняемых этими транзакциями, может нарушать эти инварианты.

Изоляция моментального снимка обеспечивает семантику, которая может быть полезна для многих приложений, и ее главное преимущество в эффективном чтении, так как не нужно устанавливать блокировки, поскольку данные моментального снимка не могут подвергаться изменениям.

Percolator — это библиотека, которая реализует транзакционный API поверх распределенной базы данных Bigtable (см. подраздел «Хранилища с широкими столбцами» на с. 33). Это отличный пример надстройки транзакционного API поверх существующей системы. Библиотека Percolator хранит записи данных, позиции элементов зафиксированных данных (метаданные операций записи) и блокировки в разных столбцах. Чтобы избежать возникновения состояния гонки и надежно блокировать таблицы в одном удаленном вызове процедур, она использует API с условной мутацией Bigtable, который позволяет ей выполнять операции чтения-модификации-записи с помощью одного удаленного вызова.

Каждая транзакция должна обращаться к *оракулу временных меток* (timestamp oracle) (источнику монотонно увеличивающихся временных меток, согласованных в рамках кластера), делая это дважды: для начальной временной метки транзакции и во время фиксации. Операции записи буферизуются и фиксируются с использованием двухфазной фиксации, управляемой клиентом (см. раздел «Двухфазная фиксация» на с. 277).

Рисунок 13.9 показывает, как содержимое таблицы изменяется во время выполнения шагов транзакции.

		Данные	Блокировки	Запись метаданных
Счет 1	TS2	-	-	TS1 — последняя
	TS1	$100	-	-
Счет 2	TS2	-	-	TS1 — последняя
	TS1	$200	-	-

а) начальное состояние перед перемещением $150 со счета 2 на счет 1

		Данные	Блокировки	Запись метаданных
Счет 1	TS3	$250	Первичная	-
	TS2	-	-	TS1 — последняя
	TS1	$100	-	-
Счет 2	TS3	$50	Первичная на счёте 1	-
	TS2	-	-	TS1 — последняя
	TS1	$200	-	-

б) состояние после установки блокировок и обновления счетов

		Данные	Блокировки	Запись метаданных
Счет 1	TS4	-	-	TS3 — последняя
	TS3	$250	-	-
	TS2	-	-	TS1 — последняя
	TS1	$100	-	-
Счет 2	TS4	-	-	TS3 — последняя
	TS3	$50	Первичная на счёте 1	-
	TS2	-	-	TS1 — последняя
	TS1	$200	-	-

в) фиксация транзакции снимает блокировки и обновляет метаданные с последней временной меткой

Рис. 13.9. Этапы выполнения транзакции библиотекой Percolator. Транзакция снимает $150 со счета 2 и заносит их на счет 1

а) Исходное состояние. После выполнения предыдущей транзакции TS1 является самой последней временной меткой для обоих счетов. Ни одна блокировка не установлена.

б) Первый этап, называемый *предварительной записью*. Транзакция пытается установить блокировки для всех ячеек, записываемых во время транзакции. Одна из блокировок помечается как *основная* и используется для восстановления клиента. Транзакция выполняет проверку на наличие конфликтов: не записала ли уже какая-либо другая транзакция какие-либо данные с более поздней временной меткой и нет ли на какой-либо временной метке неосвобожденных блокировок. Если обнаружен какой-либо конфликт, транзакция прерывается.

в) Если все блокировки были успешно установлены и возможность конфликта исключена, выполнение транзакции может быть продолжено. На втором этапе клиент снимает свои блокировки, начиная с основной. Он публикует результаты своей операции записи, заменяя блокировку записываемой записью и обновляя метаданные записи временной меткой самого последнего элемента данных.

Поскольку при попытке зафиксировать транзакцию может произойти отказ клиента, мы должны убедиться, что частичные транзакции завершены или выполнен их откат. Если более поздняя транзакция выявляет незавершенное состояние, она должна попытаться снять основную блокировку и зафиксировать транзакцию. Если основная блокировка уже освобождена, содержимое транзакции должно быть зафиксировано. Только одна транзакция может одновременно удерживать блокировку, и все переходы состояний являются атомарными, что делает невозможным возникновение ситуации, когда две транзакции пытаются выполнить операции над содержимым.

Изоляция моментального снимка является важной и полезной абстракцией, широко используемой в обработке транзакций. Поскольку она упрощает семантику, исключает некоторые аномалии и открывает возможность для улучшения параллелизма и производительности, этот уровень изолированности предлагают многие системы с многоверсионным управлением конкурентным доступом (MVCC).

Одним из примеров баз данных, основанных на модели Percolator, является TiDB (Ti означает «титан»). TiDB — это строго согласованная, высокодоступная и горизонтально масштабируемая база данных с открытым исходным кодом, совместимая с MySQL.

Исключение координации

Еще одной концепцией, касающейся затрат на сериализуемость и попыток уменьшить объем работы по координации при обеспечении сильных гарантий согласованности, является исключение координации [BAILIS14b]. Координацию можно исключить при соблюдении ограничений целостности данных, если операции обладают свойством конфлюентности инвариантов. Конфлюентность инвариантов (invariant

confluence) — это свойство, гарантирующее, что два соблюдающих инварианты, но расходящихся состояния базы данных могут быть объединены в одно действительное, конечное состояние. Инварианты в этом случае сохраняют согласованность в терминах ACID-свойств.

Поскольку любые два действительных состояния можно объединить в действительное состояние, операции, обладающие свойством конфлюентности инвариантов, могут выполняться без дополнительной координации, что значительно повышает производительность и потенциал масштабируемости.

Чтобы сохранить этот инвариант, в дополнение к определению операции, переводящей базу данных в новое состояние, мы должны определить функцию *слияния*, принимающую два состояния. Эта функция используется в том случае, если состояния обновлялись независимо, и возвращает расходящиеся состояния к конвергенции.

Транзакции выполняются для локальных версий базы данных (моментальных снимков). Если транзакции для выполнения необходимо какое-либо состояние от других секций, это состояние становится доступным для нее локально. Если транзакция фиксируется, результирующие изменения, сделанные в локальном снимке, переносятся и объединяются со снимками на других узлах. Модель системы, которая позволяет исключить координацию, должна гарантировать следующие свойства:

Глобальная действительность

Требуемые инварианты всегда соблюдаются как для слитых, так и для расходящихся зафиксированных состояний базы данных, а транзакции не могут видеть недействительные состояния.

Доступность

Если клиенту доступны все узлы, содержащие состояния, транзакция должна принять решение о фиксации или прервать работу, если фиксация нарушит один из инвариантов транзакции.

Конвергенция

Узлы могут независимо поддерживать свои локальные состояния, но при отсутствии дальнейших транзакций и неопределенных распадов сети у них должна быть возможность прийти к единому состоянию.

Свобода координации

Выполнение локальной транзакции не зависит от операций над локальными состояниями, выполняемых от имени других узлов.

Одним из примеров реализации исключения координации являются транзакции RAMP (Read-Atomic Multi Partition, многосекционные с атомарным чтением) [BAILIS14c]. Транзакции RAMP используют многоверсионное управление конкурентным доступом и метаданные текущих выполняемых операций, чтобы извлекать любые отсутствующие обновления состояния из других узлов, позволяя параллельное выполнение операций чтения и записи. Например, операции чтения, которые

перекрываются с какой-либо операцией записи, модифицирующей ту же запись, можно обнаружить и при необходимости *исправить* путем извлечения необходимой информации из метаданных текущей операции записи в дополнительном раунде обмена данными.

Использование основанных на блокировке подходов в распределенной среде может быть не самой лучшей идеей, и вместо этого транзакции RAMP обеспечивают два свойства:

Независимость синхронизации

Транзакции одного клиента не будут останавливать, прерывать или заставлять ждать транзакции другого клиента.

Независимость секций

Клиентам не нужно связываться с секциями, значения которых не используются в их транзакциях.

Вместе с транзакциями RAMP вводится уровень изолированности с атомарным чтением: транзакции не видят незавершенные изменения состояний, произведенные еще выполняемыми, незафиксированными и прерванными транзакциями. То есть для параллельных транзакций видимы либо все обновления, произведенные транзакцией, либо ни одно из них. Согласно этому определению, уровень изолированности с атомарным чтением также исключает *частичное чтение* (partial read), когда транзакция видит только подмножество результатов операций записи, выполненных какой-либо другой транзакцией.

Транзакции RAMP обеспечивают атомарную видимость результатов операций записи без необходимости во взаимном исключении, которое часто применяется для обеспечения этого свойства в других решениях, таких как распределенные блокировки. Это означает, что транзакции могут продолжать выполнение, не останавливая друг друга.

Транзакции RAMP распространяют метаданные транзакции, которые позволяют операциям чтения обнаруживать выполняемые в данный момент параллельные операции записи. Используя эти метаданные, транзакции могут определять наличие более новых версий записей, находить и извлекать последние версии и работать с ними. Чтобы исключить координацию, все локальные решения о фиксации должны быть действительными и на глобальном уровне. В транзакциях RAMP это обеспечивается путем предъявления требования о том, чтобы к тому времени, когда результаты операции записи становятся видимыми в одной секции, результаты операций записи, выполняемых из этой транзакции во всех других задействованных секциях, также становились видимыми для операций чтения в этих секциях.

Чтобы операции чтения и записи могли продолжать выполнение, не блокируя другие параллельные операции чтения и записи и сохраняя уровень изолированности с атомарным чтением как локально, так и в масштабе всей системы (во всех других секциях, изменяемых фиксируемой транзакцией), операции записи в транзакциях RAMP создаются и делаются видимыми посредством двухфазной фиксации:

Подготовка

На первом этапе операции записи подготавливаются и помещаются в соответствующие целевые секции, оставаясь невидимыми.

Фиксация/прерывание

На втором этапе публикуются изменения состояния, произведенные операцией записи фиксируемой транзакции. Изменения становятся доступными атомарно для всех секций или откатываются.

Транзакции RAMP допускают наличие нескольких версий одной и той же записи в любой момент времени, включая последнее значение, выполняемые незафиксированные изменения или устаревшие версии, перезаписанные более поздними транзакциями. Устаревшие версии должны храниться только для выполняемых в данный момент запросов на чтение. После завершения всех параллельных операций чтения устаревшие значения можно удалить.

Обеспечить эффективность и масштабируемость распределенных транзакций сложно из-за затрат на координацию, связанных с предотвращением, обнаружением и исключением конфликтов для параллельных операций. Объем этих затрат тем больше, чем масштабнее система или чем больше транзакций она пытается обработать. Подходы, описанные в этом разделе, нацелены на уменьшение объема работы по координации за счет выявления тех случаев, когда можно исключить координацию, с помощью инвариантов и выполнения этой работы в полном объеме лишь тогда, когда это абсолютно необходимо.

Итоги

В этой главе мы обсудили несколько способов реализации распределенных транзакций. Сначала мы обсудили два алгоритма атомарной фиксации — алгоритмы двух- и трехфазной фиксации. Большим преимуществом этих алгоритмов является то, что они легки для понимания и реализации. Но они имеют и ряд недостатков. Алгоритм двухфазной фиксации требует, чтобы координатор (или хотя бы его заместитель) функционировал в течение всего процесса фиксации, что значительно снижает доступность. Алгоритм трехфазной фиксации в ряде случаев снимает это требование, но может приводить к разделению роли лидера в случае распада сети.

Распределенные транзакции в современных СУБД часто реализуются с использованием алгоритмов достижения консенсуса, которые мы обсудим в следующей главе. Например, протоколы Calvin и Spanner, обсуждаемые в этой главе, используют алгоритм Паксос.

Алгоритмы достижения консенсуса сложнее алгоритмов атомарной фиксации, но обладают намного большей отказоустойчивостью и отделяют решения от их инициаторов, позволяя участникам выбирать значение, а не просто принимать или отклонять предложенное значение [GRAY04].

Дополнительная литература

Если вы хотите узнать больше о концепциях, упомянутых в этой главе, вы можете обратиться к следующим источникам:

Интеграция атомарной фиксации с локальными подсистемами обработки транзакций и восстановления

Silberschatz, Abraham, Henry F. Korth, and S. Sudarshan. 2010. Database Systems Concepts (6th Ed.). New York: McGraw-Hill.

Garcia-Molina, Hector, Jeffrey D. Ullman, and Jennifer Widom. 2008. Database Systems: The Complete Book (2nd Ed.). Boston: Pearson[1].

Последние достижения в области распределенных транзакций (в хронологическом порядке; данный список не является исчерпывающим)

Cowling, James and Barbara Liskov. 2012. "Granola: low-overhead distributed transaction coordination." In Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC '12): 21–21. USENIX.

Balakrishnan, Mahesh, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. "Tango: distributed data structures over a shared log." In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13): 324–340.

Ding, Bailu, Lucja Kot, Alan Demers, and Johannes Gehrke. 2015. "Centiman: elastic, high performance optimistic concurrency control by watermarking." In Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15): 262–275.

Dragojević, Aleksandar, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. "No compromises: distributed transactions with consistency, availability, and performance." In Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15): 54–70.

Zhang, Irene, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. "Building consistent transactions with inconsistent replication." In Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15): 263–278.

[1] *Ульман Джеффри Д., Уидом Дженнифер, Гарсиа-Молина Гектор*. Системы баз данных. Полный курс. М.: Вильямс, 2003. — *Примеч. ред.*

ГЛАВА 14

Консенсус

Мы обсудили довольно много концепций из области распределенных систем, начиная с основ, таких как каналы и процессы, проблемы распределенных вычислений; затем поговорили о моделях отказов, детекторах отказов и выборе лидера; обсудили модели согласованности и теперь, наконец, готовы собрать все это воедино и подойти к кульминации исследования распределенных систем — распределенному консенсусу.

Алгоритмы достижения консенсуса в распределенных системах позволяют нескольким процессам достичь согласия относительно некоего значения. Теорема ФЛП (см. раздел «Невозможность Фишера–Линча–Патерсона» на с. 207) показывает, что невозможно достичь консенсуса в полностью асинхронной системе за ограниченное время. Даже если доставка сообщений гарантирована, один процесс не сможет узнать, произошел ли сбой другого процесса или тот просто медленно работает.

