Алгоритмы и структуры для массивных наборов данных

Джейла Меджедович
Эмин Тахирович

MANNING
ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Джейла Меджедович, Эмин Тахирович

Алгоритмы и структуры данных для массивных наборов данных

Algorithms and Data Structures for Massive Datasets

DZEJLA MEDJEDOVIC
EMIN TAHIROVIC
Illustrated by INES DEDOVIC

MANNING
SHELTER ISLAND

Алгоритмы и структуры для массивных наборов данных

ДЖЕЙЛА МЕДЖЕДОВИЧ
ЭМИН ТАХИРОВИЧ
Иллюстрации ИНЕС ДЕДОВИЧ

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2024

УДК 32.973.2
ББК 004.021
М42

М42 Джейла Меджедович, Эмин Тахирович
Алгоритмы и структуры для массивных наборов данных / пер. с англ. А. В. Логунова. – М.: ДМК Пресс, 2024. – 340 с.: ил.

ISBN 978-5-93700-250-1

Стандартные алгоритмы и структуры при применении к крупным распределенным наборам данных могут становиться медленными — или вообще не работать. Правильный подбор алгоритмов, предназначенных для работы с большими данными, экономит время, повышает точность и снижает стоимость обработки.

Книга знакомит с методами обработки и анализа больших распределенных данных. Насыщенное отраслевыми историями и занимательными иллюстрациями, это удобное руководство позволяет легко понять даже сложные концепции. Вы научитесь применять на реальных примерах такие мощные алгоритмы, как фильтры Блума, набросок count-min, HyperLogLog и LSM-деревья, в своих собственных проектах.

ISBN 978-1-61729-803-5 (англ.)
ISBN 978-5-93700-250-1 (рус.)

Оглавление

Предисловие

Идея написания этой книги оформилась, когда мы вместе преподавали в Международном университете Сараево. В ходе обсуждения с нашими студентами, которые работали в местных компаниях, мы поняли, что структуры для массивных данных получают все большее распространение в повседневном применении инженерами и исследователями данных. Эти технологии использовались по всему миру не только компаниями Google и Facebook, чтобы решать свои задачи обеспечения масштабируемости, но и компаниями с гораздо меньшими объемами данных, чьи системы начали сталкиваться с постоянно растущими потребностями в скорости обработки данных.

За обедом мы размышляли о том, куда студент, который учится внедрять структуры данных HyperLogLog или фильтр Блума в работающую производственную систему, мог бы обратиться за удобным обзором их применения. Оригинальные статьи, в которых эти структуры данных всесторонне представляются, нередко были очень глубокими с математической точки зрения, но с малым контекстом для инженера данных, пытающегося встроить эту структуру данных в реальную систему с реальными данными. Если не считать эпизодических блог-постов с описанием реализации структур данных, ресурсов, которые объединяли бы эти специфичные для массивных данных алгоритмические знания, было мало либо вообще не существовало.

Мы хотели написать книгу, которая могла бы представить эти высокотехничные предметы в дружественном тоне, а также дать более качественный ответ на вечный вопрос студентов: «где это можно использовать?» Сочетание вероятностных и потоковых структур данных, а также структур данных внешней памяти в живую экосистему массивных данных и демонстрация практических примеров использования были непростым делом для двух преподавателей в вельветовых пиджаках. Мы не были готовы полностью отказаться от математики, поэтому поставили перед собой задачу попытаться выразить как можно больше математических концепций в легко воспринимаемой на интуитивном уровне форме, не приводя ни одного доказательства.

Нам чрезвычайно повезло работать с Инес, иллюстратором с передовым инженерным образованием, которая создала действенные и очаровательные рисунки для иллюстрации сложных алгоритмических материалов. Если вы когда-либо объясняли кому-либо алгоритм, то знаете, что он по своей сути визуален, однако в книгах по компьютерным алгоритмам зачастую не так много визуальных подсказок. Будем надеяться, что эта книга станет еще одним маленьким шагом к тому, чтобы это изменить.

Каждая хорошая история нуждается в конфликте, и в этой книге главным является компромисс, возникающий из-за ограничений, налагаемых крупными данными. Главнейшая тема нашей книги – пожертвовать точностью структуры данных ради экономии пространства. Отыскание этой золотой середины в производительности и усвоение способов уравновешивания разных конкурирующих целей в сложном конвейере данных – вот главные вызовы, которые привносятся массивными данными в повестку дня, и ключевые уроки, которые можно извлечь из этой книги.

Мы признательны за то, что у нас была возможность написать книгу на такую захватывающую и важную тему. Мы невероятно благодарны всем, кто оставлял отзывы, пока книга находилась в разработке. Начав писать ее как ученые, мы закончили ее как инженеры в компаниях, специализирующихся на обработке данных (это действительно практическая книга!). Надеемся, что ознакомление с этим материалом обогатит ваш набор алгоритмических инструментов и позволит вам взяться за решение следующей задачи обработки больших данных с любопытством и уверенностью.

Благодарности

С момента, когда начинаешь писать книгу, и до самого конца происходит много событий, и выдавать главы на-гора, ориентируясь во всех превратностях жизни и работы, не всегда легко. К счастью, у нас была целая деревня людей, которые нас поддерживали, подбадривали на протяжении всего процесса и приносили еду накануне сдачи этапов работы.

Во-первых, мы хотели бы поблагодарить наших родителей Мердзану и Сафера, а также Зикрету и Эсада. Ваши примеры и руководство на протяжении всей жизни дали нам свободу читать, учиться и осваивать, а также дали нам почувствовать, что мы можем – и должны – писать книги. Без этого данная книга никогда бы не появилась на свет. Мы также хотели бы поблагодарить наших дорогих братьев, сестер и племянниц, которые поддерживали нас на протяжении всего процесса написания: Дзейру, Энсар, Аджлу, Сериф, Мерсад, Далал и Лейну. Эмин также хотел бы поблагодарить свою тетю Индиру за то, что она терпела его во время учебы во Франкфурте.

Во-вторых, мы хотели бы поблагодарить наших друзей, которые неоднократно интересовались продвижением работы над книгой (даже зная, что им придется выслушивать реально длинный ответ). Многие наши друзья работают в областях, отличных от вычислительных наук, поэтому их желание ознакомиться с нашими первыми главами имело гораздо большее значение.

Мы хотели бы поблагодарить наших студентов из Сараевской школы науки и технологий и Международного университета Сараево, которые вдохновили нас на написание этой книги и в разное время ее рецензировали.

Завершением работы над этой книгой мы обязаны нашему редактору Карен Миллер, которая проделала великолепную работу, проведя нас через весь процесс с невероятным сочетанием профессионализма и доброты. Ее проницательность и опыт сыграли решающую роль в формировании данной книги.

В процессе написания мы были на связи со многими сотрудниками издательства Manning. Коллектив издательства стремится к совершенству и применяет гибкий подход к изданию книг с учетом ранней обратной связи, что мы, как авторы, сочли невероятно полезным и вдохновляющим.

Мы хотели бы поблагодарить рецензентов издательства Manning: Алехандро Беллогина, Алекса Гаута, Анто Аравинта, Арно Бастенхофа, Кристофера Коттмайера, Чунсу Тана, Клиффорда Тербера, Даниэля Васкеса, Диего Казеллу, Германа Гонсалеса-Морриса, Хильде Ван Гизель, Жан-Франсуа Морина, Йенса Кристиана Бредаля Мэдсена, Джима Амрейна, Хуана Хосе Дурильо Барриону-эво, Хуана Антонио Руфес де Висенте, Келум Сенанаяке, Ману Сарина, Маркуса Янга, Марка Бауэра, Ниака Васкеса, Раушана Джа, Руи Лю, Сатей Саху, Себастьена Джанаса, Стюарта Перкса, Тима ван Дюрзена, Трэвиса Нельсон и Юрия Кушча. Ваши предложения помогли сделать эту книгу лучше.

Наконец, мы хотели бы поблагодарить наших читателей, благодаря чьему участию и вкладу книга была значительно лучше приспособлена для целевой аудитории.

Об этой книге

Книга «Алгоритмы и структуры для массивных наборов данных» предназначена для оказания помощи в разработке масштабируемых приложений и понимании алгоритмических строительных блоков, лежащих в основании систем обработки массивных данных. В книге рассматриваются разные алгоритмические аспекты разработки крупномасштабных приложений, которые предусматривают экономию места за счет использования вероятностных структур данных, обработку потоковых данных, работу с данными на диске и понимание компромиссов в производительности в системах управления базами данных.

Кому следует прочитать эту книгу

Эта книга предназначена для читателей, разбирающихся в фундаментальных структурах данных и алгоритмах. Бóльшая часть содержимого этой книги основана на материале, который обычно рассматривается на ранних курсах по структурам данных / алгоритмам: большинство глав начинаются с демонстрации традиционного решения задачи и причины, по которой тот или иной алгоритм либо структура данных оказываются безуспешными в контексте массивных данных. Несмотря на то что вводные разделы глав предлагают некоторое обсуждение базовых алгоритмов, этот материал служит лишь кратким обзором тематик, с которыми читатель уже должен чувствовать себя удобно. Читатель данной книги также должен обладать промежуточными знаниями в области программирования и понимать основы теории вероятностей. Никаких знаний о какой-либо конкретной системе или фреймворке не требуется (в этом вся красота алгоритмов), кроме базового знакомства с языком Python и псевдокодом.

Как эта книга организована: дорожная карта

Книга состоит из трех частей, охватываемых 11 главами. Часть I посвящена вероятностным лаконичным структурам данных, часть II – потоковым структурам данных и алгоритмам, а часть III – структурам данных и алгоритмам внешней памяти. Ниже приводится краткое изложение каждой главы.

- Глава 1 посвящена объяснению причины, по которой массивные данные представляют такую трудность для современных систем, и того, как эти трудности влияют на разработку алгоритмов и структур данных.

Часть I: Вероятностные лаконичные структуры данных

- Глава 2 посвящена хешированию и объяснению эволюции хеш-таблиц, чтобы удовлетворять требования со стороны крупных наборов данных и сложных распределенных систем (например, согласованное хеширование). Методы хеширования широко используются в последующих главах, поэтому данная глава служит подготовкой к другим главам части I.
- Глава 3 знакомит с задачей о приближенной принадлежности и двумя структурами данных, которые помогают ее решать: блумовским и порционным фильтрами. В главе представлены примеры использования и анализ частоты ложноположительных результатов, а также плюсы и минусы использования каждой структуры данных.
- Глава 4 посвящена описанию задачи оценивания частоты и вводит набросок count-min, структуру данных, которая решает задачу оценивания частоты чрезвычайно экономичным способом. В ней обсуждаются примеры использования в обработке естественного языка, обработке сенсорных данных и других областях, а также применение наброска count-min к таким задачам, как диапазонные запросы.
- Глава 5 посвящена углубленному изложению алгоритмов оценивания кардинального числа и алгоритмам HyperLogLog, а также их применениям. В этой главе используется мини-эксперимент, чтобы показать эволюцию точности от простого вероятностного подсчета до полной структуры данных HyperLogLog.

Часть II: Структуры и алгоритмы обработки потоковых данных

- Глава 6 представляет собой краткое введение в потоки данных как алгоритмическую концепцию и обработку (приложения по обработке) потоковых данных как проявление реального мира. С помощью нескольких практических примеров использования в архитектуре обработки потоковых данных мы покажем, как структуры данных из предыдущих глав вписываются в контекст потоковых данных.
- В главе 7 объясняется методика поддержания репрезентативной выборки из потока данных или из окна, скользящего над потоком. Мы объясняем ситуации, в которых могут интересовать смещенные выборки, и приводим примеры исходного кода, показывающие реализацию смещения выборки в сторону более недавно наблюдавшихся кортежей данных.
- Глава 8 посвящена вычислению приближенных квантилей на числовых данных из непрерывного потока данных. Мы опишем две конспективные структуры данных, или дайджесты: q-дайджест и t-дайджест. Мы дадим объяснение лежащих в их основе алгоритмов и сравним их друг с другом на реалистичном наборе данных.

Часть III: Структуры данных и алгоритмы внешней памяти

- Глава 9 посвящена введению в модель внешней памяти с несколькими примерами, демонстрирующими, как стоимость операций ввода-вывода преобладает над стоимостью центрального процессора при работе с данными в дистанционном хранилище. Эта глава открывает новые перспективы для разработчика алгоритмов, привыкшего думать об оптимизации алгоритмов с точки зрения стоимости центрального процессора.
- Глава 10 посвящена структурам данных, лежащих в основе магистральных баз данных, – B-деревьям и LSM-деревьям – и охватывает различные компромиссы между операциями чтения и записи при конструировании движка базы данных. Высокоуровневое понимание принципов работы этих структур данных должно помочь вам различать разные стили баз данных и выбирать правильную для вашего приложения.
- Глава 11 посвящена сортировке во внешней памяти и демонстрации оптимальных алгоритмов сортировки файлов на диске с использованием оптимизированных под внешнюю память версий сортировки слиянием и быстрой сортировки. В главе 11 сортировка используется в качестве примера, чтобы продемонстрировать виды оптимизации, которые можно применять для пакетных задач при их перемещении во внешнюю память.

Части I и II связаны друг с другом больше, чем с частью III, поскольку обе они касаются резидентных структур данных и темы максимизации точности при экономии пространства. Часть III имеет самостоятельную тему, и читатель, интересующийся исключительно ею, может пропустить ее вперед, не потеряв контекста. Кроме того, читать часть I перед частью II необязательно, но читатель, который сначала прочтет часть I, будет более подготовлен к пониманию части II, чем тот, кто сразу перейдет к ней.

Части II и III начинаются с главы, в которой объясняется модель и контекст (соответственно главы 6 и 9), и настоятельно рекомендуется прочитать эти главы, чтобы понять другие главы в соответствующих частях. Имея это в виду, не стесняйтесь обследовать книгу самостоятельно. Мы постарались написать все главы настолько самодостаточно, насколько это возможно. При необходимости вы всегда можете вернуться назад и получить более подробную информацию. Рекомендуем всем читателям прочитать главу 1, в которой объясняется причина, по которой массивные данные вызывают такой сдвиг парадигмы в части алгоритмов и структур данных, развертываемых в загруженных работой крупных инфраструктурах.

Об исходном коде

Несколько глав книги содержат исходный код, и в некоторых более сложных алгоритмах и тех, где контекст значительно его усложнил бы (например, алгоритмах внешней памяти), мы возвращаемся к псевдокоду. В большинстве фрагментов исходного кода и для создания мини-экспериментов, демонстрирующих производительность структур данных в некоторых главах, используются языки Python и R. Читатель не должен испытывать каких-то затруднений в реализации упражнений по программированию на выбранном им языке, поскольку рассматриваемые темы не являются специфичными для какого-либо конкретного языка или технологии.

Эта книга содержит множество примеров исходного кода в виде отдельных нумерованных листингов и внутри обычного текста. В обоих случаях исходный код отформатирован `шрифтом фиксированной ширины, подобным этому`, чтобы отделять его от обычного текста. Иногда исходный код также выделяется **`жирным шрифтом`**, чтобы подчеркивать тот исходный код, который изменился по сравнению с предыдущими шагами в данной главе, например когда новая функциональная возможность добавляется в существующую строку исходного кода.

Во многих случаях изначальный исходный код был переформатирован; мы добавили переносы строк и переработали отступы, чтобы уместиться в доступное пространство страницы книги. В редких случаях даже этого было недостаточно, и листинги включали маркеры продолжения строки (➥). Вдобавок нередко комментарии в исходном коде из листингов удалялись, когда исходный код описывался в тексте. Многие листинги сопровождаются аннотациями к исходному коду, выделяющими важные понятия и концепции.

Вы можете получить исполняемые фрагменты исходного кода из онлайновой версии этой книги в Livebook по адресу **https://livebook.manning.com/book/algorithms-and-data-structures-for-massive-datasets**. Полный исходный код примеров книги доступен для скачивания с веб-сайта издательства Manning по адресу **https://www.manning.com/books/algorithms-and-data-structures-for-massive-datasets**.

Об авторах

Джейла Меджедович, доктор философии, получила докторскую степень в лаборатории прикладных алгоритмов факультета вычислительных наук Университета Стоуни Брук, Нью-Йорк, в 2014 году. Джейла работала над рядом проектов в области алгоритмов обработки массивных данных, преподавала алгоритмы на различных уровнях, а также провела некоторое время в Microsoft. Увлечена преподаванием, продвижением образования в области вычислительных наук и передачей технологий. В настоящее время работает вице-президентом по обработке данных в Social Explorer, Inc.

Эмин Тахирович, доктор философии, получил докторскую степень по биостатистике в Пенсильванском университете в 2016 году и степень магистра теоретических вычислительных наук в Университете Гете во Франкфурте в 2008 году. Его статистическая методология и теоретические знания в области вычислительных наук делают его незаурядным исследователем в области науки о данных на стыке вычислительных методов и статистики. Работал в DBahn AG ИТ-консультантом и регулярно консультирует по проектам фармацевтические и технологические компании. Эмин работал доцентом кафедры разработки программного обеспечения в Международном университете Сараево. В настоящее время работает в HAProxy Technologies старшим исследователем данных.

Доктор Инес Дедович получила докторскую степень в Институте визуализации и компьютерного зрения на факультете электротехники Университета RWTH в Ахене, Германия. Работала научным сотрудником в исследовательском центре Юлиха и в настоящее время работает разработчиком программного обеспечения для систем видеонаблюдения в компании по автоматизации Jonas & Redmann. Инес более 10 лет также работала 3D-аниматором, художником комиксов и иллюстратором учебников. В этой книге она использует свои художественные и технические навыки для создания интуитивно понятных визуализаций технических концепций.

Глава 1

Введение

Эта глава охватывает следующие ниже темы:

- тема книги и ее структура;
- отличие этой книги от других книг по алгоритмам;
- влияние массивных наборов данных на устройство алгоритмов и структур данных;
- как эта книга поможет разрабатывать практические алгоритмы на практике;
- архитектуры компьютеров и систем, усложняющие работу с крупными объемами данных.

Раз вы взяли в руки эту книгу, то, вполне возможно, интересуетесь устройством алгоритмов и структур для массивных наборов данных и хотите разобраться в их отличии от «обычных» алгоритмов, с которыми вы сталкивались до сих пор. Означает ли название этой книги, что классические алгоритмы (например, двоичный поиск, сортировка слиянием, быстрая сортировка, поиск в глубину, поиск в ширину и многие другие фундаментальные алгоритмы), а также канонические структуры данных (например, массивы, матрицы, хеш-таблицы, деревья двоичного поиска, кучи) были созданы исключительно для малых наборов данных?

Ответ на этот вопрос не является ни коротким, ни простым, но если бы нужно было дать короткий и простой ответ, то он был бы «да». Объем понятия массивный набор данных имеет относительный характер и зависит от многих факторов, но суть такова: большинство простых алгоритмов и структур данных, о которых мы знаем и с которыми работаем на ежедневной основе, были разработаны с неявно принятым допущением о том, что все данные умещаются в основной, или оперативной, памяти (ОЗУ) компьютера. И поэтому после загрузки всех данных в оперативную память можно относительно быстро и легко обращаться к любому их элементу, и с этого момента конечной целью, с точки зрения эффективности, становится достижение максимальной производительности за наименьшее число

циклов центрального процессора. Именно этому и учит нас старый добрый анализ «О» большое (O(.)): он обычно выражает число базовых операций, требующихся в наихудшем случае, которые алгоритм должен выполнить, чтобы решить задачу. Указанными единицами работы могут быть сравнения, арифметика, побитовые операции, чтение/запись/копирование ячеек памяти или что-либо еще, что непосредственно транслируется в малое число циклов центрального процессора.

Однако если вы – исследователь данных[1], разработчик или инженер бэкендов, работающий в компании, которая собирает данные у своих пользователей, то хранение всех данных в рабочей памяти вашего компьютера зачастую нереализуемо. Сегодня многие приложения в таких областях, как банковское дело, электронная коммерция, научные приложения и интернет вещей, на рутинной основе манипулируют наборами данных размером в терабайт (Тб) или петабайт (Рб) (то есть вовсе не обязательно трудиться в Facebook или Google, чтобы на работе сталкиваться с массивными данными).

Возможно, вы задаетесь вопросом, а каким большим должен быть набор данных, чтобы можно было воспользоваться методами, показанными в этой книге. Мы намеренно избегаем навешивания ярлыков с указанием величины на так называемые массивные наборы данных или «компании по обработке больших данных», поскольку эта величина зависит от решаемой задачи, доступных инженеру вычислительных ресурсов, требований к системе и т. д. Некоторые компании, имея огромные наборы данных, к тому же располагают значительными ресурсами и могут позволять себе откладывать творческое осмысление проблем масштабируемости, инвестируя в инфраструктуру (например, покупая тонны оперативной памяти). Разработчик, оперирующий на среднекрупных наборах данных, но с ограниченным бюджетом на инфраструктуру и чрезвычайно высокими требованиями к производительности системы со стороны своего клиента, может извлечь выгоду из показанных в этой книге методов не меньше, чем кто-либо другой. Тем не менее, как мы увидим, даже компании с практически бесконечными ресурсами предпочитают заполнять эту дополнительную оперативную память продуманными пространственно-эффективными структурами данных.

Проблема массивных данных существует гораздо дольше, чем социальные сети и интернет. Одна из первых работ [1], в которой были представлены алгоритмы внешней памяти (класс алгоритмов, которые пренебрегают вычислительной стоимостью программы в пользу оптимизации гораздо более времязатратной стоимости передачи данных), появилась еще в 1988 году. В качестве практической мотивации того исследования авторы использовали пример крупных банков, которым приходилось ежедневно сортировать 2 млн чеков, причем чеки объемом около 800 Мб должны были сортироваться за ночь до начала следующего рабочего дня, используя рабочие элементы памяти того времени (~2–4 Мб). Выяснение

[1] Англ. data scientist. – *Прим. перев.*

того, как сортировать все чеки, имея возможность одновременно сортировать чеки объемом всего 4 Мб, и как это делать за наименьшее число обращений к диску, было актуальной задачей того времени, и с тех пор ее актуальность только возросла. С того времени объем данных вырос колоссально, но, что важнее, он рос гораздо быстрее, чем средний объем оперативной памяти.

Главным следствием ускоренного роста объема данных и главной идеей, мотивирующей алгоритмы в этой книге, является то, что сегодня большинство приложений характеризуются интенсивностью использования данных. Интенсивность использования данных (в отличие от интенсивности использования центрального процессора) означает, что узким местом приложения является передача данных туда и обратно и доступ к ним, а не выполнение вычислений с этими данными, после того как они станут доступными. Этот факт является центральным при разработке алгоритмов для крупных наборов данных, и именно отсюда проистекают идеи лаконичных структур данных и алгоритмов, ориентированных на внешнюю память. В разделе 1.4 мы подробнее рассмотрим причины, по которым доступ к данным на компьютере происходит намного медленнее, чем вычисления.

Картина становится еще сложнее, по мере того как мы расширяем поле зрения, переходя от одного-единственного компьютера. Большинство приложений сегодня представляют собой распределенные и сложные конвейеры данных, в которых тысячи компьютеров обмениваются данными по сетям. Базы данных и кеши распределены, и многочисленные пользователи одновременно добавляют и запрашивают крупные объемы контента. Форматы данных стали разнообразными, многомерными и динамичными. В целях поддержания эффективной работы современные приложения должны быть в состоянии очень быстро реагировать на изменения.

В приложениях по обработке потоков [2] данные фактически пролетают незаметно, без какого-либо промежуточного хранения, и приложению приходится улавливать релевантные признаки данных с такой степенью точности, которая делает их актуальными и полезными, без повторного сканирования. Этот новый контекст требует нового поколения алгоритмов и структур данных, нового набора инструментов разработчика приложений, оптимизированного под решение многих задач, характерных для систем массивных данных. Настоящая книга предназначена научить вас именно этому – основополагающим алгоритмическим методам и структурам данных для разработки масштабируемых приложений.

1.1 Пример

В целях иллюстрации главных тем этой книги давайте рассмотрим следующий пример: вы работаете в медиакомпании над проектом, связанным с комментариями к новостным статьям. Вам предоставили крупное хранилище комментариев со следующими базовыми метаданными:

```
{
    comment-id: 2833908010
    article-id: 779284
    user-id: 9153647
    text: для этого рецепта нужно больше сливочного масла
    views: 14375
    likes: 43
}
```

Вы осматриваете примерно 3 млрд пользовательских комментариев общим объемом 600 Гб. Вы хотели бы получить ряд ответов на вопросы о наборе данных, включая определение наиболее популярных комментариев и статей, классификацию статей в соответствии с темами и распространенными ключевыми словами, встречающимися в комментариях, и т. д. Но сначала необходимо решить задачу о дубликатах, которые накапливались в течение нескольких операций выскабливания данных из интернета, и определить общее число несовпадающих комментариев в наборе данных.

1.1.1 Решение задачи: пример

Для хранения уникальных элементов в структуре данных широко принято создавать словарь ключ-значение, в котором уникальный ИД каждого несовпадающего элемента соотносится с частотой его появления. Словари ключ-значение реализованы во многих библиотеках, таких как `map` в C++, `HashMap` в Java, `dict` в Python и т. д. Словари ключ-значение обычно реализуются в виде сбалансированного дерева двоичного поиска (например, красно-черного дерева в `map` C++) либо, как альтернативный вариант, в виде хеш-таблиц (например, `dict` в Python).

> **Реализации красно-черного дерева и хеш-таблицы в сравнении**
>
> Реализации древовидного словаря, помимо операций поиска/вставки/удаления, которые выполняются за быстрое логарифмическое время, предлагают столь же быстрые операции предшественник/преемник, то есть возможность эффективно просматривать данные взад и вперед, используя лексикографическое упорядочение. Большинству реализаций хеш-таблиц не хватает возможности эффективно выполнять обход элементов в лексикографическом порядке; с другой стороны, реализации хеш-таблиц обеспечивают высокую постоянно-временную производительность наиболее распространенных операций поиска/вставки/удаления.

Ради упрощения нашего примера давайте допустим, что мы работаем с хеш-таблицей Python `dict`. Использование атрибута `comment-id` в качестве ключа и числа появлений этого атрибута в качестве значения поможет нам эффективно устранить дубликаты (см. словарь `(comment-id -> frequency)` в левой части рис. 1.1).