В главе 9 мы говорили, что существует компромисс между точностью и скоростью обнаружения отказов. Алгоритмы достижения консенсуса предполагают асинхронную модель и гарантируют безопасность, в то время как внешний детектор отказов может стать источником информации о других процессах, гарантируя живучесть [CHANDRA96]. Поскольку обнаружение отказов не работает абсолютно точно, в некоторых случаях алгоритм достижения консенсуса будет ожидать, пока будет обнаружен отказ процесса, или перезапускаться из-за ошибочного отнесения некоторого процесса к числу сбойных.

Процессы должны согласовать некоторое значение, предложенное одним из участников, даже если некоторые из них дадут сбой. Процесс считается *корректным*, если он не дал сбой и продолжает выполнение шагов алгоритма. Консенсус чрезвычайно полезен для расположения событий в определенном порядке и обеспечения согласованности между участниками. Используя консенсус, мы можем создать систему, в которой процессы будут переходить от одного значения к следующему без потери уверенности в том, какие именно значения видны клиентам.

С теоретической точки зрения алгоритмы достижения консенсуса обладают тремя свойствами:

Согласованность (agreement)

Значение, по которому принимается решение, одинаково для всех *корректных* процессов.

Действительность (validity)

Принятое значение было предложено одним из процессов.

Завершаемость (termination)

Все *корректные* процессы в конечном итоге приходят к решению.

Каждое из этих свойств играет чрезвычайно важную роль. Свойство согласованности является неотъемлемой составляющей общепринятых представлений о консенсусе. В словаре консенсус определяется с помощью таких слов, как «единодушие» (https://databass.dev/links/66). Это означает, что после достижения согласованности ни один процесс не может иметь отличающееся мнение о результате. Это можно сравнить с тем, как несколько друзей договариваются о месте и времени предстоящей встречи: каждый из друзей хочет прийти на встречу и согласовываются только некоторые детали мероприятия.

Свойство действительности является важным по той причине, что без него консенсус может оказаться тривиальным. Алгоритмы достижения консенсуса требуют, чтобы все процессы согласовали некоторое значение. Если процессы будут использовать в качестве результата решения некоторое заранее определенное, произвольное дефолтное значение, не принимая во внимание предложенные значения, то они достигнут единодушия, но результат такого алгоритма не будет ни действительным, ни полезным для реального применения.

Без свойства завершаемости наш алгоритм будет работать вечно, никогда не достигнув результата, или будет бесконечно долго ждать, пока восстановится сбойный процесс, что тоже малополезно. Рано или поздно процессы должны прийти к согласию, и для того, чтобы алгоритм достижения консенсуса был применим на практике, это должно происходить довольно быстро.

Рассылка

Рассылка (broadcast) — это абстракция связи, часто используемая в распределенных системах. Алгоритмы рассылки используются для распространения информации среди некоторого множества процессов. Существует много разных алгоритмов рассылки, с разными предположениями и разными гарантиями. Рассылка является важным примитивом, который используется во многих областях, включая алгоритмы достижения консенсуса. Мы уже обсуждали одну из форм рассылки — распространение слухов (см. раздел «Распространение сплетен» на с. 268).

Рассылки часто используются для репликации базы данных, когда один узел-координатор должен распространять данные на всех остальных участников. Однако сделать этот процесс надежным не так просто: если координатор даст сбой после распространения сообщения лишь на некоторую часть узлов, он оставит систему в несогласованном состоянии: некоторые узлы будут видеть новое сообщение, а некоторые — нет.

Самым простым и прямолинейным способом рассылки сообщений является *рассылка по возможности* [CACHIN11]. В этом случае отправитель несет ответственность за доставку сообщений всем адресатам. Если у него не получается, другие участники

не пытаются ретранслировать сообщение, поэтому при сбое координатора этот тип рассылки просто прекратит работу, не сообщив об ошибке.

Чтобы рассылка была *надежной* (reliable), она должна гарантировать, что все корректные процессы получат одинаковые сообщения, даже если отправитель даст сбой в ходе процесса передачи.

Мы можем реализовать простейшую версию надежной рассылки, используя детектор отказов и резервные механизмы. Самый простой резервный механизм сводится к тому, чтобы позволить каждому получившему сообщение процессу переслать его всем остальным известным ему процессам. В случае отказа процесса-источника другие процессы обнаруживают этот отказ и продолжают рассылать сообщение, что обеспечивает *лавинную рассылку* (flooding broadcast) по сети N^2 сообщений (как показано на рис. 14.1). Даже в случае отказа отправителя сообщения подхватываются и доставляются остальной частью системы, что повышает ее надежность и позволяет всем получателям видеть одни и те же сообщения [CACHIN11].

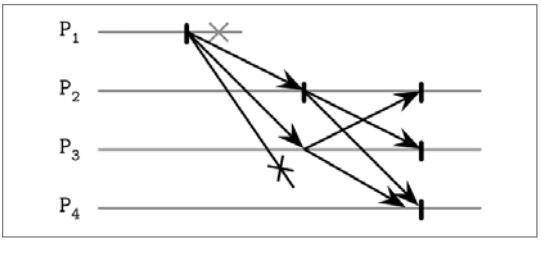

Рис. 14.1. Рассылка

Один из недостатков этого подхода состоит в том, что он использует N^2 сообщений, где N — количество *оставшихся* получателей (поскольку каждый рассылающий процесс исключает себя и исходный процесс). В идеале нам бы хотелось уменьшить количество сообщений, необходимых для надежной рассылки.

Атомарная рассылка

Хотя только что описанный алгоритм лавинной рассылки может гарантировать доставку сообщений, он не гарантирует какой-либо конкретный порядок доставки. Гарантируется лишь то, что сообщения достигнут пункта назначения в конечном итоге в неизвестный момент. Если нам нужно доставлять сообщения в определенном порядке, мы должны использовать *атомарную рассылку* (atomic broadcast) (или, иначе говоря, *многоадресную рассылку с соблюдением общего порядка*), которая гарантирует и надежную доставку, и соблюдение общего порядка.

В то время как надежная рассылка гарантирует согласованность процессов в отношении набора доставляемых сообщений, атомарная рассылка дополнительно

гарантирует и согласованность в отношении последовательности сообщений (т. е. обеспечивается одинаковый порядок доставки сообщений для каждого адресата).

Таким образом, атомарная рассылка должна гарантировать два важных свойства:

Атомарность

Процессы должны достигнуть согласия в отношении набора получаемых сообщений. Либо все работающие процессы доставляют сообщение, либо ни один из них.

Порядок

Все работающие процессы доставляют сообщения в одном и том же порядке.

В данном случае сообщения доставляются *атомарно*: каждое сообщение либо доставляется всем процессам, либо не доставляется ни одному из них; если сообщение доставляется, то каждое другое сообщение располагается до или после этого сообщения.

Виртуальная синхронизация

Один из механизмов групповой коммуникации с использованием рассылки называется *виртуальной синхронизацией* (virtual synchrony). Атомарная рассылка позволяет доставлять полностью упорядоченные сообщения *статической* группе процессов, а виртуальная синхронизация доставляет полностью упорядоченные сообщения *динамической* группе одноранговых узлов.

Виртуальная синхронизация организует процессы в группы. Пока группа существует, сообщения доставляются всем ее членам в одинаковом порядке. В данном случае порядок не указывается моделью и некоторые реализации могут использовать это в своих интересах для повышения производительности при условии, что обеспечиваемый ими порядок будет согласованным в рамках всех членов [BIRMAN10].

Процессы имеют одинаковое представление группы, а сообщения привязываются к идентичности группы: процессы могут видеть одинаковые сообщения, только если относятся к одной и той же группе.

Если один из участников присоединяется к группе, покидает группу или дает сбой, после чего выводится из группы, то меняется и представление группы. Это происходит путем объявления об изменении группы всем ее участникам. Каждое сообщение уникальным образом ассоциируется с той группой, в которой оно было создано.

Виртуальная синхронизация проводит различие между *получением* сообщения (когда участник группы получает сообщение) и его *доставкой* (когда сообщение получают все участники группы). Если сообщение было *отправлено* в некотором представлении, то оно может быть *доставлено* только в этом же представлении, в чем можно убедиться путем сравнения текущей группы с той группой, с которой ассоциировано сообщение. Полученные сообщения остаются в очереди до тех пор, пока процесс не получит уведомление об успешной доставке.

Поскольку каждое сообщение относится к определенной группе, пока все процессы в группе не *получат* его до изменения представления, ни один член группы не может считать это сообщение *доставленным*. Это подразумевает, что все сообщения отправляются и доставляются *между* изменениями представления, что дает нам гарантии атомарной доставки. В данном случае представления групп выступают в роли барьера, который не могут пересекать рассылки сообщений.

Некоторые алгоритмы рассылки упорядочивают сообщения с помощью одного процесса (задатчика (sequencer)), который задает порядок их следования. Такие алгоритмы обычно проще в реализации, но используют обнаружение отказов лидера для обеспечения живучести. Использование задатчика может повысить производительность, поскольку нам не нужно обеспечивать консенсус между процессами для каждого сообщения — вместо этого можно использовать локальное представление задатчика. Кроме того, этот подход допускает масштабирование путем секционирования запросов.

Несмотря на свою техническую обоснованность, виртуальная синхронизация не получила широкого распространения и обычно не используется в коммерческих системах конечного пользователя [BIRMAN06].

Протокол атомарной рассылки ZooKeeper

Одной из самых популярных и широко известных реализаций атомарной рассылки является протокол ZooKeeper (Zookeeper Atomic Broadcast, ZAB https://databass.dev/links/67), используемый в иерархически распределенном хранилище типа «ключ–значение» Apache Zookeeper [HUNT10] [JUNQUEIRA11] для обеспечения общего порядка событий и атомарной доставки, необходимой для поддержания согласованности между состояниями реплик.

Процессы в ZAB могут играть одну из двух ролей: *лидер* (leader) и *последователь* (follower). Роль лидера является временной. Лидер управляет выполнением шагов алгоритма, рассылает сообщения последователям и устанавливает порядок событий. Чтобы записывать новые записи и считывать самые последние значения, клиенты подключаются к одному из узлов кластера. Если этот узел является лидером, то он просто обрабатывает запрос. В противном случае он перенаправляет запрос лидеру.

Чтобы гарантировать уникальность лидера, временная шкала протокола разбита на *эпохи*, идентифицируемые уникальным монотонно и последовательно возрастающим номером. В течение одной эпохи может быть только один лидер. Процесс начинает свое выполнение с поиска *потенциального лидера*, используя любой алгоритм выбора, способный с высокой вероятностью выбирать работоспособный процесс. Поскольку безопасность гарантируется дальнейшими шагами алгоритма, определение потенциального лидера служит главным образом для оптимизации производительности. Потенциальный лидер также может появиться вследствие выхода из строя предыдущего лидера.

После выбора потенциального лидера он выполняет протокол, делая это в три этапа:

Выявление (discovery)

Потенциальный лидер узнает о последней эпохе, известной каждому другому процессу, и предлагает новую эпоху, идентификатор которой *больше*, чем у текущих эпох любого из последователей. Последователи отвечают на предложение эпохи сообщением с идентификатором последней транзакции, замеченной в предыдущей эпохе. После этого шага ни один процесс не примет рассылку с предложениями для более ранних эпох.

Синхронизация (synchronisation)

Этот этап используется для восстановления работоспособности после выхода из строя предыдущего лидера и приведения в актуальное состояние отстающих последователей. Потенциальный лидер посылает сообщение последователям, предлагая себя на роль лидера новой эпохи, и собирает их подтверждения. После получения подтверждений роль лидера утверждается. После этого шага последователи не будут принимать попытки других процессов предложить себя на роль лидера эпохи. Во время синхронизации новый лидер обеспечивает наличие одинаковой истории у всех последователей и доставляет зафиксированные предложения от утвержденных лидеров предыдущих эпох. Эти предложения доставляются *до* доставки предложений новой эпохи.

Рассылка

После синхронизации последователей начинается активный обмен сообщениями. На этом этапе лидер получает клиентские сообщения, устанавливает их порядок и рассылает их последователям: он отправляет новое предложение, ожидает подтверждения от кворума последователей и, наконец, фиксирует его. Этот процесс похож на двухфазную фиксацию без прерываний: голоса могут быть только подтверждениями, и клиент не может голосовать против предложения действительного лидера. Однако предложения от лидеров некорректных эпох не должны приниматься. Этап рассылки прекращается после того, как лидер дает сбой, отделяется от последователей распадом сети или относится к числу потенциально сбойных из-за задержки сообщения.

На рис. 14.2 показаны три этапа алгоритма ZAB и обмен сообщениями на каждом этапе.

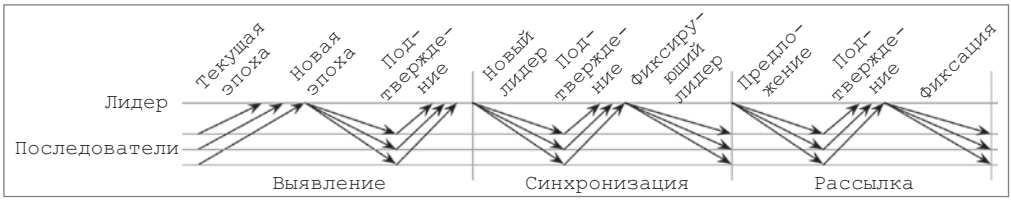

Рис. 14.2. Общая схема протокола ZAB

Безопасность этого протокола гарантируется, если последователи принимают предложения только от лидера установленной эпохи. Хотя в процессе выбора могут уча-

ствовать два процесса, только один из них может победить в этом процессе и стать лидером эпохи. Также предполагается, что процессы добросовестно выполняют предписываемые протоколом шаги.

И лидер, и последователи используют контрольные пакеты для определения жизнеспособности удаленных процессов. Если лидер не получает контрольные пакеты от кворума последователей, то он оставляет роль лидера и перезапускает процесс выбора. Аналогичным образом если один из последователей выявляет сбой лидера, он запускает новый процесс выбора.

Сообщения отправляются с соблюдением некоторого общего порядка, и лидер не пытается отправить следующее сообщение, пока не будет подтверждено получение предыдущего сообщения. Даже если некоторые сообщения получаются последователем более одного раза, их повторное применение не вызывает дополнительных побочных эффектов при соблюдении порядка доставки. Протокол ZAB допускает одновременное получение от клиентов нескольких одновременных изменений состояния, поскольку получать запросы на запись, устанавливать порядок событий и рассылать изменения может только уникальный лидер.