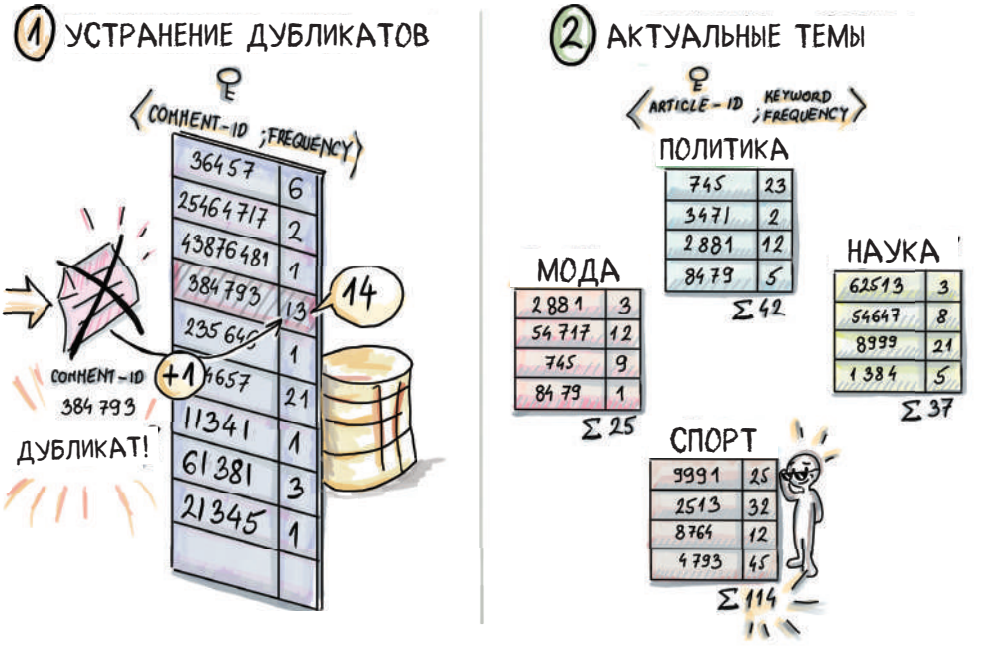


Рисунок 1.1 В данном примере создается хеш-таблица (`comment-id`, `frequency`), которая помогает хранить несовпадающие идентификаторы `comment-id` с их частотами. Входящий идентификатор `comment-id` 384793 в таблице уже содержится, и его частота возрастает. Помимо указанной хеш-таблицы, создаются хеш-таблицы, связанные с темами, в которых подсчитывается число раз, когда ассоциированные ключевые слова появлялись в комментариях к каждой статье (например, в спортивной теме ключевыми словами могут быть футбол, игрок, гол и т. д.). В крупном наборе данных в 3 млрд комментариев для таких структур данных может потребоваться от десятков до ста гигабайт оперативной памяти

Однако если использовать по 8 байт в расчете на пару (4 байта для `comment-id` и 4 байта для `frequency`), то для хранения пар `<comment-id, frequency>` нам может потребоваться до 24 Гб из 3 млрд комментариев. В зависимости от метода, используемого в реализации базовой хеш-таблицы, для служебных операций (с пустыми слотами, указателями и т. д.) структуре данных потребуется в 1.5–2 раза больше занимаемого элементами места, что приблизит нас почти к 40 Гб.

Если мы также хотим классифицировать статьи по определенным интересующим темам, то можем снова использовать словари (возможны и другие методы), создав отдельный словарь для каждой темы (например, спортивной, политической, научной и т. д.), как показано в правой части рис. 1.1. Здесь роль словарей (`article-id -> keyword_frequency`) заключается в подсчете числа появлений ключевых слов, связанных с той или иной темой, во всех комментариях; например, статья с `article-id` 745 содержит 23 ключевых слова в соответствующих комментариях, связанных с политикой. Мы предварительно фильтруем каждый `comment-id`, используя большой

словарь (`comment-id -> frequency`), чтобы учитывать только несовпадающие комментарии. Отдельная таблица такого рода может содержать десятки миллионов записей общим объемом около 1 Гб, и поддержание таких хеш-таблиц, скажем, для 30 тем может стоить до 30 Гб только для данных и примерно 50 Гб в общей сложности.

Будем надеяться, что приведенный выше небольшой пример наглядно демонстрирует, как можно начать с довольно распространенной и наивной задачи и, не успев опомниться, столкнуться с рядом неуклюжих структур данных, которые просто невозможно уместить в памяти.

Возможно, вы подумали: а разве нельзя заранее умножить пару чисел и легко предсказать будущую величину структуры данных? Дело в том, что в реальной жизни это зачастую работает не так. Люди редко начинают строить свои системы с нуля, уже имея в наличии массивные данные. Компании часто начинают с попыток создать работающую систему, а позже становятся жертвами собственного успеха, когда пользовательская база ускоренно вырастает за короткий промежуток времени и старая система, построенная покинувшими компанию разработчиками, должна справляться с этой новой взыскательной рабочей нагрузкой. Чаще всего части системы перестраиваются по мере возникновения необходимости.

Когда число элементов в наборе данных становится большим, каждый дополнительный бит в расчете на элемент увеличивает нагрузку на систему. Распространенные структуры данных, являющиеся хлебом насущным для каждого разработчика программного обеспечения, могут стать слишком большими, чтобы с ними можно было работать эффективно, и нам нужны более лаконичные альтернативы (см. рис. 1.2).

Рисунок 1.2 Наиболее распространенные структуры данных, включая хеш-таблицы, становится трудно хранить и контролировать, имея крупные объемы данных

1.1.2 Решение задачи, дубль два: пошаговый разбор книги

Из-за устрашающих размеров наборов данных мы оказываемся перед выбором. Оказывается, если согласиться на небольшую погрешность ошибки, то можно построить структуру данных, аналогичную по функциональности хеш-таблице, только более компактную. Существует целое семейство лаконичных структур данных[2], и они составляют часть I книги. В этих структурах данных используется малое пространство, чтобы аппроксимировать ответы на следующие ниже распространенные вопросы:

- *принадлежность* – существует ли комментарий/пользователь X?
- *частота* – сколько раз пользователь X оставил комментарий? Какое самое популярное ключевое слово?
- *кардинальное число*[3] – число доподлинно несовпадающих комментариев/пользователей?

Эти структуры данных потребляют гораздо меньше пространства, чтобы обрабатывать набор данных из *n* элементов, чем хеш-таблица (например, 1 байт на каждый элемент или меньше, по сравнению с 8–16 байтами на каждый элемент в хеш-таблице).

Фильтр Блума, который мы обсудим в главе 3, будет использовать в восемь раз меньше пространства, чем хеш-таблица (`comment-id -> frequency`), и будет отвечать на запросы о принадлежности примерно с 2%-ной частотой ложноположительных результатов. В этой вводной главе мы не будем вдаваться в мелкие математические подробности получения этих чисел, но все же стоит подчеркнуть разницу между фильтрами Блума и хеш-таблицами: фильтры Блума не хранят ключи (например, `comment-id`). Фильтры Блума вычисляют хеши из ключей и используют их для модифицирования структуры данных. Следовательно, размер фильтра Блума главным образом зависит от числа вставленных ключей, а не от их размера (или их типа – строковый литерал, малое либо большое целое число).

В главе 4 мы узнаем еще об одной структуре данных, именуемой наброском count-min[4], в которой используется более чем в 24 раза меньше пространства, чем в хеш-таблице (`comment-id -> frequency`), чтобы оценивать частоту каждого идентификатора `comment-id`, демонстрируя небольшое преувеличение частоты в более чем 99 % случаев. Структуру данных наброска count-min также можно использовать для замены хеш-таблиц (`article-id -> keyword_frequency`) и применить около 3 Мб в расчете на тематическую хеш-таблицу, что стоит примерно в 20 раз меньше, чем изначальная схема.

Наконец, структура данных HyperLogLog из главы 5 может оценить кардинальное число множества всего с 12 Кб, демонстрируя ошибку менее чем в 1 % от истинного кардинального числа.

---

[2] Англ. succinct data structure. – *Прим. перев.*

[3] Англ. cardinality; син. мощность множества. – *Прим. перев.*

[4] Англ. count-min sketch; син. эскиз **count-min**; вероятностная структура данных, которая оценивает частоту появления элемента в потоке данных. – *Прим. перев.*

Если в каждой из упомянутых выше структур данных еще больше ослабить требования к точности, то удастся обойтись еще меньшим пространством. Поскольку изначальный набор данных по-прежнему находится на диске, также существует возможность контроля за эпизодической ошибкой, так что мы не останемся с ложноположительными результатами; нам просто придется приложить чуть больше усилий для их верификации.

Данные комментариев в виде потока

Вполне вероятно, что мы столкнемся с задачей о комментариях к новостям и статьям не в виде статического набора данных, а в контексте быстро меняющегося потока событий. Допустим, что событие здесь представляет собой любое изменение набора данных, такое как нажатие кнопки **Нравится** или вставка/удаление комментария либо статьи, и события поступают в систему в реальном времени в виде потоковых данных. В главе 6 вы узнаете о контексте обработки потоковых данных подробнее.

Обратите внимание, что в этой конфигурации тоже можно столкнуться с дубликатами идентификатора `comment-id`, но по другой причине: всякий раз, когда кто-то кликает по кнопке **Нравится** под определенным комментарием, мы получаем событие с тем же `comment-id`, но со скорректированным числом появлений атрибута `likes`. Учитывая ускоренное и круглосуточное поступление событий, в режиме 24/7, и отсутствие возможности хранить их все, для многих представляющих интерес задач можно предложить лишь приближенные решения. В первую очередь мы заинтересованы в реально-временном вычислении базовых статистик (например, среднего числа лайков в расчете на комментарий за последнюю неделю), и, не имея возможности хранить число появлений лайков по каждому комментарию, можно прибегнуть ко взятию случайных выборок.

Мы можем формировать случайную выборку из потока данных по мере их поступления, используя алгоритм формирования выборки Бернулли, который мы рассмотрим в главе 7. В качестве примера можно привести игру «любит – не любит». Если вы когда-либо отщипывали лепестки цветка наугад, то можно сказать, что у вас в руках, скорее всего, оказались лепестки, «отобранные по Бернулли» (не делайте этого на свидании). Указанная схема формирования выборки удобно подходит для использования в контексте однопроходных данных.

Ответы на некоторые более детальные вопросы о данных о комментариях, например сколько лайков нужно поставить комментарию, чтобы он попал в верхние 10 % понравившихся комментариев, также позволят обменивать точность на пространство. Мы можем поддерживать разновидность динамической гистограммы (см. главу 8) всех наблюдавшихся данных в ограниченном, реалистичном пространстве быстрой памяти. Этот набросок, или сводку данных, затем можно использовать для ответа на поисковые запросы о любых квантилях полных данных, но с некоторой ошибкой.

Данные комментариев в базе данных

Наконец, мы можем хранить все данные комментариев в долговременном формате (например, в базе данных на диске / в облаке) и создать поверх них систему, позволяющую быстро вставлять, извлекать и изменять «живые» данные во временной динамике. В таком случае мы отдаем предпочтение точности, а не скорости, поэтому нам удобно хранить тонны данных на диске и извлекать их медленнее, если мы можем гарантировать 100%-ную точность запросов.

Хранение данных в дистанционном хранилище и организация их таким образом, чтобы оно допускало эффективное их извлечение, – это тема алгоритмической парадигмы, именуемой алгоритмами внешней памяти, которую мы начнем изучать в главе 9. Алгоритмы внешней памяти обращаются к наиболее актуальным задачам современных приложений, таким как строительство и реализация механизмов баз данных и их индексов. В нашем конкретном примере с данными комментариев нам необходимо ответить на вопрос, какая система строится: система, содержащая главным образом статические данные, но к которой пользователи постоянно направляют поисковые запросы (то есть оптимизированная под операции чтения), или же система, в которой пользователи очень часто добавляют новые данные и их изменяют, но направляют поисковые запросы лишь эпизодически (то есть оптимизированная под операции записи). Или же, возможно, она будет комбинацией, в которой одинаково важны как быстрые вставки, так и быстрые поисковые запросы (то есть оптимизированная под операции чтения-записи).

Очень немногие инженеры реализуют свои собственные движки хранения данных, но почти все их используют. Для того чтобы грамотно выбирать между различными альтернативами, нужно разбираться в структурах данных, которые лежат в их основе. Компромисс между вставкой и поиском является неотъемлемой частью баз данных, и это отражено в устройстве структур данных, которые работают под управлением MySQL, TokuDB, LevelDB и многих других существующих систем хранения данных. Среди наиболее популярных структур данных для построения баз данных можно назвать B-деревья, B^{ε}-деревья и LSM-деревья, и каждая из них обслуживает разную рабочую нагрузку. Мы обсудим эти разные типы производительности и компромиссы в главе 10. Кроме того, нам может быть интересно решить и другие задачи с данными, размещенными на диске, такие как лексикографическое упорядочение комментариев или упорядочение по числу появлений. Для этого нужен алгоритм сортировки, который будет эффективно сортировать данные в базе данных или в файле на диске. Вы научитесь это делать в последней главе книги, главе 11.

1.2 Структура этой книги

Как уже говорилось в предыдущем разделе, эта книга вращается вокруг трех главных тем и, соответственно, поделена на три части.

Часть I (главы 2–5) посвящена конспективным[5] структурам данных на основе хеша. Эта часть начинается с обзора хеш-таблиц и конкретных методов хеширования, разработанных для использования в условиях массивных данных. Несмотря на то что глава о хешировании запланирована как обзорная, мы предлагаем использовать ее в качестве памятки по хешированию, а также воспользоваться возможностью узнать о современных методах хеширования, разработанных для работы с крупными наборами данных. Глава 2 также служит хорошей подготовкой к главам 3–5, если учесть, что наброски основаны на хешах. Представленные в главах 3–5 структуры данных, такие как фильтры Блума, набросок count-min, массив *HyperLogLog* и их альтернативы, нашли множество применений в базах данных, сетях и т. д.

Часть II (главы 6–8) знакомит с потоками данных. От классических методов, таких как формирование выборки Бернулли и резервуарной выборки, до более изощренных методов, таких как формирование выборки из движущегося окна, мы представляем ряд алгоритмов формирования выборки, подходящих для разных моделей обработки потоковых данных. Созданные выборки затем используются для расчета оценок общих сумм или средних значений и т. д. Мы также вводим алгоритмы вычисления (ансамбля) ε-приближенных квантилей, таких как q-дайжест и t-дайжест.

Часть III (главы 9–11) описывает алгоритмические методы для сценариев, когда данные хранятся на твердотельном накопителе/диске. Сначала мы вводим модель внешней памяти, а затем представляем оптимальные алгоритмы для основополагающих задач, таких как поиск и сортировка, иллюстрируя ключевые алгоритмические приемы, применяемые в этой модели. В указанной части книги также охватываются структуры данных, которые используются в современных базах данных, такие как *B*-деревья, B^{ε}-деревья и LSM-деревья.

1.3 Отличие этой книги от других и ее целевая аудитория

Классическим алгоритмам и структурам данных посвящен ряд замечательных книг, в том числе The Algorithm Design Manual (3rd ed., Skiena, Springer, 2020)[6], Introduction to Algorithms (3rd ed., Cormen, Leiserson, Rivest, Stein, The MIT Press, 2022)[7], Algorithms (4th ed., Sedgewick, Wayne, Addison-Wesley,

[5] Набросок (sketch; син. конспект, резюме, эскиз, скетч) – это компактная сводка определенных аспектов данных, оптимизированная под приближенный ответ на те или иные запросы, используя постоянное или сублинейное пространство. – *Прим. перев.*

[6] Алгоритмы. Руководство по разработке. 3-е изд. – СПб.: БХВ, 2022. – *Прим. перев.*

[7] Алгоритмы. Построение и анализ. – М.: Диалектика, 2020. – *Прим. перев.*

2011)[8] и в качестве вводного и дружественного взгляда на предмет Grokking Algorithms (Bhargava, Manning, 2016)[9]. Алгоритмы и структуры данных для массивных наборов данных пока что медленно, но верно проникают в мейнстримные учебники, однако мир развивается быстро, и мы надеемся, что наша книга станет сборником самых современных алгоритмов и структур данных, которые будут помогать исследователю данных или разработчику справляться с крупными наборами данных на практике.

Данная книга имеет целью предложить хороший баланс теоретических концепций, легко воспринимаемых на интуитивном уровне, практических примеров использования и фрагментов исходного кода на Python. Мы исходим из того, что читатель обладает основополагающими знаниями об алгоритмах и структурах данных, поэтому если вы не изучали базовые алгоритмы и структуры данных, то вам непременно следует ознакомиться с этим материалом, прежде чем приступать к изучению обозначенной темы. Алгоритмы массивных данных – очень обширная тема, и эта книга призвана послужить щадящим введением.

Большинство книг по массивным данным посвящены конкретной технологии, системе или инфраструктуре. Эта книга не ориентирована на конкретную технологию; в ней не принимается никаких допущений о знакомстве читателя с какой-либо конкретной технологией. Напротив, в ней рассматриваются базовые алгоритмы и структуры данных, которые играют важную роль в обеспечении масштабируемости этих систем.

Зачастую в книгах, в которых алгоритмические аспекты массивных данных все же затрагиваются, основное внимание уделяется машинному обучению. Однако в литературе нередко игнорируется важный аспект работы с крупными данными, который конкретно не связан с выводом знаний из данных, а скорее имеет отношение к оперированию размерами данных и их эффективной обработке, какими бы ни были данные. Цель этой книги – восполнить данный пробел.

В ряде замечательных книг рассматриваются специализированные аспекты массивных наборов данных [3]. В настоящей книге мы намерены представить эти разные темы в одном месте, часто цитируя передовые научно-изыскательские и технические работы по соответствующим темам. Наконец, мы надеемся, что эта книга преподнесет продвинутый алгоритмический материал в доступной форме, предоставляя математические концепции, легко воспринимаемые на интуитивном уровне, вместо технических доказательств, которыми характеризуется большинство ресурсов по этому предмету. Иллюстрации играют важную роль в изложении продвинутых технических концепций, и мы надеемся, что они вам понравятся (и благодаря им вы многому научитесь).

Теперь, когда со вступительными замечаниями покончено, давайте обсудим центральный вопрос, который лежит в основе тем этой книги.

---

[8] Алгоритмы на C++. – М.: Диалектика, 2019. – *Прим. перев.*

[9] Грокаем алгоритмы. – СПб.: Питер, 2024. – *Прим. перев.*

1.4 Почему массивные данные представляют трудности для современных систем?

На производительность приложения влияет огромное число параметров, существующих в компьютерах и архитектурах распределенных систем. Среди главных трудностей, с которыми компьютеры сталкиваются при обработке крупных объемов данных, есть те, которые связаны с аппаратным обеспечением и общей архитектурой компьютера. Эта книга не посвящена аппаратному обеспечению, но, разрабатывая эффективные алгоритмы для массивных данных, важно понимать физические ограничения, которые сильно затрудняют передачу данных. В этой главе мы обсудим некоторые из этих главных трудностей, включая большую асимметрию между скоростями центрального процессора и памяти, разными уровнями памяти и компромиссы между скоростью и размером каждого уровня, а также проблему задержки относительно пропускной способности.

1.4.1 Разрыв в производительности центрального процессора и памяти

Первой важной асимметрией, которую мы обсудим, является соотношение скоростей операций центрального процессора и операций доступа к памяти в компьютере, так называемый разрыв в производительности центрального процессора и памяти [4]. На рис. 1.3 показан, начиная с 1980 года, средний разрыв между скоростями доступа к процессорной памяти и доступа к основной памяти (динамической ОЗУ), выраженный в числе запросов к памяти в секунду (величине, обратной задержке).

На интуитивном уровне этот разрыв показывает, что вычисления выполняются намного быстрее, чем доступ к данным. Если застрять на стереотипе, что в центре внимания должна находиться только оптимизация вычислений центрального процессора, то во многих случаях анализ не будет хорошо сочетаться с реальностью.

1.4.2 Иерархия памяти

Помимо разрыва между центральным процессором и памятью, существует встроенная в компьютер иерархия разных типов памяти, которые обладают разными характеристиками. Превалирующий компромисс заключался в наличии, с одной стороны, малой, но быстрой (и дорогой) памяти, и, с другой стороны, большой, но медленной (и дешевой) памяти. Как показано на рис. 1.4, начиная с самого малого и быстрого, компьютерная иерархия обычно содержит следующие уровни: регистры, кеш-память 1-го уровня, кеш-память 2-го уровня, кеш-память 3-го уровня, основная память, твердотельный накопитель (SSD) и/или жесткий диск (HDD). Последние две являются долговременной (энергонезависимой) памятью, то есть данные сохраняются после выключения компьютера и, следовательно, подходят для хранения.

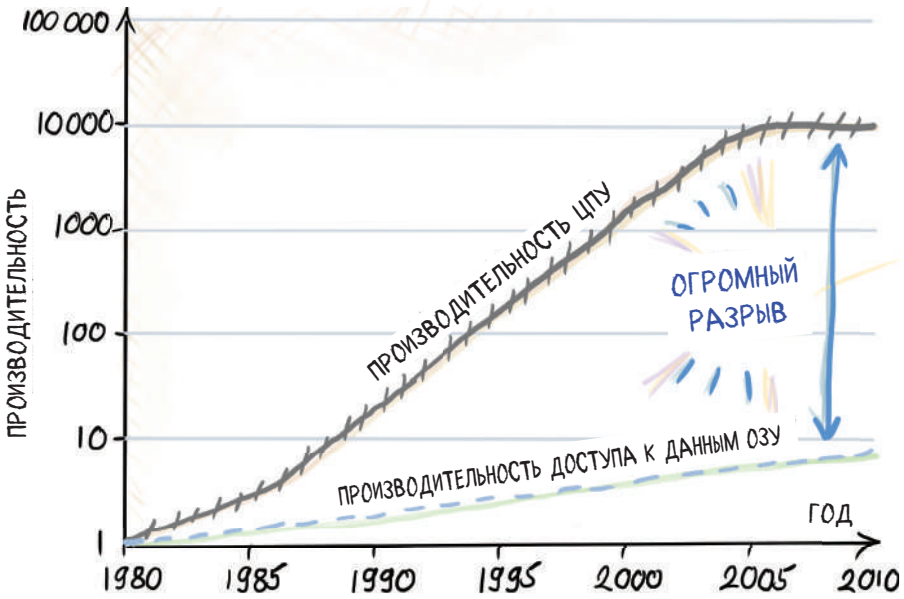


Рисунок 1.3 График разрыва в производительности центрального процессора и памяти, заимствованный из архитектуры вычислительной системы Hennessy & Patterson. На графике показан увеличивающийся разрыв между скоростями доступа к памяти центрального процессора и оперативной памяти (среднее число обращений к памяти в секунду во временной динамике). Вертикальная ось находится в логарифмической шкале. Процессоры демонстрировали улучшение примерно в 1.5 раза в год вплоть до 2005 года, а улучшение доступа к основной памяти составляло всего около 1.1 раза в год. С 2005 года ускорение центрального процессора несколько снизилось, но оно нивелируется за счет использования нескольких ядер и параллелизма

На рис. 1.4 мы видим времена доступа и пропускные способности каждого уровня памяти в примере архитектуры [5]. Цифры варьируются в зависимости от архитектуры и более полезны, если рассматривать их с точки зрения соотношений между разными временами доступа, а не конкретными значениями. Например, извлечение фрагмента данных из кеша происходит примерно в 1 млн раз быстрее, чем с диска.

Жесткий диск и магнитная головка, немногие из оставшихся механических частей компьютера, работают во многом подобно проигрывателю грампластинок. Позиционирование магнитной головки на нужный трек – это времязатратная часть доступа к дисковым данным. После того как головка спозиционирована на нужном треке, передача данных может быть очень быстрой, в зависимости от скорости вращения диска.

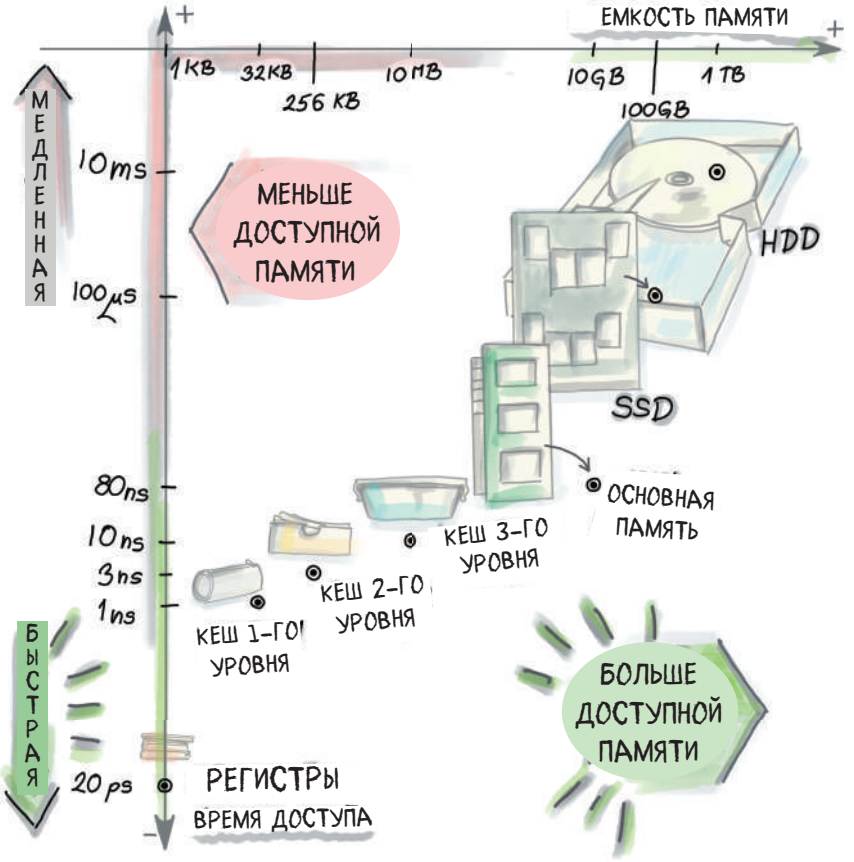


Рисунок 1.4 Разные типы памяти в компьютере. Начиная с регистров в левом нижнем углу, которые поразительно быстры, но в то же время очень малоемкостны, мы продвигаемся вверх (при этом скорость памяти становится медленнее) и вправо (емкость памяти увеличивается) с кешем 1-го уровня, кешем 2-го уровня, кешем 3-го уровня и основной памятью, вплоть до SSD и/или жесткого диска. Смешивание разных элементов памяти в одном компьютере создает иллюзию наличия как скорости, так и емкости хранения, поскольку каждый уровень служит кешем для следующего, более крупного

1.4.3 Задержка относительно пропускной способности

Аналогичное явление наблюдается там, где «задержка отстает от пропускной способности» [6], и характерно для разных типов памяти. За последние несколько десятилетий пропускная способность в различных системах, начиная от микропроцессоров и заканчивая оперативной памятью, жестким диском и сетью, значительно улучшилась, но задержка не улучшалась с той же скоростью, хотя задержка является важной мерой во

многих сценариях, где обычное поведение пользователя предусматривает множество малых произвольных обращений, в отличие от одного большого последовательного обращения.

В целях компенсации стоимости дорогостоящего первоначального вызова передача данных между разными уровнями памяти осуществляется порциями из нескольких элементов. Эти порции называются строками кеша, страницами или блоками, в зависимости от уровня памяти, с которым мы работаем, и их размер пропорционален размеру соответствующего уровня памяти; для кеша они находятся в диапазоне 8–64 байт, а для дисковых блоков они могут достигать 1 Мб [7]. Благодаря концепции под названием пространственная локальность, когда мы ожидаем, что программа будет обращаться к ячейкам памяти, которые находятся в непосредственной близости друг от друга и близки по времени, передача данных последовательными блоками эффективно обеспечивает предварительную доставку элементов, которые нам, скорее всего, понадобятся в ближайшем будущем.