Общий порядок сообщений также позволяет протоколу ZAB повысить эффективность восстановления. На этапе синхронизации последователи отправляют в ответ самое последнее зафиксированное предложение. Для восстановления лидер может просто выбрать узел с самым последним предложением, и при этом обычно нужно копировать сообщения только с этого узла.

Одним из преимуществ протокола ZAB является его эффективность: процесс рассылки требует только двух раундов сообщений, а при отказе лидера восстановление можно произвести путем потоковой передачи недостающих сообщений из одного процесса, хранящего актуальные данные. Наличие долговременного лидера может оказать положительное влияние на производительность: нам не потребуются дополнительные раунды достижения консенсуса для установления истории событий, поскольку лидер может задавать их порядок на основе своего локального представления.

Паксос

Задача атомарной рассылки эквивалентна задаче достижения консенсуса в асинхронной системе с отказами-авариями [CHANDRA96], поскольку участники должны согласовать порядок сообщений и у них должна быть возможность узнать о нем. Много общего можно найти и в принципах, и в реализации алгоритмов атомарной рассылки и достижения консенсуса.

Вероятно, наиболее широко известным алгоритмом достижения консенсуса является алгоритм *Паксос* (Paxos). Впервые он был представлен Лесли Лэмпортом в статье *The Part-Time Parliament* («Парламент с неполной занятостью») [LAMPORT98]. В этой статье консенсус описывается в терминах, навеянных законодательным процессом и процессом голосования на эгейском острове Паксос. В 2001 году автор выпустил следующую статью под названием *Paxos Made Simple* («Простой Паксос») [LAMPORT01],

в которой была введена более простая терминология, широко используемая сейчас для объяснения этого алгоритма.

В алгоритме Паксос участники могут выступать в одной из трех ролей: *заявитель*, *акцептор* и *ученик*:

Заявители (proposers)

Получают значения от клиентов, создают предложения по принятию этих значений и пытаются собрать голоса от акцепторов.

Акцепторы (acceptors)

Голосуют за принятие или отклонение значений, предложенных заявителем. Для обеспечения отказоустойчивости алгоритм требует наличия нескольких акцепторов, но для обеспечения живучести достаточно, чтобы предложение принималось при наличии кворума (большинства) голосов акцепторов.

Ученики (learner)

Играют роль реплик, хранящих результаты принятых предложений.

Любой участник может выступать в любой роли, и большинство реализаций совмещает их: один процесс может одновременно быть заявителем, акцептором и учеником.

Каждое предложение содержит предложенное клиентом значение и уникальный монотонно увеличивающийся номер предложения. Этот номер в дальнейшем используется для обеспечения общего порядка выполняемых операций и установления между ними отношений типа «произошло до»/«произошло после». Номера предложений часто реализуются с использованием пары (идентификатор, временная метка), где идентификаторы узлов также сопоставимы и могут использоваться для разрешения неоднозначности временных меток.

Алгоритм Паксос

Алгоритм Паксос в целом можно разделить на два этапа: *голосование* (или этап *предложения*) (Voting (Propose phase)) и *репликация* (Replication). На этапе голосования заявители соревнуются в установлении своего лидерства. Во время репликации заявитель распространяет значение среди акцепторов.

Заявитель является начальной точкой контакта для клиента. Он получает значение, по которому должно быть принято решение, и пытается собрать голоса из кворума акцепторов. После этого акцепторы распространяют информацию о согласованном значении среди учеников, утверждая результат. Ученики повышают коэффициент репликации согласованного значения.

Только один заявитель может набрать большинство голосов. При некоторых обстоятельствах голоса могут равномерно распределиться между заявителями, и тогда ни один из них не сможет набрать большинство в этом раунде, поэтому им придется перезапуститься. Мы обсудим этот и другие сценарии с конкурирующими заявителями в подразделе «Сценарии отказа» на с. 308.

На этапе предложения заявитель отправляет сообщение `Подготовка(n)` (где `n` — номер предложения) большинству получателей и пытается собрать их голоса.

Когда акцептор получает запрос о подготовке, он должен ответить, сохранив следующие инварианты [LAMPORT01]:

- Если этот акцептор еще не ответил на запрос о подготовке с бóльшим порядковым номером, он *обещает*, что не примет предложение с меньшим порядковым номером.
- Если этот акцептор ранее уже принял любое другое предложение (получил сообщение `Принятие(m, v`$_{принятых}$`)`), он отвечает сообщением `Обещание(m, v`$_{принятых}$`)`, уведомляющим заявителя о том, что он уже принял предложение с порядковым номером `m`.
- Если этот акцептор уже ответил на запрос о подготовке с бóльшим порядковым номером, он уведомляет заявителя о существовании предложения с бóльшим номером.
- Акцептор может ответить более чем на один запрос о подготовке при условии, что последний запрос будет иметь наибольший порядковый номер.

На этапе репликации, собрав большинство голосов, заявитель может начать репликацию, в ходе которой он фиксирует предложение, отправляя акцепторам сообщение `Принятие(n, v)` со значением `v` и номером предложения `n`. `v` — это значение, связанное с предложением с наибольшим номером среди ответов, полученных им от акцепторов, или любое его собственное значение, если их ответы не содержали ранее принятых предложений.

Акцептор принимает предложение с номером `n`, если только на этапе предложения он уже не ответил на сообщение `Подготовка(m)`, где `m` больше, чем `n`. Если акцептор отклоняет предложение, он уведомляет об этом заявителя, посылая максимальный порядковый номер, который он видел вместе с запросом, чтобы помочь заявителю актуализировать свои данные [LAMPORT01].

Обобщенная схема раунда Паксос приведена на рис. 14.3.

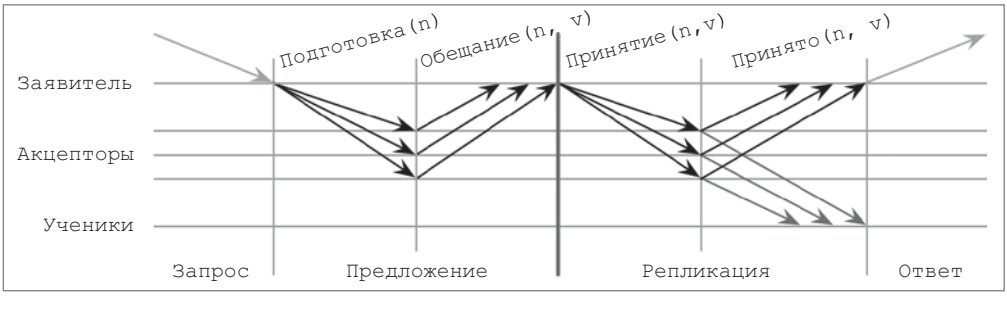

Рис. 14.3. Алгоритм Паксос: нормальное выполнение

После достижения консенсуса касательно значения (т. е. после его принятия как минимум одним акцептором) последующие заявители должны принять решение касательно того же значения, чтобы гарантировать согласованность. Именно по-

этому акцепторы отвечают последним значением, которое они приняли. Если ни один из акцепторов не видел предыдущее значение, заявителю разрешается выбрать собственное значение.

Ученик должен узнать значение, по которому было принято решение. Это значение он может узнать после получения уведомления от большинства акцепторов. Чтобы ученик узнал о новом значении как можно скорее, акцепторы могут уведомить его о значении сразу после его принятия. При наличии нескольких учеников каждый акцептор должен будет уведомить каждого ученика. Можно одного или более учеников сделать *выделенными*, в случае чего они будут уведомлять других учеников о принятых значениях.

Резюмируя, можно сказать, что цель первого этапа алгоритма состоит в том, чтобы установить лидера для раунда и понять, какое значение будет принято, позволив лидеру перейти ко второму этапу — рассылке значения. В случае базового алгоритма предполагается, что мы должны выполнять оба этапа каждый раз, когда нам нужно принять решение относительно значения. На практике хотелось бы уменьшить количество шагов в алгоритме, что можно сделать, разрешив заявителю предлагать более одного значения. Мы обсудим это подробнее позже в подразделе «Мульти-Паксос» на с. 310.

Кворумы в алгоритме Паксос

Кворумы используются, чтобы гарантировать, что при отказе некоторых участников мы все равно сможем продолжать выполнение, пока есть возможность собирать голоса от активных участников. *Кворум* — это *минимальное* количество голосов, необходимое для выполнения операции. Это число обычно равняется *большинству* участников. Основная идея кворумов состоит в том, что, даже если участники выходят из строя или отделяются из-за распада сети, есть по крайней мере один участник, который выступает в качестве арбитра, обеспечивающего корректность протокола.

Если достаточное количество участников примет предложение, это значение будет гарантированно принято протоколом, поскольку любые два большинства имеют как минимум одного общего участника.

Алгоритм Паксос гарантирует безопасность при наличии любого количества отказов. Не существует конфигурации, способной привести к некорректным или несогласованным состояниям, поскольку это противоречило бы определению консенсуса.

Важно помнить, что кворумы описывают только блокирующие свойства системы. Чтобы гарантировать безопасность, на каждом шаге необходимо дождаться ответа от *как минимум* кворума узлов. Мы можем отправлять предложения и принимать команды от большего количество узлов; нам просто не нужно ждать их ответа для того, чтобы продолжить выполнение. Мы можем отправлять сообщения большему количеству узлов (некоторые системы используют *спекулятивное выполнение*, выдавая избыточные запросы, помогающие обеспечить требуемое количество ответов в случае отказа узлов), но для обеспечения живучести нам достаточно получить информацию от кворума, чтобы продолжить выполнение.

Живучесть гарантируется при наличии `f` отказов процессов. Для этого протоколу требуется в общей сложности `2f + 1` процессов, чтобы в случае отказа `f` процессов оставалось `f + 1` процессов, способных продолжать выполнение. Используя кворумы вместо того, чтобы требовать присутствия всех процессов, Паксос (как и другие алгоритмы достижения консенсуса) гарантирует результаты даже в случае отказа `f` процессов. В подразделе «Гибкий Паксос» на с. 316 мы поговорим о кворумах в несколько иных терминах и узнаем, как можно создавать протоколы, требующие пересечения кворумов только в рамках *шагов* алгоритма.

Сценарии отказа

Обсуждение распределенных алгоритмов становится особенно интересным, когда дело доходит до отказов. Один из сценариев отказа, демонстрирующий отказоустойчивость, сводится к следующему: заявитель дает сбой во время второго этапа, еще не успев передать значение всем акцепторам (такая ситуация может возникнуть, если заявитель активен, но работает медленно или не может связаться с некоторыми акцепторами). В этом случае новый заявитель может подхватить и зафиксировать значение, распространив его на других участников.

На рис. 14.4 показана такая ситуация:

- Заявитель P_1 проходит этап выбора с номером предложения `1`, но выходит из строя после отправки значения `V1` только одному акцептору A_1.
- Другой заявитель P_2 начинает новый раунд с предложением с бóльшим номером — `2`, собирает ответы от кворума акцепторов (в данном случае A_1 и A_2) и продолжает выполнение, зафиксировав *старое* значение `V1`, предложенное заявителем P_1.

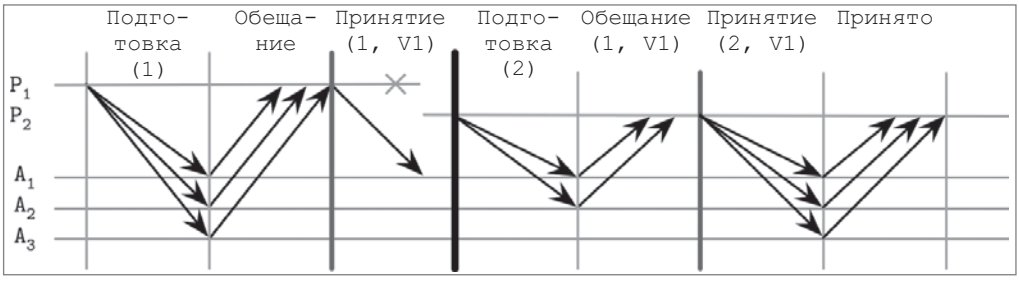

Рис. 14.4. Сценарий отказа при выполнении алгоритма Паксос: ошибка заявителя, выбирающего старое значение

Поскольку состояние алгоритма реплицируется на несколько узлов, отказ заявителя не приводит к невозможности достижения консенсуса. Если отказ текущего заявителя происходит после того, как хотя бы один акцептор A_1 принял значение, то его предложение может быть подхвачено следующим заявителем. Это также подразумевает, что все это может произойти без ведома первоначального заявителя об этом.

В клиент-серверном приложении, где клиент подключен только к первоначальному заявителю, это может привести к тому, что клиент не будет знать о результате выполнения раунда Паксос[1].

Однако возможны и другие сценарии, как показано на рис. 14.5. Например:

- Заявитель P_1 дает сбой так же, как в предыдущем примере, после отправки значения V1 только акцептору A_1.
- Следующий заявитель P_2 начинает новый раунд с предложением с большим номером 2 и собирает ответы от кворума акцепторов, но на этот раз первыми отвечают акцепторы A_2 и A_3. После сбора кворума заявитель P_2 фиксирует *свое собственное значение*, несмотря на то что теоретически акцептор A_1 может содержать другое зафиксированное значение.