1.4.4 Как насчет распределенных систем?

Сегодня большинство приложений работают на нескольких компьютерах, и отправка данных с одного компьютера на другой приводит к еще одному уровню задержки. Передача данных между компьютерами может длиться от сотен миллисекунд до пары секунд, в зависимости от нагрузки на систему (например, числа пользователей, обращающихся к одному и тому же приложению), числа переходов к месту назначения и других деталей архитектуры (см. рис. 1.5).

1.5 Конструирование алгоритмов с учетом аппаратного обеспечения

Рассмотрев некоторые важнейшие аспекты архитектуры современных компьютеров, можно сделать первый важный вывод: хотя технологии постоянно совершенствуются (например, твердотельные накопители являются относительным новшеством и у них нет многих проблем, присущих жестким дискам), некоторые проблемы, такие как компромисс между скоростью и размером элементов памяти, в ближайшее время никуда не исчезнут. Отчасти причина тому чисто физическая: для хранения большого объема данных требуется много пространства, а скорость света устанавливает физический предел скорости, с которой данные могут передаваться из одной части компьютера в другую или из одной части сети в другую. Распространив это на сеть компьютеров, можно процитировать пример [8], показывающий, что для двух компьютеров, находящихся на расстоянии 300 м друг от друга, нижний физический предел обмена данными составит 1 микросекунду.

Рисунок 1.5 Время доступа к облаку может быть большим из-за нагрузки на сеть и сложной инфраструктуры. Доступ к облаку может занимать сотни миллисекунд или даже секунды. Его можно рассматривать как еще один уровень памяти, который еще больше и медленнее, чем жесткий диск. Повышение производительности облачных приложений бывает затруднено еще и потому, что время доступа или записи данных в облаке непредсказуемо

Следовательно, возникает потребность в конструировании алгоритмов, которые могли бы обходить аппаратные ограничения. Разработка лаконичных структур данных (или формирование выборок данных), умещающихся в малых и быстрых уровнях памяти, помогает избегать дорогостоящего поиска на диске. Другими словами, сокращение пространства экономит время.

Тем не менее во многих приложениях по-прежнему приходится работать с данными на диске. И конструирование алгоритмов с оптимизированными схемами доступа к диску и механизмами кеширования, обеспечивающими наименьшее число передач данных из памяти, здесь приобретает особую важность, и это, в свою очередь, связано с размещением и организацией данных на диске (к примеру, в реляционной базе данных). Дисковые алгоритмы предпочитают плавное сканирование диска произвольному и скачкообразному; благодаря этому мы получаем возможность использовать хорошую пропускную способность и избегать низкой задержки, поэтому одним из важных направлений является преобразование алгоритма, кото-

рый выполняет много произвольных операций чтения/записи, в алгоритм, который выполняет последовательные операции чтения/записи. В этой книге вы увидите способы преобразования классических алгоритмов и конструирования новых с учетом пространственных ограничений.

Однако также важно учитывать, что современные системы имеют множество метрик результативности, отличных от масштабируемости: безопасность, доступность, техническая сопроводимость и т. д. Под капотом реальных производственных систем нужны эффективные структура данных и алгоритм, но с большими «танцами с бубнами» поверх них, чтобы все остальное работало на их потребителей (см. рис. 1.6). Однако при постоянно растущих объемах данных конструирование эффективных структур данных и алгоритмов стало еще важнее, чем когда-либо прежде, и будем надеяться, что на следующих далее страницах вы узнаете, как именно это делать.

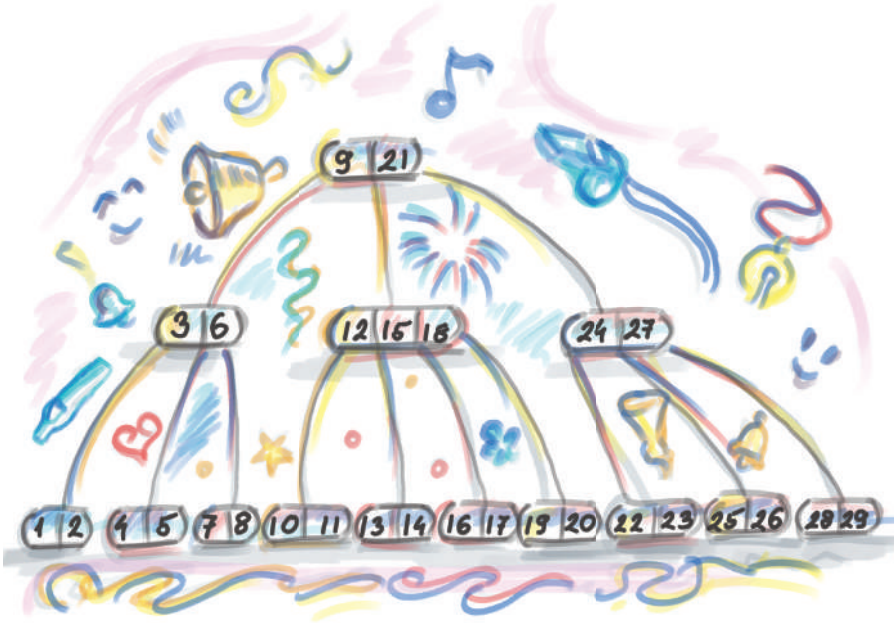


Рисунок 1.6 Эффективная структура данных с «танцами с бубнами»

Резюме

- Современные приложения генерируют и обрабатывают крупные объемы данных на повышенных скоростях. Традиционные структуры данных, такие как словари ключ-значение, могут становиться слишком большими, чтобы умещаться в оперативную память, что может приводить к зависанию приложения из-за узкого места операций ввода-вывода.

- Для эффективной обработки крупных наборов данных можно конструировать пространственно-эффективные наброски на основе хешей, собирать реально-временную аналитику с помощью случайных выборок и аппроксимировать статистику или более эффективно работать с данными на диске и в других дистанционных хранилищах.
- Эта книга служит естественным продолжением книги/курса по базовым алгоритмам и структурам данных, поскольку она учит преобразовывать основополагающие алгоритмы и структуры данных в алгоритмы и структуры данных, которые хорошо масштабируются на крупные наборы данных.
- Ключевые причины, по которым крупные данные являются серьезной проблемой для современных компьютеров и систем, заключаются в том, что скорости центрального процессора (и многопроцессорной системы) повышаются гораздо быстрее, чем скорости памяти, а компромисс между скоростью и размером разных типов памяти в компьютере, а также феномен задержки относительно пропускной способности приводят к тому, что приложения обрабатывают данные с меньшей скоростью, чем выполняют вычисления. Эти тренды в ближайшее время вряд ли изменятся, поэтому важность алгоритмов и структур данных, которые решают проблемы стоимости операций ввода-вывода и пространства, со временем будет только возрастать.
- В приложениях с интенсивным использованием данных оптимизация пространства означает оптимизацию времени.

Часть I

Наброски на основе хеша

В следующих нескольких главах мы проведем разведывательный анализ вероятностных лаконичных структур данных. Мы увидим, как по мере роста объема данных все труднее становится решать простые задачи из мира обычных алгоритмов, такие как оценивание частоты, запросы на принадлежность и задача о числе несовпадающих элементов, и классические структуры данных неизбежно начинают выплескиваться через край оперативной памяти. Мы обратимся к коллекции структур данных, которые помогают решать те же задачи, только занимая гораздо меньше пространства. В чем же подвох? Эти структуры данных не всегда будут давать 100%-ную точность. Однако есть и хорошие новости – частоты ошибки зачастую невелики и в значительной степени компенсируются крупными выигрышами в хранении структур данных. Структуры данных, представленные в части I, включают фильтры Блума, порционные фильтры, набросок count-min, алгоритм/структуру данных HyperLogLog и несколько компактных вариантов хеш-таблиц. Эти структуры данных легко конфигурируются под желаемую частоту ошибки и в этом смысле обладают высокой универсальностью. Следующие несколько глав будут всецело посвящены втискиванию максимальной функциональности в наименьший объем оперативной памяти, и каждый бит будет иметь значение. Но сначала мы проведем обзор хеш-таблиц и хеширования, которые послужат строительными блоками многих будущих структур данных.

Глава 2

Обзор хеш-таблиц и современного хеширования

Эта глава охватывает следующие ниже темы:

- обзор словарей и причин, по которым хеширование получило широкое распространение в современных системах;
- памятка по базовым методам урегулирования коллизий;
- обследование эффективности кеширования в хеш-таблицах;
- использование хеш-таблиц в распределенных системах и согласованное хеширование;
- изучение принципа работы согласованного хеширования в одноранговых сетях.

Мы начинаем с темы хеширования по ряду причин. Во-первых, классические хеш-таблицы оказались незаменимыми в современных системах, из-за чего найти систему, которая их не использует, сложнее, чем ту, которая их использует. Во-вторых, в последнее время было проведено много инновационных работ, направленных на решение алгоритмических проблем, возникающих при росте хеш-таблиц до размеров массивных данных, таких как эффективное изменение размера, компактное представление и пространственно-экономные приемы. Хеширование со временем было адаптировано в том же русле под использование в массивных одноранговых системах, в которых хеш-таблица разбивается между серверами; здесь ключевая трудность состоит в отведении ресурсов серверам и нагрузочной балансировки ресурсов, по мере того как серверы динамически присоединяются к сети и покидают ее. Наконец, мы начинаем с хеширования, поскольку оно формирует основу всех лаконичных структур данных, с которыми мы знакомимся в части I книги.

Помимо основ работы хеш-таблиц, в этой главе мы покажем примеры хеширования в современных приложениях, таких как устранение дубликатов и обнаружение плагиата. В рамках обсуждения компромиссов при конструировании хеш-таблиц мы коснемся темы реализации словарей в

языке Python. В разделе 2.8 обсуждается метод согласованного хеширования, используемый для реализации распределенных хеш-таблиц. В этом разделе представлены примеры исходного кода на языке Python, с которыми можно поэкспериментировать и поиграть, чтобы лучше понять реализацию хеш-таблиц в распределенной и динамической многосерверной среде. Последняя часть раздела, посвященного согласованному хешированию, содержит упражнения по программированию для читателя, которому нравится принимать вызов. Если вы чувствуете себя удобно во всем, что связано с классическим хешированием, то советуем перейти к разделу 2.8, а если вы знакомы с согласованным хешированием, то пролистать до главы 3.

2.1 Хеширование повсюду

Хеширование – один из тех предметов, которому, вероятно, всегда будет не хватать внимания, независимо от количества времени, отводимого данной теме в рамках курсов программирования, структур данных и алгоритмов. Хеш-таблицы и хеш-функции есть практически везде. В качестве иллюстрации рассмотрим процесс написания электронного письма (см. рис. 2.1–2.4). При входе в учетную запись электронной почты введенный пароль сначала хешируется, и хеш сверяется с базой данных, чтобы подтвердить совпадение.

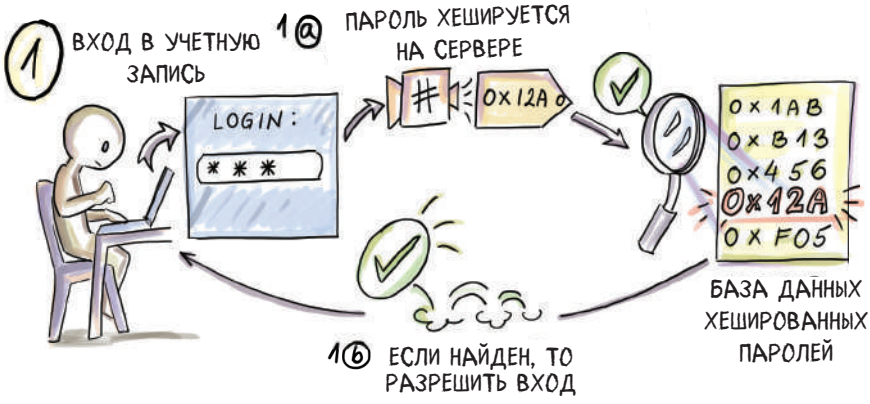


Рисунок 2.1 Вход в учетную запись электронной почты и хеширование

При написании электронного письма подпрограмма-орфокорректор использует хеширование, чтобы проверить существование данного слова в словаре.

При отправке электронного письма нередко IP-адреса пары отправитель–получатель хешируются, чтобы определять промежуточный сервер, на который пакет должен быть направлен, дабы эффективно сбалансировать нагрузку на трафик.

Рисунок 2.2 Проверка орфографии и хеширование

Рисунок 2.3 Сетевые пакеты и хеширование

Наконец, когда электронное письмо прибывает к получателю, содержимое электронного письма иногда хешируется спам-фильтрами, чтобы найти слова, похожие на спам, и отфильтровать возможный спам.

Рисунок 2.4 Спам-фильтры и хеширование

Мы готовы поспорить, что во всех местах, где важна безопасность, и во всех местах, где важна скорость поиска, вы обязательно обнаружите хеширование данных.

Эта глава посвящена как хешированию, так и хеш-таблицам, и иногда изложение в ней неожиданно переключается туда и обратно. Очевидно, что это не одно и то же, но хеширование будет рассматриваться в меньшей степени в контексте криптографии и в большей степени в контексте использования в хеш-таблице – или, в следующих главах, в некоторых других

структурах данных. Хеш-таблицы получили такое же широкое распространение, как и хеширование, и программисты используют их каждый день (например, при построении таблиц ключ-значение), зачастую не зная, что под ними находится хеш-таблица.

Если мы хотим выяснить причину, по которой хеш-таблицы получили столь широкое распространение, то нужно сравнить их с другими структурами данных и посмотреть, насколько хорошо в различных структурах данных реализовано то, что мы называем словарем, – абстрактный тип данных, выполняющий операции поиска, вставки и удаления.

2.2 Ускоренный курс по структурам данных

Многие структуры данных могут выполнять роль словаря, но разные структуры данных демонстрируют разные компромиссы относительно производительности и, таким образом, годятся для разных сценариев использования. Например, рассмотрим обычный несортированный массив. Эта довольно простая структура данных обеспечивает идеальную постоянно-временную[10] производительность на вставках (O(1)) по мере добавления новых элементов в журнал. Однако поиск в наихудшем случае требует полного линейного сканирования данных (O(n)). Несортированный массив может хорошо служить реализацией словаря в приложениях, в которых нужны чрезвычайно быстрые вставки и в которых поиск выполняется крайне редко[11].

Сортированные массивы позволяют выполнять быстрый поиск за логарифмическое время с помощью двоичного поиска (O(log n)), который при разных размерах массивов выполняется практически за постоянное время (логарифм с основанием 2 из 1 млрд равен менее 30 сравнениям). Однако за поддержание сортированного порядка при вставке или удалении приходится расплачиваться и в наихудшем случае перемещаться по линейному числу элементов (O(n)). Линейно-временные операции означают, что в течение одной операции нужно посещать примерно каждый элемент, что в большинстве сценариев является непреодолимой стоимостью.

Связные списки, в отличие от сортированных массивов, позволяют вставлять элементы за постоянное время за счет вставки в голову списка. Удалять можно из любого места списка за постоянное время (O(1)) путем переустановки нескольких указателей, при условии что были локализованы позиции вставки/удаления. Односвязному списку приходится уделять больше внимания, так как при удалении нужно предоставлять указатель на позицию перед удаляемым элементом. Единственным способом локализации этой позиции является обход связного списка, следуя по его указателям, даже если связный список был отсортирован, что возвращает нас

[10] Англ. constant-time; означает, что независимо от размера предоставляемых входных данных временная сложность алгоритма остается неизменной. – *Прим. перев.*

[11] Если мы гарантированно никогда не будем нуждаться в поиске, то имеется даже более оптимальный способ «реализовать» вставки: ничего не делать.

к линейному времени. С какой стороны ни посмотри, в простых линейных структурах, таких как массивы и связные списки, есть по меньшей мере одна операция, которая стоит O(n), и чтобы ее избежать, нужно вырваться из этой линейной структуры.

Операции над словарем в *сбалансированных деревьях двоичного поиска* зависят от глубины дерева, и в них используются различные механизмы балансирования (АВЛ-дерево, красно-черное дерево и т. д.), которые поддерживают глубину дерева на уровне O(log n). Соответственно, все операции вставки, поиска и удаления в наихудшем случае выполняются за логарифмическое время. Как и в случае двоичного поиска, для многих размеров деревьев разница в производительности между постоянным и логарифмическим временем невелика. В части скорости логарифмическое время гораздо ближе к постоянному, чем к линейному, поэтому возможность выполнять все операции со словарем за это гарантированное количество времени должна нас радовать.

В дополнение к этому в сбалансированных деревьях двоичного поиска поддерживается сортированный порядок элементов, что делает их отличным вариантом для выполнения быстрых диапазонных запросов, а также запросов на получение предшествующих и последующих элементов. Сбалансированные деревья двоичного поиска, несомненно, являются оптимальным вариантом для словаря, если сравнивать все структуры данных, которые работают на основе сравнения элементов (<, >, =).

Однако мы не ограничены построением структур данных только на сравнениях; компьютеры способны выполнять множество других операций, включая побитовый сдвиг, арифметические и другие операции, и все это очень умело используется хеш-функциями, для того чтобы вырваться из логарифмической границы.

Предельным преимуществом хеш-таблиц и хеширования является то, что они сокращают стоимости оперирования над словарем до O(1) на всех операциях. Если вы думаете, что это слишком хорошо, чтобы быть правдой, то в какой-то степени вы правы: в отличие от упомянутых до сих пор границ, где время выполнения гарантируется (то есть в наихудшем случае), постоянное время выполнения в хеш-таблицах – это ожидаемая величина. Наихудший случай все еще может быть таким же плохим, как и линейное время O(n), но при хорошем устройстве хеш-таблицы мы почти всегда будем избегать подобных случаев.

Таким образом, даже несмотря на то, что наихудший случай при поиске в хеш-таблице такой же, как и в несортированном массиве, в случае хеш-таблицы событие O(n) почти никогда не произойдет, тогда как в случае массива оно будет происходить довольно систематически.

Причина заключается в следующем: в хеш-таблице хорошая хеш-функция беспорядочно перемешивает входной элемент и, основываясь на этом перемешанном результате, отправляет элемент в некую корзину хеш-таблицы, в которой его можно найти позже. Слово хеш происходит от английского слова *hash*, а то, в свою очередь, от французского *hachis*,

часто используемого для описания вида блюда, в котором мясо рубится на множество маленьких кусочков разного размера (также связано со словом *hatchet* – топорик для рубки мяса). Поскольку в среднем разные элементы будут измельчаться на разные кусочки, они обычно распределяются по разным корзинам хеш-таблицы, в силу чего обеспечивается быстрый поиск, поскольку ни в одной конкретной корзине не будет слишком много элементов. Операция поиска будет измельчать запрашиваемый элемент и заглядывать непосредственно в соответствующую корзину. Однако существует возможность, что хеш-функция измельчит очень разные входные элементы в одно и то же число и отправит их все в одну корзину. В этом случае измельчение не помогло, и нам нужно будет просмотреть все элементы в корзине, чтобы проверить наличие запрашиваемого элемента. Это чрезвычайно редкий случай, и когда он происходит, можно применить иную хеш-функцию, предназначенную для такого конкретного входного элемента.

С другой стороны, хеш-таблицы плохо подходят для всех приложений, в которых важно поддерживать упорядоченность данных. Естественным следствием измельчения данных является то, что порядок элементов не сохраняется. Проблема возникает в основном в базах данных, где для ответа на диапазонный запрос требуется навигация по сортированным элементам; например, чтобы составить список всех сотрудников в возрасте от 35 до 56 лет или отыскать все точки на координате x от 3 до 45 в пространственной базе данных. Хеш-таблицы наиболее полезны в базе данных при поиске точного совпадения. Вместе с тем хеширование можно использовать и для ответа на вопросы о сходстве (например, в обнаружении плагиата), как мы увидим в сценариях из следующего далее раздела. В табл. 2.1 сравниваются наиболее распространенные структуры данных.

Таблица 2.1 Сводная таблица сравнения производительности разных структур данных для операций со словарем. Несортированные массивы хорошо работают в качестве журналов данных. Сортированные массивы хорошо подходят для поиска в статическом наборе данных. Связные списки хороши для быстрого удаления, когда указана нужная позиция в списке. Сбалансированные деревья двоичного поиска быстры и универсальны по части разных операций и гарантируют высокую производительность в наихудшем случае. Операции извлечения предшественника/преемника в сбалансированных деревьях двоичного поиска выполняются за постоянное время, если указана позиция элемента, предшественника/преемника которого мы ищем; в противном случае оно будет логарифмическим. Хеш-таблицы являются самыми быстрыми в смысле ожидаемого времени выполнения. Однако их способность выполнять обход в сортированном порядке не так хороша, как у сбалансированных деревьев двоичного поиска

	Поиск	**Вставка**	**Удаление**	**Предшественник/ преемник**
Несортированный массив	O(n)	O(1)	O(n)	O(n)
Сортированный массив	O(log n)	O(n)	O(n)	O(1)
Связный список	O(n)	O(1)	O(1)*	O(n)

	Поиск	Вставка	Удаление	Предшественник/ преемник
Сбалансированное дерево двоичного поиска	O(log *n*)	O(log *n*)	O(log *n*)	O(1)
Хеш-таблица	O(1) (ожидаемое)	O(1) (ожидаемое)	O(1) (ожидаемое)	O(*n*)

2.3 Сценарии использования в современных системах

Куда ни глянь, везде можно найти множество применений хеширования. Вот два из них, которые нам особенно нравятся.

2.3.1 Дедупликация в программных решениях по резервному копированию/хранению данных

Многие компании, такие как Dropbox и Dell EMC Data Domain Storage Systems, занимаются хранением крупных объемов пользовательских данных путем частой фиксации снимков и резервных копий данных. Клиентами этих компаний нередко являются крупные корпорации, которые хранят огромные объемы данных, и если снимки делаются достаточно часто (скажем, каждые 24 ч), то бóльшая часть данных между поочередными снимками остается неизменной. В этом случае важно быстро находить измененные части и сохранять только их, тем самым экономя время и пространство при сохранении новой копии. Для этого нужно уметь эффективно выявлять дублированный контент.

Процесс устранения дубликатов называется дедупликацией, и в большинстве его современных реализаций используется хеширование. Например, рассмотрим систему дедупликации ChunkStash [1], специально разработанную под обеспечение высокой пропускной способности с использованием флеш-памяти. В ChunkStash файлы разбиваются на малые блоки фиксированного размера (к примеру, по 8 Кб), и каждый блок хешируется в 20-байтовый отпечаток SHA-1; если отпечаток уже присутствует, то указывается только существующий отпечаток. Если отпечаток – новый, то можно допустить, что блок тоже новый, и одновременно сохраняется блок в хранилище данных и отпечаток в хеш-таблице с указателем на позицию соответствующего блока в хранилище данных (см. рис. 2.5).

Разбивка файлов на блоки помогает выявлять неполные дубликаты[12], когда в крупный файл были внесены небольшие правки.

[12] Англ. near-duplicate; син. почти дубликаты. – *Прим. перев.*

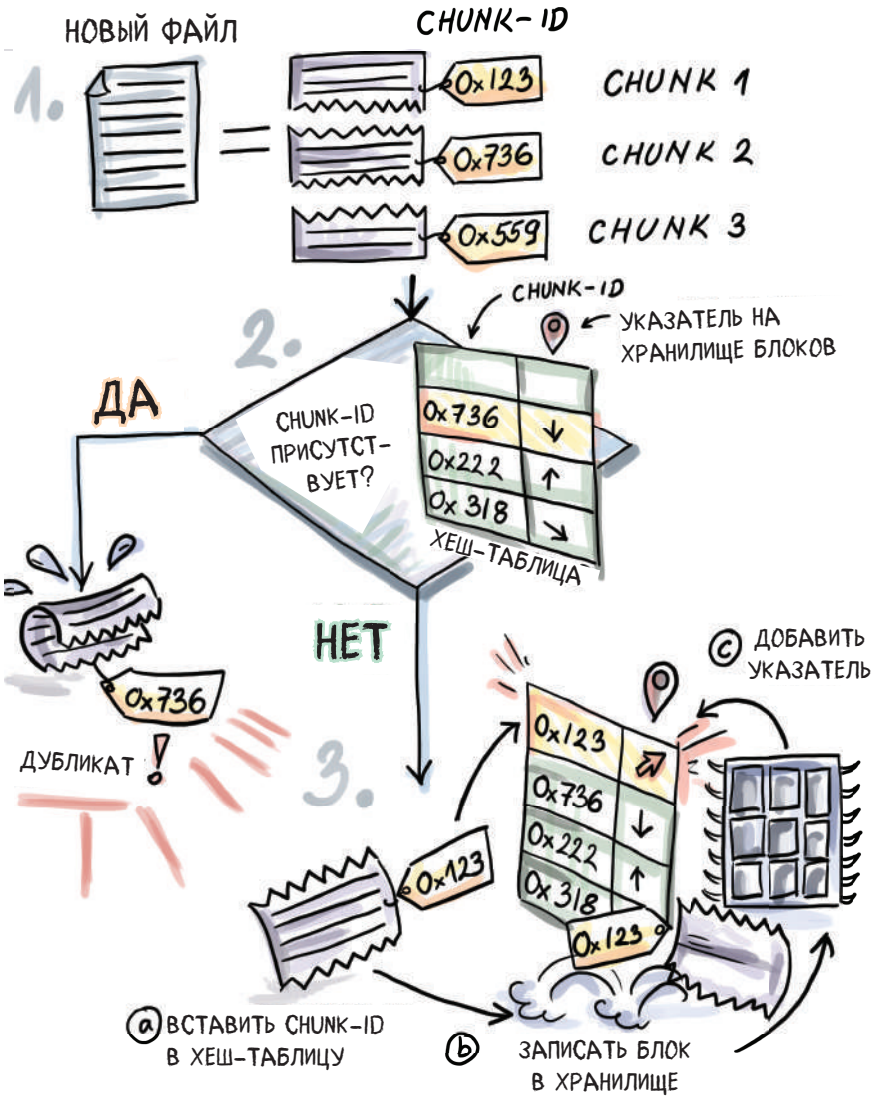


Рисунок 2.5 Процесс дедупликации в программных решениях по резервному копированию/хранению данных. По прибытии нового файла он разбивается на малые блоки. В нашем примере файл разбит на три блока, и каждый блок хешируется (например, блок 1 имеет `chunk-id 0x123`, а блок 2 имеет `chunk-id 0x736`). Идентификатор `chunk-id 0x123` в хеш-таблице не найден. Для этого конкретного `chunk-id` создается новая запись, а сам блок сохраняется. Идентификатор `chunk-id 0x736`, будучи найденным в хеш-таблице, расценивается как дубликат и не сохраняется

В этом процессе гораздо больше тонкостей, чем мы показываем. Например, при записи нового блока во флеш-хранилище блоки сначала накапливаются в резидентном буфере операций записи, а после заполнения

буфера он одним махом сбрасывается во флеш-память. Это делается во избежание повторных малых правок на одной и той же странице – такие операции обходятся особенно дорого во флеш-памяти. Но давайте пока останемся в резидентной области; эффективная буферизация и запись на диск получат больше внимания в части III.