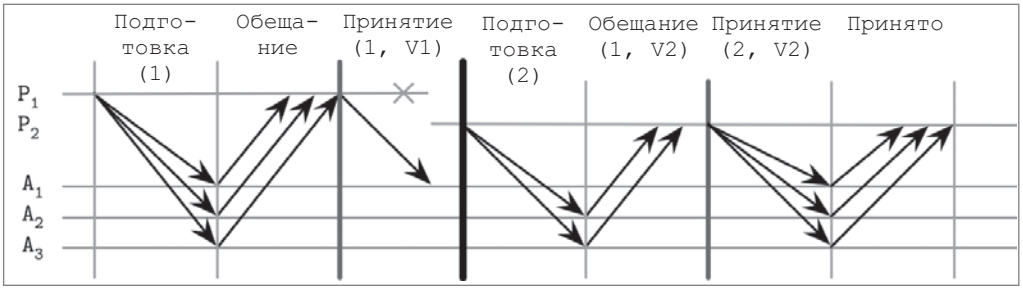

Рис. 14.5. Сценарий отказа при выполнении алгоритма Паксос: отказ заявителя, выбор нового значения

Здесь есть еще одна возможность, показанная на рис. 14.6:

- Заявитель P_1 дает сбой после того, как только один акцептор A_1 принимает значение V1. Акцептор A_1 дает сбой вскоре после принятия предложения, не успев уведомить следующего заявителя о своем значении.
- Заявитель P_2, который начал раунд после отказа заявителя P_1, не охватывает акцептора A_1 и вместо этого фиксирует собственное значение.
- Любой заявитель, который появится после этого раунда и будет охватывать акцептора A_1, проигнорирует значение акцептора A_1 и выберет вместо этого более свежее принятое предложение.

Еще один сценарий отказа сводится к следующему: два или более заявителей начинают соревноваться, пытаясь пройти этап предложения, но раз за разом не могут собрать большинство из-за того, что другой заявитель успевает получить эти голоса быстрее.

Хотя акцепторы обещают не принимать предложений с меньшим номером, они все же могут отвечать на несколько запросов на подготовку, если последний из них имеет

[1] Например, такая ситуация была описана в статье, приведенной по адресу https://databass.dev/links/68.

больший порядковый номер. Когда заявитель пытается зафиксировать значение, он может обнаружить, что акцепторы уже ответили на запрос на подготовку с бóльшим порядковым номером. Это может привести к тому, что несколько заявителей будут постоянно повторять попытки и препятствовать дальнейшему продвижению друг друга. Эта проблема обычно решается путем введения произвольно выбранной отсрочки, которая в конечном итоге позволяет одному из заявителей продолжить работу, пока другие находится в режиме сна.

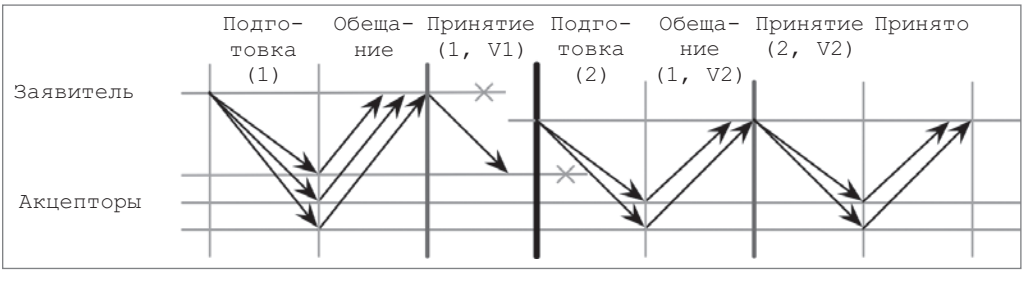

Рис. 14.6. Сценарий отказа при выполнении алгоритма Паксос: отказ заявителя, за которым следует отказ акцептора

Алгоритм Паксос допускает отказы акцепторов, но только в том случае, если оставшихся акцепторов все еще достаточно для того, чтобы сформировать большинство.

Мульти-Паксос

До сих пор мы обсуждали классический алгоритм Паксос, где мы выбираем произвольного заявителя и пытаемся начать раунд Паксос. Одна из проблем такого подхода заключается в том, что для каждого раунда репликации, который происходит в системе, требуется раунд с предложением. Заявитель может начать репликацию только после утверждения свой кандидатуры для раунда, а для этого необходимо, чтобы большинство акцепторов ответило сообщением `Обещание` на сообщение заявителя `Подготовка`. Чтобы избежать повторения этапа предложения и позволить заявителю повторно использовать свое, уже признанное положение, мы можем использовать алгоритм мульти-Паксос, который вводит понятие *лидера* — некоторого *выделенного заявителя* [LAMPORT01]. Это важное дополнение, способное существенно повысить эффективность алгоритма.

При наличии некоторого заданного лидера мы можем пропустить этап предложения и перейти непосредственно к репликации: распространению значения и сбору подтверждений от акцепторов.

В классическом алгоритме чтение можно реализовать путем запуска раунда Паксос, который будет собирать любые значения из незавершенных раундов, если таковые имеются. Это необходимо потому, что нет гарантии того, что последний известный заявитель хранит самые последние данные, поскольку другой заявитель мог изменить состояние так, что последний заявитель не узнал об этом.

Аналогичная ситуация возможна и в случае алгоритма мульти-Паксос: мы пытаемся выполнить чтение из известного лидера в момент, когда уже выбран другой лидер, получая при этом устаревшие данные, что противоречит гарантиям линеаризуемости, предоставляемым консенсусом. Чтобы избежать этого и гарантировать, что другие процессы не смогут успешно передать значения, некоторые реализации алгоритма мульти-Паксос используют *аренду* (lease). Лидер периодически связывается с участниками, уведомляя их о том, что он все еще активен, и продлевая тем самым свой срок аренды. Участники должны ответить и позволить лидеру продолжить работу, пообещав, что они не примут предложения от других лидеров в течение срока аренды [CHANDRA07].

Аренда — это не гарантия корректности, а оптимизация производительности, которая позволяет выполнять чтение из активного лидера без сбора кворума. Чтобы гарантировать безопасность, механизм аренды использует ограниченную синхронизацию часов между участниками. В случае слишком большого ухода показаний часов, когда лидер считает, что его срок аренды еще не истек, а другие участники — наоборот, *невозможно* гарантировать линеаризуемость.

Алгоритм мульти-Паксос иногда описывается как *реплицируемый журнал* операций, применяемых к некоторой структуре. При этом не играет роли семантика этой структуры, важно лишь обеспечить согласованную репликацию значений, добавляемых в этот журнал. Чтобы сохранить состояние в случае аварийного завершения работы, участники ведут долговременный журнал полученных сообщений.

Чтобы избежать бесконечного роста журнала, его содержимое должно применяться к вышеупомянутой структуре. После синхронизации содержимого журнала с первичной структурой (с созданием моментального снимка) журнал можно усечь. Снимки журнала и состояния должны быть взаимно согласованными, а изменения снимка должны применяться атомарно с усечением сегмента журнала [CHANDRA07].

Алгоритм Паксос с одним решением можно рассматривать как *регистр с однократной записью*: у нас есть ячейка, куда можно поместить значение, дальнейшие модификации которого после записи невозможны. На первом этапе заявители соревнуются за право владения регистром, а на втором этапе один из них записывает значение. В то же время алгоритм мульти-Паксос можно рассматривать как журнал с доступом только для добавления, в котором содержится последовательность таких значений: мы можем записывать одно значение за раз, все значения строго упорядочены и нельзя изменить уже записанные значения [RYSTSOV16]. Существуют примеры алгоритмов достижения консенсуса, такие как Active Disk Paxos [CHOCKLER15] и CASPaxos [RYSTSOV18], которые предлагают наборы регистров чтения-модификации-записи и применяют совместное использование состояний, а не реплицируемые конечные автоматы.

Быстрый Паксос

Мы можем уменьшить количество циклов «запрос–подтверждение» на один по сравнению с классическим алгоритмом Паксос, позволив *любому* заявителю связываться

с акцепторами напрямую, а не через лидера. Для этого нужно сделать размер кворума равным `2f + 1` (где `f` — допустимое количество сбойных процессов), а не `f + 1`, как в классическом алгоритме Паксос, а общее количество акцепторов — равным `3f + 1` [JUNQUEIRA07]. Эта оптимизация получила название *быстрый Паксос* [LAMPORT06].

Классический алгоритм Паксос требует, чтобы во время этапа репликации заявитель мог выбрать любое значение из тех, которые он собрал во время этапа предложения. Быстрый Паксос предусматривает два типа раундов: *классический*, при котором алгоритм работает так же, как классическая версия алгоритма, и *быстрый*, при котором акцепторам разрешено принимать другие значения.

При описании данного алгоритма мы будем называть *координатором* того заявителя, который собрал достаточное количество ответов в течение этапа предложения, а *заявителем* — всех остальных заявителей. Иногда при описании быстрого Паксоса говорят, что напрямую связываться с акцепторами могут *клиенты* [ZHAO15].

В ходе быстрого раунда, если координатору разрешено выбирать собственное значение на этапе репликации, он может вместо этого выдать акцепторам специальное сообщение `Любое`. В этом случае акцепторам разрешается обрабатывать *любое* значение заявителя, как если бы они получили сообщение с этим значением от координатора в ходе классического раунда. То есть акцепторы самостоятельно принимают решения относительно значений, полученных ими от разных заявителей.

На рис. 14.7 показаны примеры классического и быстрого раундов алгоритма Быстрый Паксос. Из рисунка может показаться, что в быстром раунде больше шагов выполнения, но нужно иметь в виду, что в классическом раунде, чтобы предоставить свое значение, заявителю потребуется пройти через координатора для фиксации своего значения.

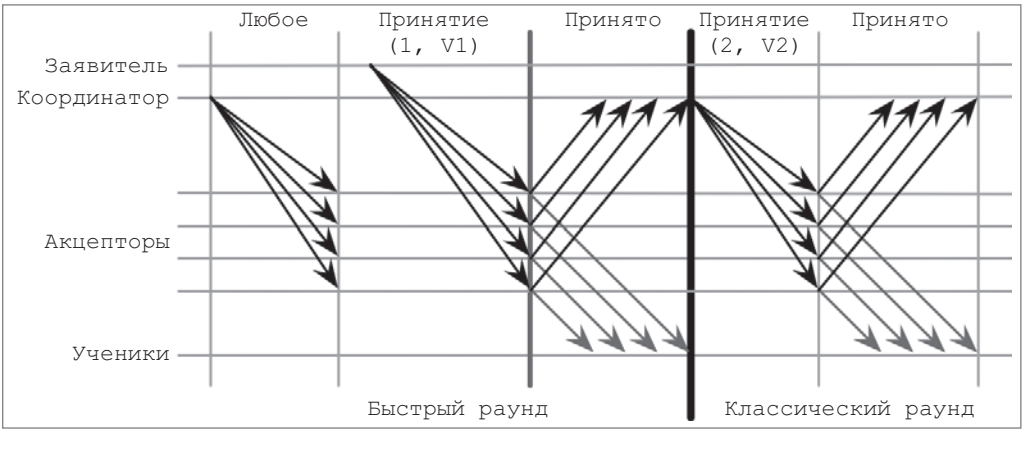

Рис. 14.7. Алгоритм быстрый Паксос: быстрые и классические раунды

Данный алгоритм подвержен *коллизиям*, возникающим, если два или более заявителей пытаются использовать *быстрый* раунд, сократив количество циклов «запрос–подтверждение», и акцепторы получают разные значения. В таком случае координатор должен вмешаться и запустить восстановление, инициировав новый раунд.

Это означает, что акцепторы после получения значений от разных заявителей могут принять решение о принятии конфликтующих значений. Когда координатор обнаруживает конфликт (коллизию значений), он должен заново инициировать этап предложения, чтобы позволить акцепторам прийти к одному значению.

Одним из недостатков быстрого Паксоса является большое количество циклов «запрос–подтверждение» и задержки запросов из-за коллизий при высокой частоте запросов. В источнике [JUNQUEIRA07] показано, что из-за большего количества реплик и как следствие — более интенсивного обмена *сообщениями* между участниками, несмотря на меньшее количество шагов, алгоритм Быстрый Паксос может приводить к более высоким задержкам по сравнению с классической версией алгоритма.

Эгалитарный Паксос

При использовании выделенного заявителя в качестве лидера система становится подверженной отказам: в случае отказа лидера система должна выбрать нового лидера для того, чтобы продолжить выполнение дальнейших действий. Другая проблема заключается в том, что на лидера может лечь непропорциональная нагрузка, следствием чего станет снижение производительности системы.

Один из способов избежать возложения на лидера всей нагрузки системы сводится к использованию *секционирования*. Многие системы разбивают диапазон возможных значений на более мелкие сегменты, позволяя отдельным частям системы отвечать за определенную часть диапазона, не беспокоясь о других его частях. Такой подход повышает доступность (за счет изолирования отказов в одной секции без распространения их на другие части системы), производительность (поскольку сегменты, обслуживающие разные значения, не перекрываются) и масштабируемость (поскольку мы можем масштабировать систему, увеличивая количество секций). Важно помнить, что для выполнения операций над несколькими секциями требуется атомарная фиксация.

Вместо использования лидера и номера предложения для построения последовательности команд мы можем использовать лидера, отвечающего за фиксацию *конкретной* команды, и установить порядок путем просмотра и установки зависимостей. Такой подход обычно называют *эгалитарным Паксосом* или *ЭПаксосом* (egalitarian Paxos, EPaxos) [MORARU11]. Идея о том, что неконфликтующие операции записи можно независимо фиксировать в реплицируемом конечном автомате, была впервые представлена в источнике [LAMPORT05] и получила название «обобщенный Паксос». ЭПаксос является первой реализацией обобщенного Паксоса.

ЭПаксос является попыткой объединить преимущества классической версии Паксоса и мульти-Паксоса. Классический Паксос обеспечивает высокую доступность, поскольку лидер устанавливается в каждом раунде, но имеет более сложный обмен сообщениями. Мульти-Паксос обеспечивает высокую пропускную способность и требует меньше сообщений; но здесь узким местом может стать лидер.

ЭПаксос начинает работу с этапа *предварительного принятия*, в ходе которого некоторый процесс становится лидером для конкретного предложения. Каждое предложение должно включать в себя следующее:

Зависимости

Все команды, которые могут помешать текущему предложению, но, возможно, еще не зафиксированы.

Порядковый номер

Разрывает циклы между зависимостями. Должен быть больше, чем любой порядковый номер известных зависимостей.

После сбора этой информации данный алгоритм пересылает сообщение `Предварительное принятие` *быстрому кворуму* реплик. Быстрый кворум включает в себя `(f/4)` реплик, где `f` — число допустимых отказов.

Реплики проверяют свои локальные журналы команд, обновляют зависимости предложений на основе своих представлений о потенциально конфликтующих предложениях и отправляют эту информацию обратно лидеру. Если лидер получает ответы от «быстрого» кворума реплик и их списки зависимостей согласуются друг с другом и с самим лидером, он может зафиксировать команду.

Если лидер не получает достаточного количества ответов или полученные от реплик списки команд различаются и содержат конфликтующие команды, он обновляет свое предложение новым списком зависимостей и новым порядковым номером. Новый список зависимостей основан на предыдущих ответах реплик и объединяет *все* собранные зависимости. Новый порядковый номер должен быть больше самого большего порядкового номера, видимого репликами. После этого лидер отправляет новую обновленную команду `(f/2) + 1` репликам. После этого лидер может наконец зафиксировать предложение.

Фактически у нас есть два возможных сценария:

Быстрый путь

Когда зависимости совпадают и лидер может безопасно перейти к этапу фиксации, используя только *быстрый кворум* реплик.

Медленный путь

При наличии разногласий между репликами необходимо обновить их списки команд для того, чтобы лидер мог перейти к фиксации.

Эти сценарии показаны на рис. 14.8, где заявитель P_1 инициирует прохождение по быстрому пути, а заявитель P_5 — прохождение по медленному пути:

- Заявитель P_1 начинает работу с предложения с номером 1 и без зависимостей и отправляет сообщение Предварительное_принятие(1, ∅). Поскольку журналы команд заявителей P_2 и P_3 пусты, заявитель P_1 может перейти к фиксации.
- Заявитель P_5 создает предложение с порядковым номером 2. Поскольку его журнал команд к этому моменту пуст, он также не объявляет зависимости и отправляет сообщение Предварительное_принятие(2, ∅). Заявитель P_4 не знает о зафиксированном предложении 1, но заявитель P_3 уведомляет заявителя P_5 о конфликте и отправляет свой журнал команд: {1}.
- Заявитель P_5 обновляет свой локальный список зависимостей и отправляет сообщение с тем, чтобы обеспечить наличие у реплик одинаковых зависимостей: Принятие(2, {1}). После того как реплики дают ответ, он может зафиксировать значение.

Две команды, A и B, конфликтуют, только если их порядок выполнения имеет значение, другими словами, если выполнение команды A перед командой B и выполнение команды B перед командой A ведет к разным результатам.

Фиксация выполняется путем предоставления ответа клиенту и асинхронного уведомления реплик сообщением Фиксация. Команды выполняются после их фиксации.

Поскольку зависимости собираются во время этапа предварительного принятия, к моменту выполнения запросов порядок команд уже установлен и ни одна команда не может внезапно появиться в каком-либо промежутке: команда может быть только добавлена после команды с наибольшим порядковым номером.

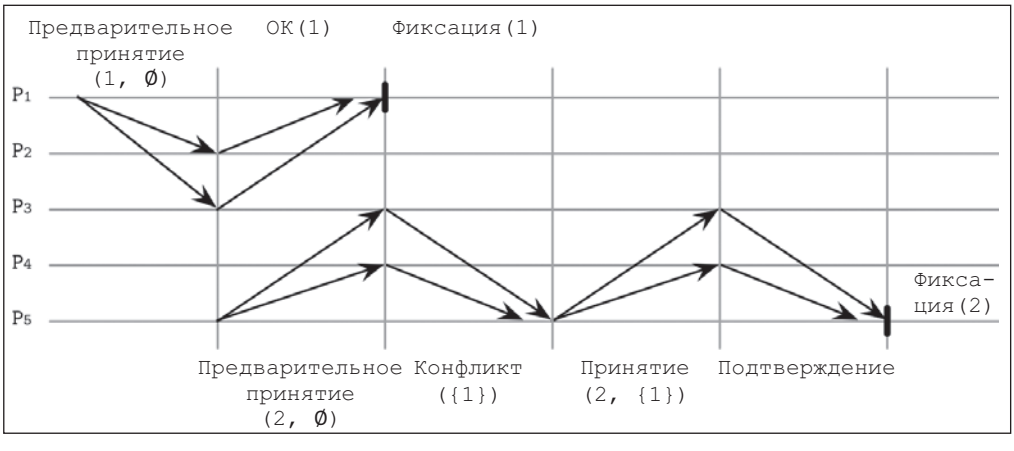

Рис. 14.8. Работа алгоритма ЭПаксос

Чтобы выполнить команду, реплики строят граф зависимостей и выполняют все команды в порядке, обратном порядку зависимостей. То есть, для того чтобы выполнить команду, сначала нужно выполнить все ее зависимости (а следовательно,

и все их зависимости). Поскольку зависеть друг от друга будут только конфликтующие команды, такая ситуация должна быть относительно редкой для большинства рабочих нагрузок [MORARU13].

Как и классический Паксос, ЭПаксос использует номера предложений, что исключает возможность распространения устаревших сообщений. Порядковые номера включают в себя *эпоху* (идентификатор текущей конфигурации кластера, который изменяется, когда узлы покидают кластер или присоединяются к нему), монотонно увеличивающийся локальный счетчик узла и идентификатор реплики. Если реплика получает предложение, порядковый номер которого меньше уже виденного ею номера, то она выдает отрицательное подтверждение, отправляя в ответ самый большой порядковый номер и имеющийся у нее обновленный список команд.

Гибкий Паксос

Кворум обычно определяется как большинство процессов. По определению у нас есть *пересечение* между двумя кворумами независимо от того, как мы выбираем узлы: всегда есть хотя бы один узел, позволяющий разорвать связи.

Нам необходимо ответить на два важных вопроса:

- Нужно ли связываться с *большинством* серверов на *каждом* шаге выполнения?
- *Все* ли кворумы должны пересекаться? То есть должны ли быть общие узлы у кворума, который мы используем для выбора выделенного заявителя (первый этап), у кворума, который мы используем для принятия решения относительно значения (второй этап) и у каждого экземпляра выполнения (например, если несколько экземпляров второго этапа выполняются параллельно)?

Поскольку мы все еще говорим о консенсусе, мы не можем изменять определения безопасности: алгоритм должен гарантировать согласованность.

В алгоритме мульти-Паксос этап выбора лидера выполняется нечасто, и выделенному заявителю разрешается фиксировать несколько значений без повторения этапа выборов, потенциально оставаясь лидером в течение более длительного периода. В разделе «Настраиваемая согласованность» на с. 253 мы обсуждали формулы, позволяющие находить конфигурации, обеспечивающие наличие пересечений между наборами узлов. Одним из примеров является подход, когда мы ожидаем подтверждение операции записи только от одного узла (и позволяем запросам к остальным узлам завершиться асинхронно) и производим чтение из *всех* узлов. То есть если справедливо неравенство `R + W > N`, наборы чтения и записи имеют как минимум один общий узел.

Можем ли мы использовать аналогичную логику для достижения консенсуса? Как оказывается, можем, и в случае алгоритма Паксос нам достаточно, чтобы группа узлов из первого этапа (которая выбирает лидера) перекрывалась с группой из второго этапа (которая участвует в принятии предложений).

Другими словами, кворум может определяться не как большинство, а просто как непустая группа узлов. Если мы определим общее число участников как `N`, количество узлов,

необходимое для успешного завершения этапа предложения как Q_1, а количество узлов, необходимое для успешного завершения этапа принятия как Q_2, то нам нужно будет лишь убедиться, что $Q_1 + Q_2 > N$. Поскольку второй этап обычно выполняется чаще, чем первый, множество Q_2 может содержать только `N / 2` акцепторов, если множество Q_1, соответственно, будет настроено на большее значение ($Q_1 = N - Q_2 + 1$). Этот вывод является важным наблюдением, необходимым для понимания консенсуса. Алгоритм на основе этого подхода называется «Гибкий Паксос» (flexible Paxos) [HOWARD16].

Например, если у нас есть пять акцепторов, то, поскольку, чтобы выиграть раунд выборов, нам нужно собрать голоса от четырех из них, на этапе репликации мы можем позволить лидеру ждать ответов от двух узлов. Более того, поскольку существует перекрытие между любым подмножеством из двух акцепторов и кворумом для выбора лидера, мы можем отправлять предложения непересекающимся наборам акцепторов. Вполне очевидно, что это работает, поскольку всякий раз, когда новый лидер избирается без ведома текущего, всегда найдется хотя бы один акцептор, который знает о существовании нового лидера.

Гибкий Паксос позволяет пожертвовать доступностью ради уменьшения задержки: мы уменьшаем количество узлов, участвующих во втором этапе, но должны при этом собирать больше голосов, что, в свою очередь, требует, чтобы нам было доступно больше участников на этапе выбора лидера. Хорошая новость заключается в том, что такая конфигурация может продолжать этап репликации и выдержать отказ до $N - Q_2$ узлов, если текущий лидер стабилен и новый раунд выбора не требуется.

Еще одной разновидностью алгоритма Паксос, использующей идею пересекающихся кворумов, является вертикальный Паксос (vertical Paxos). Вертикальный Паксос проводит различие между кворумами чтения и записи. Эти кворумы должны пересекаться. Лидер должен собрать меньший кворум чтения для одного или нескольких предложений с меньшим номером и больший кворум записи для своего собственного предложения [LAMPORT09]. В источнике [LAMPSON01] также проводится различие между *исходящим кворумом* (out quorum) и *кворумом решения* (decision quorum), используемыми для этапов подготовки и принятия; при этом определение «кворума» звучит так же, как в случае с гибким Паксосом.

Обобщенное решение для достижения консенсуса

Рассмотрение алгоритма Паксос порой вызывает трудности: достаточно трудно разобраться со всем этим множеством ролей, шагов и возможных вариантов алгоритма. Однако этот алгоритм также можно рассматривать, используя более простую терминологию. Вместо того чтобы распределять роли между участниками и проводить раунды принятия решений, мы можем использовать простой набор концепций и правил, позволяющий обеспечить гарантии алгоритма Паксос с одним решением. Мы обсудим этот подход лишь в общих чертах, поскольку это сравнительно новая разработка [HOWARD19], — важно знать о его существовании, но примеры его реализации и практического применения еще должны появиться.

Здесь у нас имеется клиент и набор серверов. Каждый сервер имеет несколько *регистров*. Регистр имеет идентифицирующий его индекс, предназначен только для однократной записи и может находиться в одном из трех состояний: не записан, содержит значение и содержит нуль (специальное пустое значение).

Регистры с одинаковым индексом, расположенные на разных серверах, образуют *набор регистров*. Каждый набор регистров может иметь один или несколько кворумов. В зависимости от состояния регистров в кворуме кворум может находиться в одном из *неопределенных* (`Любое` и `Возможно v`) или *определенных* (`Ни одно` и `Принято v`) состояний:

`Любое`

: В зависимости от последующих операций данный набор кворума может выбрать любое значение.

`Возможно v`

: Данный кворум может выбрать только значение `v`.

`Ни одно`

: Этот кворум не может выбрать значение.

`Принято v`

: Этот кворум выбрал значение `v`.

Клиент обменивается сообщениями с серверами и поддерживает таблицу состояний, в которой отслеживаются значения и регистры, и может выяснять принятое кворумом решение.

Чтобы поддерживать корректность, мы должны ограничить способы взаимодействия клиента с серверами и допустимые для записи значения. Что касается чтения значений, клиент может вывести выбранное значение, только если он прочитал его из кворума серверов в том же наборе регистров.

Правила записи чуть более сложны, потому что для обеспечения безопасности алгоритма мы должны поддерживать несколько инвариантов. Во-первых, мы должны гарантировать, что клиент не будет выбирать новые значения: ему разрешается записать определенное значение в регистр, только если он получил его в качестве входных данных или прочитал его из регистра. Клиенты не могут записывать значения, которые позволяют другим кворумам в том же регистре выбирать другие значения. Наконец, клиенты не могут записывать значения, которые отменяют предыдущие решения, сделанные в предыдущих наборах регистров (в наборах регистров до `r – 1` должны быть приняты решения `Ни одно`, `Возможно v` или `Принято v`).

Обобщенный алгоритм Паксос

Сведя эти правила воедино, мы можем реализовать обобщенный алгоритм Паксос, который достигает консенсуса в отношении одного значения с помощью регистров с однократной записью [HOWARD19]. Допустим, у нас есть три сервера `[S0, S1, S2]`, регистры `[R0, R1,...]` и клиенты `[C0, C1, ...]`; при этом клиент может произво-

дить запись только в назначенное подмножество регистров. Для всех регистров мы используем кворумы простого большинства (`{S0, S1}, {S0, S2}, {S1, S2}`).

Процесс принятия решения здесь состоит из двух этапов. Первый этап обеспечивает безопасность записи значения в регистр, а на втором этапе значение записывается в регистр:

В ходе этапа 1

Клиент проверяет, не записан ли регистр, который он собирается записать, отправляя команду $P1_A$(`регистр`) на сервер. Если регистр не записан, то все регистры до (`регистр - 1`) устанавливаются в nil — после чего другие клиенты больше не смогут производить запись в предыдущие регистры. В качестве ответа сервер выдает набор регистров, записанных на данный момент. Если клиент получает ответы от большинства серверов, то он выбирает либо непустое значение из регистра с наибольшим индексом, либо свое собственное значение, если значения отсутствуют. В противном случае он перезапускает первый этап.

В ходе этапа 2

Клиент уведомляет все серверы о значении, выбранном на первом этапе, отправляя им команду $P2_A$(`регистр, значение`). Если большинство серверов отвечает на это сообщение, он может вывести принятое значение. В противном случае он снова запускает этап 1.

Эта обобщенная версия алгоритма Паксос показана на рис. 14.9 (адаптировано из [HOWARD19]). Клиент `C0` пытается зафиксировать значение `V`. На первом шаге его таблица состояний пуста, и серверы `S0` и `S1` выдают в качестве ответа пустой набор регистров, тем самым указывая, что регистры пока не записаны. В ходе второго шага клиент может передать свое значение `V`, так как не было записано ни одно другое значение.