2.3.2 Обнаружение плагиата с помощью идентификации цифровых отпечатков на основе меры MOSS и алгоритма Рабина–Карпа

Мера подобия программного обеспечения (Measure of Software Similarity, аббр. MOSS) – это служба обнаружения плагиата, в основном используемая для выявления плагиата в заданиях по программированию. Одна из главных алгоритмических идей метода MOSS [2] представляет собой вариант алгоритма сопоставления строк Рабина–Карпа [3], который основан на идентификации k-граммных цифровых отпечатков (k-грамма – это сплошная подстрока длины k). Давайте сначала рассмотрим алгоритм.

Имея строку t, представляющую большой текст, и строку p, представляющую шаблон меньшего размера, задача о сопоставлении строк состоит в ответе на вопрос, существует ли вхождение p в t. Алгоритмам сопоставления строк посвящена обширная литература, и бóльшая их часть ориентирована на сравнение подстрок между p и t, но алгоритм Рабина–Карпа сравнивает хеши подстрок и делает это умным образом. Он чрезвычайно хорошо работает на практике, и его высокая производительность (что на данный момент не должно вас удивлять) частично обусловлена хешированием.

В частности, алгоритм выполняет проверку подстрок на совпадение в посимвольном режиме (только тогда, когда хеши подстрок совпадают). В наихудшем случае мы получим много ложных совпадений из-за коллизий хешей, когда две разные подстроки имеют одинаковый хеш, но подстроки отличаются. В этом случае общее время выполнения составляет $O(|t||p|)$, как и в алгоритме сопоставления строк методом грубой силы. Но в большинстве ситуаций, когда истинных совпадений не так много и когда имеется хорошая хеш-функция, алгоритм проносится по t с молниеносной скоростью (то есть работает за линейное время). Ложные совпадения могут способствовать производительности наихудшего случая, но, как обсуждалось ранее, хорошая хеш-функция будет обеспечивать, чтобы это происходило не столь часто. Пример работы алгоритма приведен на рис. 2.6.

Время вычисления хеша зависит от размера подстроки (хорошая хеш-функция должна учитывать все символы), поэтому само по себе хеширование не ускоряет алгоритм. Однако в алгоритме Рабина–Карпа используются скользящие хеши, где при наличии хеша k-граммы $t[j, \ldots, j + k - 1]$ вычисление хеша для k-граммы, сдвигаемой на одну позицию вправо, $t[j + 1, \ldots, j + k]$, занимает исключительно постоянное время (см. рис. 2.7). Это можно сделать, если скользящая хеш-функция такова, что она позво-

ляет некоторым образом «вычитать» первый символ из первой *k*-граммы и «прибавлять» последний символ из второй *k*-граммы (очень простым примером такого скользящего хеша является функция суммы ASCII-значений символов в строке).

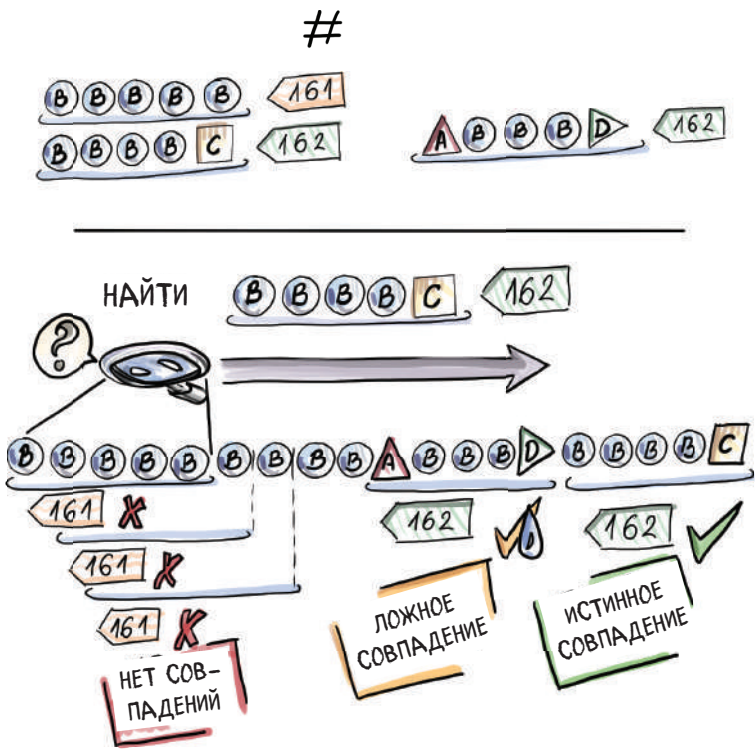


Рисунок 2.6 Пример алгоритма идентификации цифровых отпечатков Рабина–Карпа. Мы ищем шаблон `p=BBBBC` в большей строке `t=BBBBBBBBBBBBBBBBBC`. Хеш `BBBBC` равен 162 и не совпадает с хешем 161 для `BBBBB`, который встречается в начале длинной строки. По мере сдвига вправо мы неоднократно сталкиваемся с несовпадением хеша до тех пор, пока не получим подстроку `ABBBD` с хешем 162. Затем проверяем подстроки и устанавливаем ложное совпадение. В самом конце строки мы снова встречаем совпадение хеша в `BBBBC` и после проверки подстрок сообщаем об истинном совпадении

Алгоритм Рабина–Карпа может использоваться в простой форме для сравнения двух заданий на предмет плагиата путем разбивки файлов на более мелкие блоки и идентификации их цифровых отпечатков. Однако в MOSS нас интересует крупная группа предоставленных заданий и все потенциальные случаи плагиата. Это означает сравнение всех со всеми и непрактичный алгоритм с квадратичным временем выполнения. Борясь с квадратичным временем, метод MOSS отбирает для сравнения небольшое число отпечатков в качестве представителей каждого файла. Приложение строит инвертированный индекс – как средство соотнесения отпечатка с

его позицией в документах, где он встречается. Из этого соотнесения можно, в свою очередь, вычислить список похожих документов. Обратите внимание, что в списке будут только те документы, которые имеют совпадения, поэтому мы избегаем слепого сравнения всех со всеми.

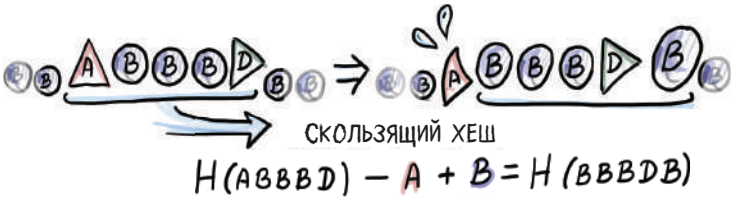


Рисунок 2.7 Скользящий хеш. Вычисление хеша для всех, кроме первой подстроки *t*, является постоянно-временной операцией. Например, для BBBDB нужно было «вычесть» A и «прибавить» B к ABBBD

Выбору набора репрезентативных цифровых отпечатков для документа посвящен целый ряд методов. В MOSS задействуется один из них: для каждого окна с поочередными символами файла (например, длина окна может составлять 50 символов) выбирается минимальный хеш из *k*-грамм, принадлежащих этому окну. Наличие одного отпечатка на окно полезно тем, что, помимо прочего, помогает избегать пропуска больших поочередных/непрерывных совпадений.

2.4 O(1): что в этом такого?

Ознакомившись с несколькими вариантами использования хеширования в словарях, давайте снимем с них еще один слой шелухи. Прочтя обо всех компромиссах между разными аспектами производительности в разделе 2.2, вы, возможно, задаетесь вопросом, почему так трудно сконструировать идеальную структуру данных – такую, которая выполняет операции поиска, вставки и удаления всего за O(1) в наихудшем случае. И конкретнее, вы, возможно, хотите узнать, можно ли сконструировать хеш-таблицу, которая могла бы гарантировать постоянно-временные операции. Это вопрос о структурах данных из разряда «почему нельзя иметь все это сразу?». Хотя в целом это невозможно, существуют особые ситуации, которые позволяют это делать.

Например, допустим, у вас есть набор данных; для простоты предположим, что у вас набор из 100 чисел и хеш-таблица такого же размера. Поскольку у вас статический набор данных, вы можете придумать собственную хеш-функцию, которая будет обеспечивать, чтобы каждый элемент попадал в разную корзину хеш-таблицы, хеш-функцию, настроенную на этот конкретный набор данных. За счет этого будет обеспечена идеальная производительность.

Еще один подобный сценарий – если имеется набор целых положительных чисел и известен максимум числа (назовем его *M*). Если *M* не слишком

велик, то можно сконструировать хеш-таблицу размера M и помещать каждое число в корзину, пронумерованную по его значению. Опять же, при условии отсутствия дубликатов мы получаем по одному элементу на корзину, что приводит к постоянно-временной производительности на операциях вставки, поиска и удаления.

Но это особые ситуации, и, вообще говоря, ситуации, когда мы знаем данные заранее или имеем входные данные очень специфического вида, – это больше, чем можно ожидать в большинстве случаев.

Главная трудность правильного хеширования заключается в том, что хеш-функции должны обеспечивать соотнесение каждого потенциального элемента с соответствующей корзиной хеш-таблицы. Множество, которое представляет все потенциальные элементы, независимо от типа данных, с которыми мы имеем дело, вероятно, крайне велико, намного больше размера фактического набора данных и, следовательно, числа корзин хеш-таблицы. Мы будем называть такое множество всех потенциальных элементов универсальным множеством U, размер нашего набора данных – n, а размер хеш-таблицы – m.

Значения n и m примерно пропорциональны. Другими словами, если вы собираетесь хранить 1 млн элементов, то, вероятно, захотите запланировать хеш-таблицу аналогичного размера. В зависимости от того, какую конструкцию хеш-таблицы мы хотим использовать, можно использовать 0.5 млн корзин, или 2 млн корзин, или что-то еще; так или иначе, нам нужен постоянный коэффициент, близкий к n. Но оба этих значения значительно меньше, чем U. Вот почему хеш-функция, которая соотносит элементы из U с m корзинами, неизбежно будет приводить к довольно большому подмножеству универсального множества U, соотносящему с одной и той же корзиной хеш-таблицы. Даже если хеш-функция распределяет элементы из универсального множества идеально равномерно, существует по меньшей мере одна корзина, в которую попадает по меньшей мере $|U| / m$ элементов. Мы не знаем, какие элементы будут содержаться в нашем наборе данных, и если $|U| / m \geqslant n$, то вполне возможно, что все элементы набора данных будут хешироваться в одну и ту же корзину. Маловероятно, что мы получим такой набор данных, но это возможно.

Например, рассмотрим универсальное множество всех потенциальных телефонных номеров формата *ddd-dd-dddd-ddddd*, где *d* – цифра от нуля до девяти. Поскольку каждая из 12 цифр может принимать 10 разных значений, это означает, что $|U| = 10^{12}$, и если $n = 10^5$ (размер набора данных) и $m = 10^6$ (размер таблицы), то даже если хеш-функция идеально распределит элементы универсального множества, мы все равно можем получить все элементы в одной корзине. Рассмотрим случай идеально равномерного распределения универсального множества по корзинам; тогда каждой корзине назначается $10^{12}/10^6 = 10^6$ элементов. Поскольку размер нашего набора данных меньше 10^6, можно найти такой набор данных, в котором все элементы попадают в одну и ту же корзину. Не лучше было бы, если в одну и

ту же корзину попадала какая-то постоянная доля нашего набора данных (то есть половина или треть)?

Возможность такой ситуации не должна нас обескураживать. В большинстве практических приложений даже простые хеш-функции достаточно хороши, чтобы это происходило очень редко, но коллизии будут происходить в общих случаях, и нам нужно уметь с ними бороться.

2.5 Урегулирование коллизий: теория и практика

Мы посвятим этот раздел двум распространенным механизмам урегулирования коллизий: линейному опробыванию и прохождению по цепочкам. Есть много других, но мы рассмотрим эти два, поскольку они являются наиболее популярными вариантами хеш-таблиц, работающих внутри вашего исходного кода. Как вы, вероятно, знаете, в механизме прохождения по цепочкам[13] с каждой корзиной хеш-таблицы ассоциируется дополнительная структура данных (например, связный список или дерево двоичного поиска), в которой хранятся элементы, хешированные в соответствующей корзине. Новые элементы вставляются спереди (O(1)), но для поиска и удаления требуется перемещение по указателям соответствующего списка – операция, время выполнения которой сильно зависит от равномерности распределения элементов по корзинам. Если вы хотите освежить свои знания о механизме прохождения по цепочкам, то взгляните на рис. 2.8.