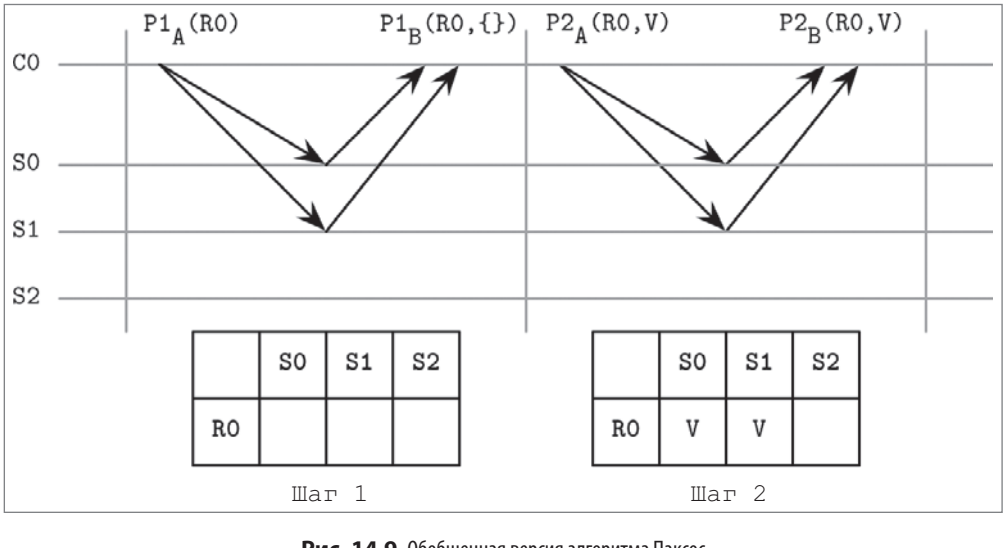

Рис. 14.9. Обобщенная версия алгоритма Паксос

В этот момент любой другой клиент может отправить серверам запрос, чтобы узнать текущее состояние. Кворум {S0, S1} достиг состояния Принято A, а кворумы {S0, S2} и {S1, S2} достигли состояния Возможно V для регистра R0, поэтому клиент C1 выбирает значение V. После этого ни один клиент не может выбрать какое-либо другое значение.

Такой подход помогает понять семантику алгоритма Паксос. Вместо того чтобы рассматривать состояние с точки зрения взаимодействия удаленных участников (например: заявитель выясняет, принял ли уже акцептор другое предложение), мы можем рассматривать его с точки зрения последнего известного состояния, делая наш процесс принятия решений простым и устраняя возможные неясности. При этом также проще корректно реализовать неизменяемое состояние и передачу сообщений.

Мы также можем провести параллели с исходной версией алгоритма Паксос. Например, сценарий, когда клиент обнаруживает, что для одного из предыдущих наборов регистров было принято решение Возможно V, после чего он подхватывает значение V и пытается зафиксировать его снова, аналогичен случаю, когда заявитель в алгоритме Паксос может предложить значение после отказа предыдущего заявителя, которому удалось зафиксировать это значение хотя бы в одном акцепторе. Точно так же, если в Паксос конфликты лидеров разрешаются путем перезапуска голосования с более высоким номером предложения, в обобщенном алгоритме любые незаписанные регистры с более низким рангом устанавливаются в нуль.

Raft

Алгоритм Паксос использовался в качестве алгоритма достижения консенсуса в течение более десяти лет, но считался при этом достаточно сложным для рассмотрения в сообществе разработчиков распределенных систем. В 2013 году появился новый алгоритм под названием Raft. Его создатели хотели создать алгоритм, который легко поддавался бы пониманию и реализации. Впервые данный алгоритм был представлен в статье под названием *In Search of an Understandable Consensus Algorithm* («В поисках понятного алгоритма достижения консенсуса») [ONGARO14].

Распределенные системы уже сами по себе являются достаточно сложными, что делает очень желательным наличие более простых алгоритмов. Наряду со статьей авторы выпустили эталонную реализацию LogCabin, с тем чтобы устранить возможные неоднозначности и помочь будущим разработчикам в понимании алгоритма.

В данном случае участники хранят локально журнал, содержащий последовательность команд, выполняемых конечным автоматом. Поскольку входные данные, которые получают процессы, идентичны, а журналы содержат одинаковые команды в одинаковом порядке, применение этих команд к конечному автомату гарантирует одинаковый результат. Raft упрощает алгоритм достижения консенсуса, делая сущность лидера объектом первого класса. Лидер используется для координации репликации и манипуляций конечного автомата. Алгоритм Raft имеет много обще-

го с алгоритмом атомарной рассылки, а также алгоритмом мульти-Паксос: среди реплик выявляется один лидер, он принимает атомарные решения и устанавливает порядок сообщений.

Каждый участник алгоритма Raft может выступать в одной из трех ролей:

Кандидат (candidate)

Лидерство — временное состояние, и любой участник может взять на себя эту роль. Чтобы стать лидером, узел должен сначала перейти в состояние кандидата и попытаться набрать большинство голосов. Если кандидат и не побеждает, и не проигрывает при выборе (голоса распределяются между несколькими кандидатами, и ни один из них не получает большинства голосов), назначается новый период и выбор перезапускается.

Лидер (leader)

Текущий временный лидер кластера, который обрабатывает клиентские запросы и взаимодействует с реплицируемым конечным автоматом. Лидер избирается на промежуток времени, называемый *периодом*. Каждый период идентифицируется монотонно возрастающим номером и может продолжаться неопределенно долго. Новый лидер избирается, если текущий лидер дает сбой, перестает отвечать на запросы или подозревается другими процессами в наличии отказа, что может произойти из-за распада сети или задержки сообщений.

Последователь (follower)

Пассивный участник, который персистентно сохраняет записи журнала и отвечает на запросы от лидера и кандидатов. Последователь в Raft сочетает в себе роли акцептора и ученика из алгоритма Паксос. Каждый процесс начинает свое выполнение как последователь.

Чтобы гарантировать глобальное частичное упорядочение, не полагаясь на синхронизацию часов, время делится на *периоды* (также называемые эпохами), в течение которых лидер уникален и неизменен. Периоды имеют монотонную нумерацию, а каждая команда уникально идентифицируется номером периода и номером сообщения в пределах периода [HOWARD14].

Участники могут иметь разное представление о том, какой период является *текущим*, поскольку они могут узнать о новом периоде в разное время или пропустить этап выбора лидера для одного или несколько периодов. Поскольку каждое сообщение содержит идентификатор периода, если один из участников обнаруживает, что его период является устаревшим, он обновляет период до самого высокого номера [ONGARO14]. Это означает, что в любой момент времени могут быть задействованными несколько периодов, но в случае конфликта побеждает период с самым большим номером. Узел обновляет период, только если он начинает новый процесс выбора или обнаруживает, что его период устарел.

Последователь запускает процесс выбора лидера при запуске или в том случае, когда он не получает сообщения от лидера, что вызывает у него подозрения в том, что

лидер дал сбой. Участник пытается стать лидером, переходя в состояние кандидата и собирая голоса от большинства узлов.

На рис. 14.10 представлена схема последовательности действий, показывающая основные компоненты алгоритма Raft:

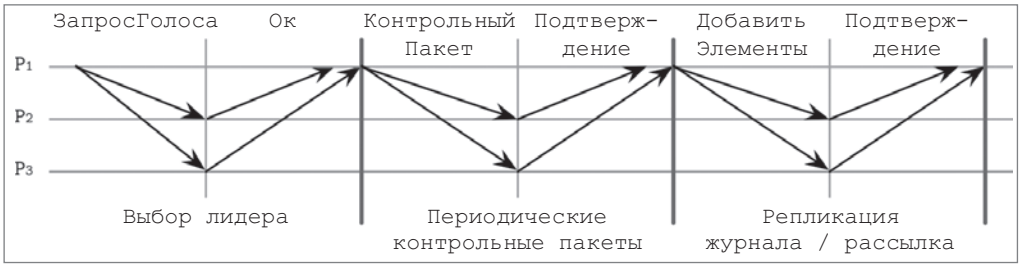

Рис. 14.10. Общая схема алгоритма достижения консенсуса Raft

Выбор лидера

Кандидат P_1 отправляет сообщение `ЗапросГолоса` другим процессам. Это сообщение включает в себя период кандидата, последний известный ему период и идентификатор последней записи в журнале, которую он видел. Собрав большинство голосов, кандидат успешно избирается лидером на период. Каждый процесс может отдать свой голос только одному кандидату.

Периодические контрольные пакеты

Данный протокол использует механизм контрольных пакетов, чтобы гарантировать жизнеспособность участников. Лидер периодически посылает контрольные пакеты всем последователям, чтобы сохранить свой период. Если последователь не получает новые контрольные пакеты в течение *«срока ожидания выбора»,* то он делает вывод о том, что произошел отказ лидера, и начинает новый процесс выбора.

Репликация / рассылка журнала

Лидер может многократно добавлять новые значения в реплицируемый журнал, отправляя сообщения `ДобавитьЭлементы`. Это сообщение включает в себя период лидера, индекс и период записи журнала, непосредственно предшествующей тем записям, которые он отправляет в данный момент, а также *одну или несколько* сохраняемых записей.

Роль лидера в Raft

Лидер может быть выбран только из узлов, содержащих все зафиксированные записи: если во время выбора информация журнала последователя более актуальна (другими словами, имеет больший идентификатор периода или более длинную последовательность записей журнала, если периоды равны), чем у кандидата, то голосование по его кандидатуре отменяется.

Чтобы выиграть голосование, кандидат должен набрать большинство голосов. Записи всегда реплицируются по порядку, поэтому достаточно сравнить идентификаторы последних записей, чтобы понять, содержит ли участник актуальные данные.

После избрания лидер должен принимать запросы клиентов (которые также могут быть перенаправлены на него с других узлов) и реплицировать их последователям. Это делается путем добавления записи в его журнал и параллельной ее отправки всем последователям.

Когда последователь получает сообщение `ДобавитьЭлементы`, он добавляет записи из сообщения в локальный журнал и подтверждает сообщение, подтверждая тем самым лидеру, что оно сохранено. Как только достаточное количество реплик отправляет свои подтверждения, запись считается зафиксированной и помечается соответственным образом в журнале лидера.

Поскольку лидером могут стать только кандидаты с самыми актуальными данными, последователям не нужно отправлять актуальные данные для обновления лидеру, а записи журнала передаются только от лидера к последователю, а не наоборот.

На рис. 14.11 показан этот процесс:

а) Новая команда `x = 8` добавляется в журнал лидера.

б) Перед тем как зафиксировать значение, его необходимо реплицировать на большинство участников.

в) После того как лидер завершает репликацию, он фиксирует значение локально.

г) Решение о фиксации реплицируется на последователей.

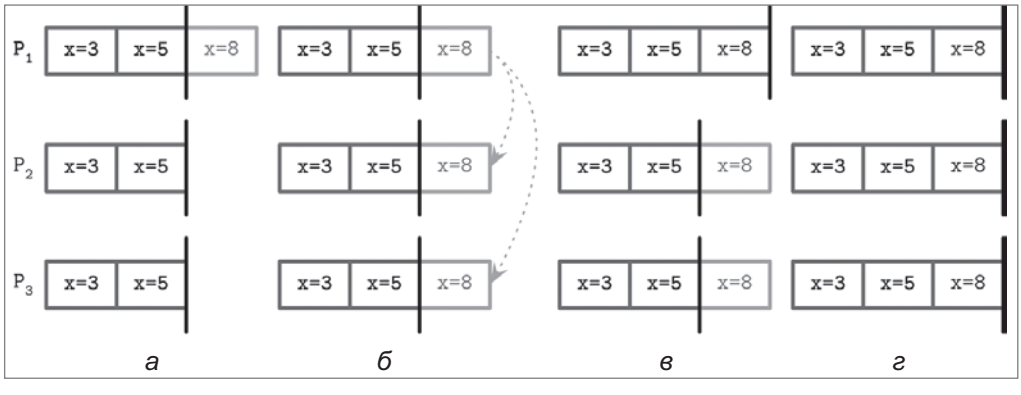

Рис. 14.11. Процедура фиксации в алгоритме Raft с лидером P_1

Рисунок 14.12 показывает пример раунда достижения консенсуса, где P_1 является лидером, который имеет самое актуальное представление о событиях. Лидер продолжает выполнение алгоритма, реплицируя записи на последователей и фиксируя их после сбора подтверждений. Фиксация записи также фиксирует все записи,

предшествующие ей в журнале. Только лидер может принять решение, можно ли фиксировать запись. Каждая запись в журнале помечается идентификатором периода (число в верхнем правом углу каждого поля записи в журнале) и индексом журнала, определяющим его положение в журнале. Зафиксированные записи гарантированно реплицируются в кворум участников, и их можно безопасно применять к конечному автомату в порядке их появления в журнале.

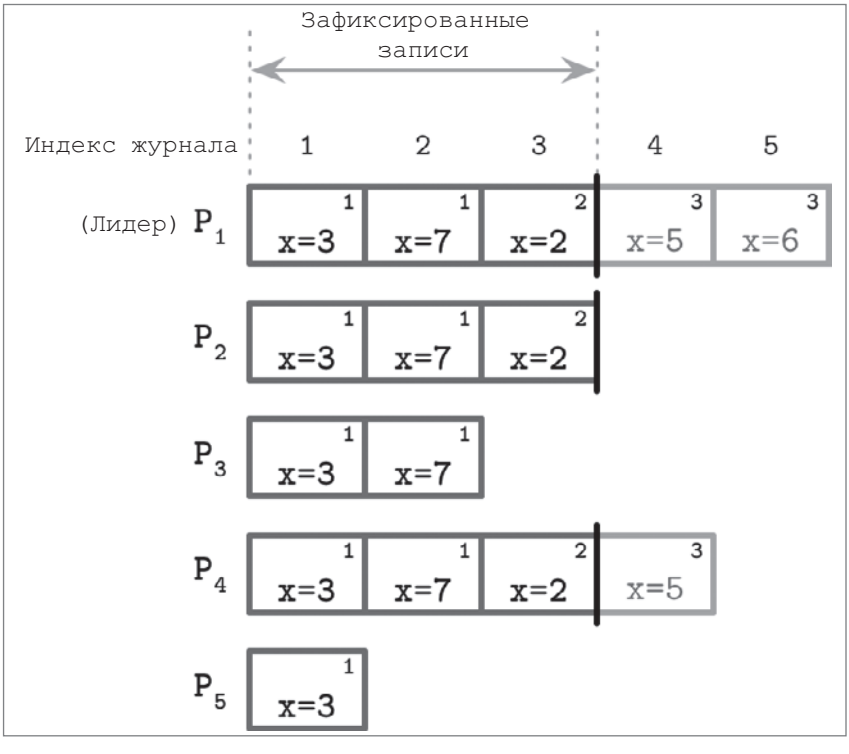

Рис. 14.12. Конечный автомат алгоритма Raft

Сценарии отказов

Когда несколько последователей решают стать кандидатами и ни один из кандидатов не может набрать большинство голосов, такая ситуация называется *разделенным голосованием*. В алгоритме Raft используются рандомизированные таймеры, чтобы уменьшить вероятность того, что несколько последовательных процессов выбора окажутся разделенными. Один из кандидатов может начать следующий раунд выбора раньше и набрать достаточно голосов, в то время как другие находятся в спящем режиме и уступают ему. Такой подход ускоряет выбор, не требуя дополнительной координации между кандидатами.

Последователи могут отключиться или медленно отвечать, а лидер должен приложить все усилия, чтобы обеспечить доставку сообщений. Он может попытаться

отправить сообщения еще раз, если он не получил подтверждение в течение ожидаемого времени. Для оптимизации производительности он может отправлять несколько сообщений параллельно.

Поскольку записи, реплицируемые лидером, идентифицируются однозначно, повторная доставка сообщений гарантированно не нарушает порядок ведения журнала. Последователи дедуплицируют сообщения, используя их порядковые идентификаторы, гарантируя, что двойная доставка не окажет нежелательных побочных эффектов.

Порядковые идентификаторы также используются для обеспечения упорядочения журнала. Последователь отклоняет отправленную лидером запись с самым большим номером, если идентификатор и период записи, которая непосредственно предшествует ей, не соответствуют записи с самым высоким номером согласно его собственным записям. Если записи в двух журналах в разных репликах имеют один и тот же период и один и тот же индекс, они хранят одну и ту же команду и все записи, предшествующие им, одинаковы.

Алгоритм Raft гарантирует, что незафиксированное сообщение не будет отображаться как зафиксированное, но из-за медленной работы сети или реплики уже зафиксированные сообщения все еще могут рассматриваться как находящиеся в процессе выполнения, что является довольно безвредный свойством, и этот вопрос можно решить, повторяя команду клиента, пока она в конечном итоге не будет зафиксирована [HOWARD14].

С целью обнаружения отказа лидер должен посылать последователям контрольные пакеты. Таким образом лидер сохраняет свой период. Когда один из узлов замечает, что текущий лидер не работает, он пытается инициировать выбор. Вновь избранный лидер должен восстановить состояние кластера до последней известной актуальной записи в журнале. Это достигается путем нахождения общей позиции (запись в журнале с самым большим номером, касательно которой согласны и лидер, и последователь) и приказа последователям отбросить все (незафиксированные) записи, добавленные после этой точки. Затем он отправляет самые последние записи из своего журнала, перезаписывая историю подписчиков. Собственные записи журнала лидера никогда не удаляются и не перезаписываются: в свой журнал он может только добавлять записи.

Резюмируя, можно сказать, что алгоритм Raft предоставляет следующие гарантии:

- Только один лидер может быть избран одновременно на заданный период; в течение одного периода не может быть двух активных лидеров.
- Лидер не удаляет и не переупорядочивает содержимое журнала; он только добавляет новые сообщения к нему.
- Зафиксированные записи в журнале гарантированно присутствуют в журналах для последующих лидеров, и журнал нельзя вернуть в прежнее состояние, поскольку перед фиксацией запись, как известно, реплицируется лидером.
- Все сообщения однозначно идентифицируются по идентификаторам сообщений и периодов; ни текущий, ни последующие лидеры не могут повторно использовать один и тот же идентификатор для другой записи.

С момента своего появления алгоритм Raft стал весьма популярным и в настоящее время используется во многих базах данных и других распределенных системах, включая CockroachDB (https://databass.dev/links/70), Etcd (https://databass.dev/links/71) и Consul (https://databass.dev/links/72). Это может быть связано с его простотой, но также может означать, что Raft оправдывает обещание быть надежным алгоритмом достижения консенсуса.

Византийский консенсус

Все алгоритмы достижения консенсуса, которые мы обсуждали до сих пор, предполагают невизантийские ошибки (см. подраздел «Произвольные ошибки» на с. 216). То есть узлы выполняют алгоритм «добросовестно» и не пытаются использовать его в своих интересах или подделать результаты.

Как мы увидим, это предположение позволяет достичь консенсуса с меньшим числом доступных участников и с меньшим количеством циклов «запрос–подтверждение», необходимых для фиксации. Однако распределенные системы иногда развертываются в потенциально опасной среде, где узлы не управляются одним и тем же объектом, и потому нам нужны алгоритмы, способные гарантировать корректное функционирование системы даже в случае неустойчивого или злонамеренного поведения некоторых узлов. Причиной византийских ошибок может быть не только злой умысел, но и программные ошибки, неправильная настройка, аппаратные проблемы или искажение данных.

Большинство византийских алгоритмов достижения консенсуса требуют N^2 сообщений для выполнения каждого шага алгоритма, где N — размер кворума, поскольку все узлы в кворуме должны связываться каждым другим узлом. Это необходимо для перекрестной проверки других узлов на каждом шаге, поскольку узлы не могут полагаться друг на друга или на лидера и должны проверять поведение других узлов, сравнивая возвращаемые результаты с ответами большинства.

Здесь мы обсудим только один византийский алгоритм достижения консенсуса — *практическую устойчивость к византийским ошибкам* (Practical Byzantine Fault Tolerance, PBFT) [CASTRO99]. Алгоритм PBFT предполагает независимые отказы узлов (т. е. отказы могут быть скоординированными, но система не может быть охвачена целиком, или по крайней мере это не может быть сделано с использованием одного и того же метода атаки). Система делает слабые предположения о синхронизации, например определяет, что при нормальном поведении сети в ней могут возникать сбои, но они не продолжаются бесконечно долго, и в конечном итоге сеть восстанавливает свою работоспособность.

Весь обмен данными между узлами шифруется, что предотвращает подделку сообщений и сетевые атаки. Реплики знают открытые ключи друг друга для проверки подлинности и шифрования сообщений. Неисправные узлы могут пропускать информацию из системы наружу, поскольку, несмотря на использование шифрования,

каждый узел должен интерпретировать содержимое сообщений, чтобы реагировать на них. Это не ставит под сомнение алгоритм, поскольку он служит другой цели.

Алгоритм PBFT

Чтобы алгоритм PBFT мог гарантировать безопасность и живучесть, неисправными могут быть не более `(n – 1)/3` реплик (где `n` — общее количество участников). Чтобы система выдерживала наличие `f` скомпрометированных узлов, необходимо иметь как минимум `n = 3f + 1` узлов. Это объясняется тем, что большинство узлов должно согласиться со значением: `f` реплик могут быть неисправными, и может быть `f` реплик, которые не отвечают, но могут и не быть неисправными (например, из-за распада сети, отключения питания или технического обслуживания). Алгоритм должен быть в состоянии собрать достаточно ответов от исправных реплик, чтобы их количество превосходило количество неисправных.

Свойства консенсуса для алгоритма PBFT аналогичны свойствам других алгоритмов достижения консенсуса: все исправные реплики должны согласовать и набор полученных значений, и их порядок, несмотря на возможные отказы.

Чтобы различать конфигурации кластера, алгоритм PBFT использует *представления*. В каждом представлении одна из реплик является *основной*, а остальные считаются *резервными*. Все узлы нумеруются последовательно, а индекс основного узла равен остатку от целочисленного деления `v` на `N`, где `v` — это идентификатор представления, а `N` — количество узлов в текущей конфигурации. Представление может измениться в случаях, когда отказывает основной узел. Клиенты применяют свои операции к основному узлу. Основной узел рассылает запросы резервным узлам, которые выполняют запросы и отправляют ответ обратно клиенту. Чтобы операция прошла успешно, клиент ожидает, чтобы `f + 1` реплик ответило *с одинаковым результатом*.

После того как основной узел получает запрос клиента, выполнение протокола происходит в три этапа:

Предварительная подготовка (preprepare)

Основной узел передает сообщение, содержащее идентификатор представления, уникальный монотонно увеличивающийся идентификатор, полезную нагрузку (запрос клиента) и хэш-сумму полезной нагрузки. Хэш-сумма вычисляется с использованием надежной хэш-функции, устойчивой к коллизиям, и подписывается отправителем. Резервный узел принимает сообщение, если его представление совпадает с представлением основного узла и запрос клиента не был подделан: вычисленная хэш-сумма полезных данных совпадает с полученной.

Подготовка (prepare)

Если резервный узел принимает сообщение предварительной подготовки, он входит в этап подготовки и начинает рассылку всем другим репликам (включая основную) сообщений подготовки, содержащих идентификатор представления, идентификатор сообщения и хэш-сумму полезных данных, но без самих по-

лезных данных. Реплики завершают этап подготовки только после получения `2f` сообщений подготовки от *разных* резервных узлов, которые соответствуют сообщению, полученному во время предварительной подготовки: у них должны быть одинаковые представления, идентификатор и хэш-сумма.

Фиксация (commit)

Далее резервный узел переходит к этапу фиксации, в ходе которого он рассылает сообщения о фиксации всем остальным репликам и ожидает сбора `2f + 1` совпадающих сообщений фиксации (возможно, включая свои собственные) от других участников.

В этом случае *хэш-сумма* используется для уменьшения размера сообщения на этапе подготовки, поскольку нет необходимости пересылать все полезные данные для проверки, так как хэш-сумма служит в качестве краткого представления полезных данных. Криптографические хэш-функции устойчивы к коллизиям: сложно получить два значения с одинаковой хэш-суммой, а тем более два сообщения с совпадающими хэш-суммами, которые бы *имели смысл* в контексте системы. Кроме того, хэш-суммы *подписываются*, чтобы гарантировать поступление хэш-сумм из надежного источника.

Число `2f` здесь играет важную роль, поскольку алгоритм должен гарантировать, чтобы клиенту ответило как минимум `f + 1` исправных реплик.

На рис. 14.13 представлена схема последовательности действий, отражающая случай нормальной работы раунда алгоритма PBFT: клиент отправляет запрос узлу P_1, после чего узлы переходят от этапа к этапу, собирая достаточное количество совпадающих ответов от *корректно работающих* одноранговых узлов. Узел P_4, возможно, дал сбой или выдал в качестве ответа несовпадающие сообщения, поэтому его ответы не были засчитаны.

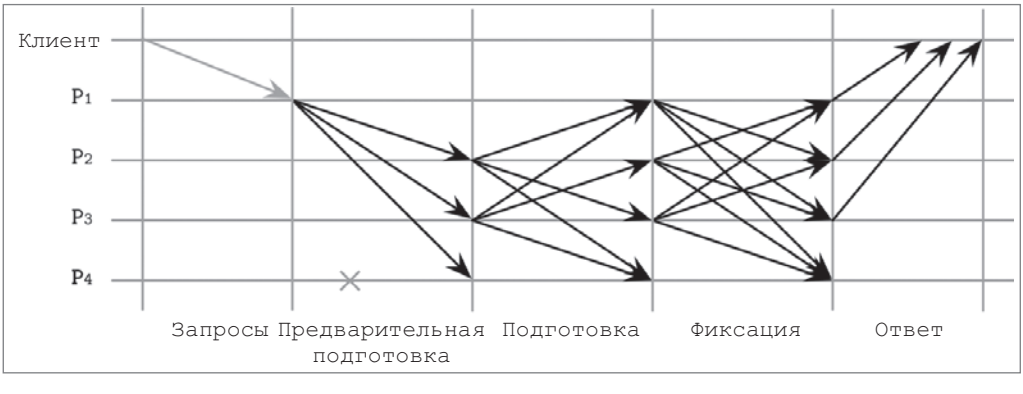

Рис. 14.13. Достижение консенсуса алгоритмом PBFT, случай нормальной работы

На этапах подготовки и фиксации узлы обмениваются данными, отправляя сообщения друг другу и ожидая сообщений от соответствующего числа других узлов, чтобы проверить, совпадают ли они, и не допустить рассылки некорректных сообщений.

Одноранговые узлы перекрестно проверяют все сообщения, чтобы только исправные узлы могли успешно зафиксировать сообщения. Если не удается собрать достаточное количество совпадающих сообщений, то узел не переходит к следующему шагу.

Когда реплики собирают достаточное количество сообщений фиксации, они уведомляют клиента, заканчивая раунд. Клиент может быть уверен в корректности выполнения только после получения `f + 1` совпадающих ответов.

Изменения в представлении происходят, когда реплики замечают, что основной узел неактивен, что вызывает у них подозрения в том, что он дал сбой. Узлы, обнаружившие отказ основного узла, перестают отвечать на дальнейшие сообщения (за исключением сообщений, относящихся к контрольной точке и изменению представления), рассылают уведомление об изменении представления и ждут подтверждения. После того как основной узел нового представления получает `2f` событий смены представления, он инициирует новое представление.

Чтобы уменьшить количество сообщений в протоколе, клиенты могут собирать `2f + 1` совпадающих ответов от узлов, которые *предварительно* выполняют запрос (например, после того, как они собрали достаточное количество совпадающих сообщений `Подготовлен`). Если клиент не может собрать достаточное количество совпадающих предварительных ответов, он повторяет попытку и ожидает `f + 1` непредварительных ответов, как описано ранее.

Операции с доступом только для чтения в алгоритме PBFT могут выполняться всего за один цикл «запрос–подтверждение». Клиент отправляет запрос на чтение всем репликам. Реплики выполняют запрос в своих предварительных состояниях, после того как фиксируются все текущие изменения состояния в предназначенном для чтения значении, и отвечают клиенту. После сбора `2f + 1` ответов с одинаковыми значениями от разных реплик операция завершается.

Восстановление и контрольные точки

Реплики сохраняют принятые сообщения в постоянном журнале. Каждое сообщение должно храниться до тех пор, пока оно не будет выполнено как минимум `f + 1` узлами. Журнал можно использовать для приведения в актуальное состояние других реплик в случае распада сети, но для восстановления реплик нужны некоторые средства проверки корректности состояния, которое они получают, поскольку в противном случае восстановление можно использовать в качестве вектора атаки.

Чтобы показать, что состояние корректно, узлы вычисляют хэш-сумму состояния для сообщений до заданного порядкового номера. Узлы могут сравнивать хэш-суммы, проверять целостность состояния и следить за тем, чтобы сообщения, полученные во время восстановления, добавлялись в корректное конечное состояние. Этот процесс слишком затратен для того, чтобы его можно было выполнять при каждом запросе.

После каждых `N` запросов, где `N` — настраиваемая константа, основной узел создает стабильную контрольную точку, в которой он рассылает последний порядковый

номер последнего запроса, выполнение которого отражено в состоянии, и хэш-сумму этого состояния. Затем он ждет ответа от `2f + 1` реплик. Эти ответы служат в качестве доказательства для этой контрольной точки, а также в качестве гарантии того, что реплики могут безопасно отбросить состояние для всех сообщений предварительной подготовки, собственно подготовки, фиксации и проверки до заданного порядкового номера.

Византийскую отказоустойчивость важно понимать и использовать в системах хранения, которые развертываются в потенциально опасных сетях. В большинстве случаев можно обойтись проверкой подлинности и шифрованием межузловой связи, но если нет доверия между отдельными частями системы, следует использовать алгоритмы наподобие PBFT.

Поскольку алгоритмы, устойчивые к византийским ошибкам, требуют значительных затрат в силу большого количества пересылаемых сообщений, важно понимать их сценарии использования. В ряде других протоколов, примеры которых описаны в источниках [BAUDET19] и [BUCHMAN18], предпринимается попытка оптимизировать алгоритм PBFT для систем с большим количеством участников.

Итоги

Алгоритмы достижения консенсуса — одна из самых интересных, но в то же время и самых сложных тем в сфере распределенных систем. За последние несколько лет появились новые алгоритмы и множество реализаций существующих алгоритмов, что доказывает возрастающую важность и популярность этой темы.

В этой главе мы обсудили классический алгоритм Паксос и несколько его вариантов, каждый из которых улучшает его свойства:

Мульти-Паксос

Позволяет заявителю сохранять свою роль и реплицировать несколько значений вместо одного.

Быстрый Паксос

Позволяет сократить количество сообщений за счет использования быстрых раундов, когда акцепторы могут продолжать выполнение, принимая сообщения от заявителей, отличных от выбранного лидера.

ЭПаксос

Устанавливает порядок событий путем обработки зависимостей между отправленными сообщениями.

Гибкий Паксос

Снижает требования к кворуму, предъявляя лишь требование о том, чтобы кворум для первого этапа (голосования) пересекался с кворумом для второго этапа (репликации).

Алгоритм Raft описывает консенсус в более простых терминах и делает сущность лидера объектом первого класса. Этот алгоритм отделяет друг от друга репликацию журнала, выбор лидера и безопасность.

Чтобы гарантировать безопасность процесса достижения консенсуса в средах с возможным вмешательством злоумышленников, следует использовать византийские отказоустойчивые алгоритмы, например PBFT. В алгоритме PBFT участники перекрестно проверяют ответы друг друга и продолжают выполнение шагов только при наличии достаточного количества узлов, соблюдающих предписываемые алгоритмом правила.

Дополнительная литература

Если вы хотите узнать больше о концепциях, упомянутых в этой главе, вы можете обратиться к следующим источникам:

Атомарная рассылка

Junqueira, Flavio P., Benjamin C. Reed, and Marco Serafini. "Zab: High-performance broadcast for primary-backup systems." 2011. In Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN'11): 245–256.

Hunt, Patrick, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. "ZooKeeper: wait-free coordination for internet-scale systems." In Proceedings of the 2010 USENIX conference on USENIX annual technical conference (USENIXATC'10): 11.

Oki, Brian M., and Barbara H. Liskov. 1988. "Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems." In Proceedings of the seventh annual ACM Symposium on Principles of distributed computing (PODC '88): 8–17.

Van Renesse, Robbert, Nicolas Schiper, and Fred B. Schneider. 2014. "Vive la Différence: Paxos vs. Viewstamped Replication vs. Zab."

Классический Паксос

Lamport, Leslie. 1998. "The part-time parliament." ACM Transactions on Computer Systems 16, no. 2 (May): 133–169.

Lamport, Leslie. 2001. "Paxos made simple." ACM SIGACT News 32, no. 4: 51–58.

Lamport, Leslie. 2005. "Generalized Consensus and Paxos." Technical Report MSR-TR-2005-33. Microsoft Research, Mountain View, CA.

Primi, Marco. 2009. "Paxos made code: Implementing a high throughput Atomic Broadcast." (Libpaxos code: https://bitbucket.org/sciascid/libpaxos/src/master/.

Быстрый Паксос

Lamport, Leslie. 2005. "Fast Paxos." 14 July 2005. Microsoft Research.

Мульти-Паксос

Chandra, Tushar D., Robert Griesemer, and Joshua Redstone. 2007. "Paxos made live: an engineering perspective." In Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing (PODC '07): 398–407.

Van Renesse, Robbert and Deniz Altinbuken. 2015. "Paxos Made Moderately Complex." ACM Computing Surveys 47, no. 3 (February): Article 42. https:// doi.org/10.1145/2673577.

ЭПаксос

Moraru, Iulian, David G. Andersen, and Michael Kaminsky. 2013. "There is more consensus in Egalitarian parliaments." In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13): 358–372.

Moraru, I., D. G. Andersen, and M. Kaminsky. 2013. "A proof of correctness for Egalitarian Paxos." Technical report, Parallel Data Laboratory, Carnegie Mellon University, Aug. 2013.

Raft

Ongaro, Diego, and John Ousterhout. 2014. "In search of an understandable consensus algorithm." In Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (USENIX ATC'14), Garth Gibson and Nickolai Zeldovich (Eds.): 305–320.

Howard, H. 2014. «ARC: Analysis of Raft Consensus." Technical Report UCAM-CL-TR-857, University of Cambridge, Computer Laboratory, July 2014.

Howard, Heidi, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. 2015. "Raft Refloated: Do We Have Consensus?" SIGOPS Operating Systems Review 49, no. 1 (January): 12–21. https://doi.org/10.1145/2723872.2723876.

Последние разработки

Howard, Heidi and Richard Mortier. 2019. "A Generalised Solution to Distributed Consensus." 18 Feb 2019.

ЧАСТЬ II

Заключение

Производительность и масштабируемость — важные свойства любой СУБД. Подсистема хранения и локальный для узла путь чтения-записи влияют главным образом на *производительность* системы — на то, насколько быстро она может локально обрабатывать запросы. В то же время подсистема, отвечающая за связь в кластере, влияет главным образом на *масштабируемость* СУБД — на максимальный размер и емкость кластера. Однако подсистему хранения можно будет использовать лишь в ограниченном количестве сценариев использования, если она не поддается масштабированию или ее производительность снижается по мере роста набора данных. В то же время мы не получим хороших результатов, разместив медленный протокол атомарной фиксации поверх даже самой быстрой подсистемы хранения.

Процессы, происходящие на уровнях распределенной системы, кластера и узла, взаимосвязаны, и их необходимо рассматривать как единое целое. При проектировании СУБД нужно учитывать, насколько хорошо сочетаются друг с другом различные подсистемы.

Часть II началась с обсуждения того, чем распределенные системы отличаются от одноузловых приложений и какие трудности ожидаются в таких средах.

Мы рассмотрели основные «строительные блоки» распределенной системы, различные модели согласованности и несколько важных классов распределенных алгоритмов, некоторые из которых можно использовать для реализации этих моделей:

Обнаружение отказов

Точное и эффективное определение отказа удаленного процесса.

Выбор лидера

Быстрый и надежный выбор одного процесса для временного выполнения функции координатора.

Распространение

Надежное распространение информации с использованием одноранговой связи.

Антиэнтропия

Определение и устранение расхождения в состоянии между узлами.

Распределенные транзакции

Атомарное выполнение серии многосекционных операций.

Консенсус

Достижение согласия между удаленными участниками при устойчивости к отказам процессов.

Эти алгоритмы используются во многих СУБД, очередях сообщений, планировщиках и в другом важном инфраструктурном программном обеспечении. Используя почерпнутую из этой книги информацию, вы сможете лучше понять, как они работают, что, в свою очередь, поможет вам взвешенно выбирать необходимое программное обеспечение и выявлять потенциальные проблемы.

Дополнительная литература

В конце каждой главы можно найти источники, связанные с материалом, представленным в главе. Здесь вы найдете книги, к которым можно обратиться для дальнейшего изучения, охватывающие как концепции, упомянутые в этой книге, так и многие другие. Данный список не претендует на полноту, но в то же время эти источники содержат много важной и полезной информации по темам, актуальным для энтузиастов СУБД, некоторые из которых не были освещены в этой книге.

СУБД

Bernstein, Philip A., Vassco Hadzilacos, and Nathan Goodman. 1987. Concurrency Control and Recovery in Database Systems. Boston: Addison-Wesley Longman.

Korth, Henry F. and Abraham Silberschatz. 1986. Database System Concepts. New York: McGraw-Hill.

Gray, Jim and Andreas Reuter. 1992. Transaction Processing: Concepts and Techniques (1st Ed.). San Francisco: Morgan Kaufmann.

Stonebraker, Michael and Joseph M. Hellerstein (Eds.). 1998. Readings in Database Systems (3rd Ed.). San Francisco: Morgan Kaufmann.

Weikum, Gerhard and Gottfried Vossen. 2001. Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. San Francisco: Morgan Kaufmann.

Ramakrishnan, Raghu and Johannes Gehrke. 2002. Database Management Systems (3rd Ed.). New York: McGraw-Hill.

Garcia-Molina, Hector, Jeffrey D. Ullman, and Jennifer Widom. 2008. Database Systems: The Complete Book (2nd Ed.). Upper Saddle River, NJ: Prentice Hall[1].

[1] *Ульман Джеффри Д., Уидом Дженнифер, Гарсиа-Молина Гектор*. Системы баз данных. Полный курс.

Bernstein, Philip A. and Eric Newcomer. 2009. Principles of Transaction Processing (2nd Ed.). San Francisco: Morgan Kaufmann.

Elmasri, Ramez and Shamkant Navathe. 2010. Fundamentals of Database Systems (6th Ed.). Boston: Addison-Wesley.

Lake, Peter and Paul Crowther. 2013. Concise Guide to Databases: A Practical Introduction. New York: Springer.

Härder, Theo, Caetano Sauer, Goetz Graefe, and Wey Guy. 2015. Instant recovery with write-ahead logging. Datenbank-Spektrum.

Распределенные системы

Lynch, Nancy A. Distributed Algorithms. 1996. San Francisco: Morgan Kaufmann.

Attiya, Hagit, and Jennifer Welch. 2004. Distributed Computing: Fundamentals, Simulations and Advanced Topics. Hoboken, NJ: John Wiley & Sons.

Birman, Kenneth P. 2005. Reliable Distributed Systems: Technologies, Web Services, and Applications. Berlin: Springer-Verlag.

Cachin, Christian, Rachid Guerraoui, and Lus Rodrigues. 2011. Introduction to Reliable and Secure Distributed Programming (2nd Ed.). New York: Springer[1].

Fokkink, Wan. 2013. Distributed Algorithms: An Intuitive Approach. The MIT Press[2].

Ghosh, Sukumar. Distributed Systems: An Algorithmic Approach (2nd Ed.). Chapman & Hall/CRC.

Tanenbaum Andrew S. and Maarten van Steen. 2017. Distributed Systems: Principles and Paradigms (3rd Ed.). Boston: Pearson[3].

Эксплуатация баз данных

Beyer, Betsy, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016 Site Reliability Engineering: How Google Runs Production Systems (1st Ed.). Boston: O'Reilly Media.

Blank-Edelman, David N. 2018. Seeking SRE. Boston: O'Reilly Media.

Campbell, Laine and Charity Majors. 2017. Database Reliability Engineering: Designing and Operating Resilient Database Systems (1st Ed.). Boston: O'Reilly Media. + Sridharan, Cindy. 2018. Distributed Systems Observability: A Guide to Building Robust Systems. Boston: O'Reilly Media.

---

[1] *Качин К., Гуерру Р., Родригес Л.* Введение в надежное и безопасное распределенное программирование. М.: ДМК Пресс, 2016.

[2] *Фоккинк У.* Распределенные алгоритмы. Интуитивный подход. СПб.: Питер, 2017.

[3] *Таненбаум Э., Стеен М. ван.* Распределенные системы. Принципы и парадигмы. СПб.: Питер, 2003.

Об авторе

Алекс Петров — инженер по инфраструктуре данных, энтузиаст систем хранения и баз данных, один из разработчиков и член комитета по управлению проектом СУБД Apache Cassandra. Сферы интересов: системы хранения, распределенные системы и алгоритмы.

Об обложке

На обложке книги изображен павлиний ботус. Это название относится к *Bothus lunatus* и *Bothus mancus*, обитателям мелких прибрежных вод Средней Атлантики и Индо-Тихоокеанской области соответственно.

Своим названием павлиний ботус обязан синей коже с цветочным рисунком, однако он способен менять внешность в зависимости от непосредственного окружения. Эта способность к маскировке, возможно, связана со зрением, так как рыба не может изменить свой внешний вид, если один из глаз закрыт.

Взрослые камбаловые плавают в горизонтальном положении, а не в вертикальном — спиной вверх и животом вниз, — как большинство других рыб. Плавая, камбаловые скользят примерно в дюйме (2,54 см) от дна, точно повторяя его контуры. По мере роста особи один из глаз смещается, чтобы присоединиться ко второму на той же стороне, позволяя рыбе смотреть одновременно вперед и назад. Это естественно, так как вместо того, чтобы плавать вертикально, павлиний ботус повторяет контуры морского дна, скользя примерно в дюйме от него, с узорчатой стороной, всегда обращенной вверх.

Хотя сегодняшний природоохранный статус павлиньего ботуса обозначен как «находятся под наименьшей угрозой», многие животные, обитающие на обложках издательства O'Reilly, находятся под угрозой исчезновения; все они важны для мира.

Иллюстрация на обложке выполнена Карен Монтгомери на основе черно-белой гравюры из Музея естественной истории Лоури.