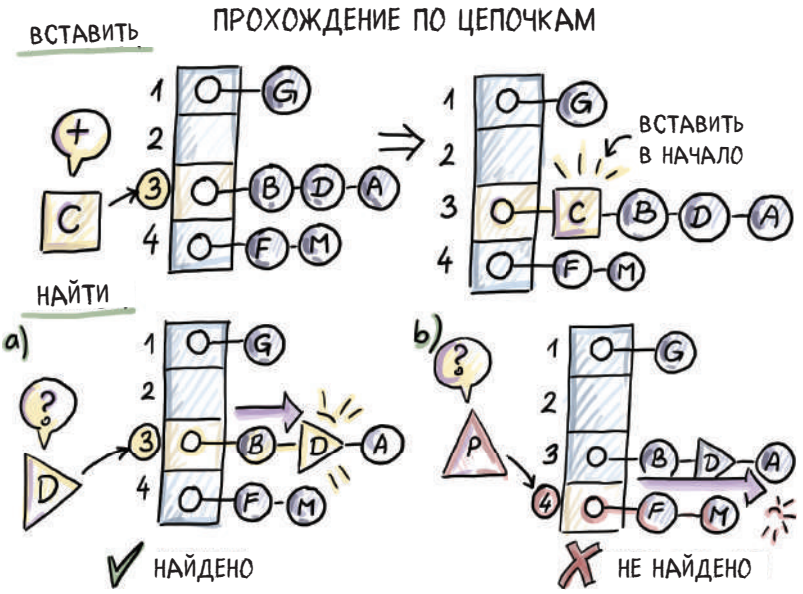


Рисунок 2.8 Пример вставки и поиска с прохождением по цепочкам

[13] Англ. chaining. – *Прим. перев.*

Линейное опробывание[14] – это частный случай открытой адресации, схемы хеширования, при котором элементы хранятся внутри слотов фактической хеш-таблицы. В рамках линейного опробывания, перед тем как вставить элемент, он хешируется в соответствующую корзину, и если обусловленный корзиной слот пуст, то элемент сохраняется в нем. Если же он занят, то мы ищем первую свободную позицию, сканируя таблицу вниз, и, при необходимости, продолжаем с начала таблицы. В альтернативном варианте открытой адресации, квадратичном опробывании, поиск следующей позиции для вставки выполняется шагами квадратичного размера.

Поиск в линейном опробывании, так же как и при вставке, начинается с позиции обусловленного корзиной слота, в который элемент был хеширован, и выполняется вниз до тех пор, пока не будет найден искомый элемент либо не встречен пустой слот. С удалением дело обстоит несколько сложнее, поскольку здесь нельзя просто удалить элемент из его слота – это может привести к разрыву цепочки, что приведет к неверному результату будущего поиска. Решить эту проблему можно разными способами, простым из которых является установка надгробного флага[15] на место удаляемого элемента. Пример линейного опробывания приведен на рис. 2.9.

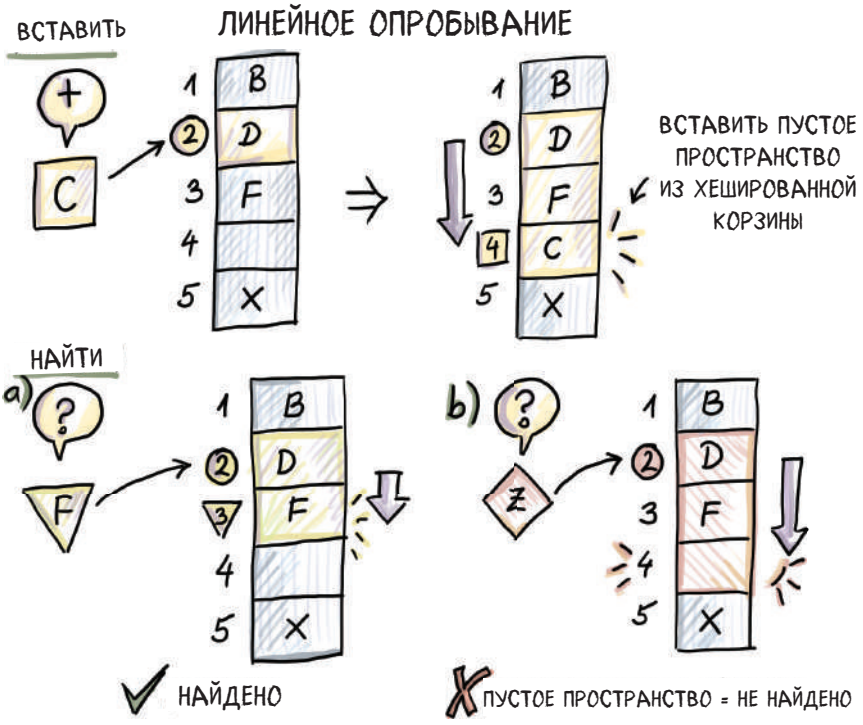


Рисунок 2.9 Пример вставки и поиска в линейном опробывании

[14] Англ. linear probing; син. линейное зондирование/прощупывание. – *Прим. перев.*

[15] Англ. tombstone flag. – *Прим. перев.*

Сначала посмотрим, что говорит нам теория о преимуществах и недостатках этих двух методов урегулирования коллизий. С теоретической точки зрения при изучении хеш-функций и методов урегулирования коллизий специалисты по информатике часто принимают допущение о том, что хеш-функции являются идеально случайными, что позволяет анализировать процесс хеширования, используя вероятность и аналогию с равномерным и случайным бросанием n шаров в n корзин.

В самой полной корзине с высокой вероятностью будет $O(\log n/\log\log n)$ шариков (**http://mng.bz/QWjm**); следовательно, самая длинная цепочка в методе прохождения по цепочкам не длиннее $O(\log n/\log\log n)$, давая верхнюю границу производительности поиска и удаления.

Высоковероятностные границы сильнее, чем ожидаемые границы, которые мы обсуждали ранее. Выражение «с высокой вероятностью» означает, что если входной элемент имеет размер n, тогда высоковероятностное событие произойдет с вероятностью не менее $1 - 1/n^c$, где $c \leqslant 1$ – это некая константа. В нашем случае высоковероятностным событием будет цепочка, или непрерывный отрезок, в хеш-таблице с верхним логарифмическим ограничением на ее размер. Другими словами, мы устанавливаем верхнюю границу вероятности того, что цепочка/отрезок будет иметь длину, превышающую логарифмическую. Таким образом, чем выше константа и размер входного элемента, тем меньше возможность того, что высоковероятностное событие не произойдет, но при $c = 1$ все уже хорошо. Практически это означает, что прежде чем высоковероятностное событие даст сбой, произойдет много других сбоев.

Логарифмическое время поиска – это неплохо, но если бы все случаи поиска были такими, то хеш-таблица не имела бы существенных преимуществ, скажем, перед деревом двоичного поиска. Однако в большинстве случаев мы ожидаем, что время поиска будет константой (исходя из допущения, что число элементов является пропорциональным числу корзин в таблице цепочек).

Используя попарно независимое хеширование, можно показать, что наихудший случай поиска при линейном опробывании близок к $O(\log n)$ [4]. Семейство k-попарно независимых хеш-функций наиболее близко к тому, к чему мы подошли на данный момент, имитируя случайное поведение. Во время выполнения одна из хеш-функций семейства выбирается равномерно случайно для использования во всей программе. Это защищает нас от злоумышленников, которые могут увидеть наш исходный код: выбирая одну из множества хеш-функций случайно во время выполнения, мы усложняем генерирование патологического набора данных, и даже если это произойдет, то не по нашей вине. Подобные решения могут повлиять и на безопасность нашего приложения.

Интуитивно понятно, что стоимость поиска в худшем случае при линейном опробывании слегка выше, чем при прохождении по цепочкам, поскольку хеширование элементов в разные корзины может способствовать увеличению длины одного и того же отрезка линейного опробывания. Но

отражает ли причудливая теория реальные различия в производительности?

На самом деле мы упускаем важную деталь. Отрезки линейного опробывания размещаются в памяти последовательно, и большинство отрезков короче одной строки кеша, которая должна доставляться в любом случае, независимо от продолжительности отрезка. То же самое нельзя сказать об элементах списка в рамках механизма прохождения по цепочкам, память для которых отводится непоследовательным образом. Следовательно, прохождение по цепочкам может нуждаться в большем доступе к памяти, что существенно влияет на фактическое время выполнения. Аналогичный случай имеет место с другим умным методом урегулирования коллизий, именуемым кукушечным хешированием[16], который обещает, что содержащийся в таблице элемент будет найден в одной из двух позиций, определяемых двумя хеш-функциями, расценивая стоимость поиска постоянной в наихудшем случае. Однако пробы нередко берутся в очень разных областях таблицы, поэтому могут понадобиться две точки доступа к памяти.

Учитывая разрыв между временем, требуемым для доступа к памяти, и центральным процессором, о котором мы говорили в главе 1, вполне резонно, что во многих практических реализациях линейное опробывание нередко используется в качестве предпочтительного метода урегулирования коллизий. Далее мы рассмотрим пример современного языка программирования, в котором словарь ключ-значение реализован с помощью хеш-таблиц.

2.6 Сценарий использования: принцип работы словаря в языке Python

Словари ключ-значение получили широкое распространение в разных языках программирования. Например, в стандартных библиотеках C++ и Java они реализованы как `map`, `unordered_map` (C++) и `HashMap` (Java); `map` – это красно-черное дерево, в котором элементы упорядочены, а `unordered_map` и `HashMap` не упорядочены, и внутри них используются хеш-таблицы. В обоих для урегулирования коллизий используется механизм прохождения по цепочкам. В Python словарем ключ-значение является `dict`. Ниже приведен простой пример, показывающий, как создавать, изменять и обращаться к ключам и значениям в `dict`:

```
d = {'turmeric':  , 'cardamom': 5, 'oregano': 12}
print(d.keys())
print(d.values())
print(d.items())
d.update({'saffron': 11})
print(d.items())
```

[16] Англ. cuckoo hashing. – *Прим. перев.*

Результат выглядит следующим образом:

```
dict_keys(['turmeric', 'cardamom', 'oregano'])
dict_values([7, 5, 12])
dict_items([('turmeric', 7), ('cardamom', 5), ('oregano', 12)])
dict_items([('turmeric', 7), ('cardamom', 5), ('oregano', 12), ('saffron', 11)])
```

Авторы дефолтной реализации Python, CPython, в своей документации [5] объясняют реализацию словаря `dict` так (здесь мы сосредоточимся только на случае, когда ключи являются целыми числами): для размера таблицы $m = 2^i$ хеш-функция равна $h(x) = x \bmod 2^i$ (то есть номер корзины определяется последними i битами двоичного представления элемента x). Она хорошо работает в ряде распространенных случаев, таких как последовательность поочередных чисел, где она не создает коллизий; также легко найти случаи, когда она работает крайне плохо, например набор всех чисел с одинаковыми последними i битами. Более того, при использовании в сочетании с линейным опробыванием эта хеш-функция может приводить к кластеризации и длинным отрезкам поочередных элементов. Во избежание длинных отрезков в Python используется следующий механизм опробывания:

```
j = ((5*j) + 1) mod 2**i
```

где j – это индекс корзины, куда мы попытаемся вставить в следующий раз. Если слот занят, то мы повторим процесс, используя новый j. Эта последовательность обеспечивает посещение всех m корзин хеш-таблицы с течением времени и делает достаточно пропусков, чтобы избегать кластеризации в общем случае. Использование старших битов ключа при хешировании обеспечивает переменная `perturb`, которая изначально инициализируется значением $h(x)$, и константа `PERTURB_SHIFT`, установленная равной 5:

```
perturb >>= PERTURB_SHIFT
j = (5*j) + 1 + perturb      ❶
```

❶ j % 2^i – это следующая корзина, которую мы попытаемся попробовать

Если вставки совпадают с нашим шаблоном $(5 * j) + 1$, то нас ждут неприятности, но в Python и большинстве практических реализаций хеш-таблиц, по всей видимости, основное внимание уделяется очень важному практическому принципу конструирования алгоритмов: делать общий случай простым и быстрым и не беспокоиться об эпизодической заминке, когда происходит редкий тяжелый случай.

2.7 Хеш-функция MurmurHash

В данной книге нас интересуют быстрые, надежные и простые хеш-функции. С этой целью мы кратко упомянем хеш-функцию MurmurHash,

которая была изобретена Остином Эпплби (Austin Appleby) и представляет собой быструю некриптографическую хеш-функцию, используемую во многих реализациях структур данных, которые мы представим в будущих главах нашей книги. Название *Murmur* происходит от базовых операций умножения и поворота, которые используются для измельчения ключей. Одной из Python'овских оберток хеш-функции MurmurHash является `mmh3` (`https://pypi.org/project/mmh3/`), которую можно установить в консоли с помощью команды

```
pip install mmh3
```

Пакет `mmh3` предоставляет несколько способов выполнения хеширования. Базовая хеш-функция позволяет генерировать 32-битовые целые числа со знаком и без знака с разными начальными позициями генератора псевдослучайных чисел:

```
import mmh3
print(mmh3.hash("Привет"))
print(mmh3.hash(key = "Привет", seed = 5, signed = True))
print(mmh3.hash(key = "Привет", seed = 20, signed = True))
print(mmh3.hash(key = "Привет", seed = 20, signed = False))
```

В результате чего генерируется разный хеш для разных значений параметров `seed` и `signed`:

```
316307400
-196410714
-1705059936
2589907360
```

Для генерирования 64-битовых и 128-битовых хешей используются функции `hash64` и `hash128`, где `hash64` применяет 128-битовую хеш-функцию и генерирует пару 64-битовых хешей со знаком либо без знака. Как 64-битовые, так и 128-битовые хеш-функции позволяют указывать архитектуру (x64 или x86), чтобы оптимизировать функцию под заданную архитектуру

```
print(mmh3.hash64("Привет"))
print(mmh3.hash64(key = "Привет", seed = 0, x64arch= True, signed = True))
print(mmh3.hash64(key = "Привет", seed = 0, x64arch= False, signed = True))
print(mmh3.hash128("Привет"))
```

В результате чего генерируются следующие ниже (пары) хешей:

```
(3871253994707141660, -6917270852172884668)
(3871253994707141660, -6917270852172884668)
(6801340086884544070, -5961160668294564876)
212681241822374483335035321234914329628
```

2.8 Хеш-таблицы для распределенных систем: согласованное хеширование

Впервые согласованное хеширование было замечено в контексте веб-кеширования [6]. Кеши имеют основополагающее значение в информатике и усовершенствовали системы во многих областях. Например, в интернете кеши устраняют горячие точки, возникающие, когда много клиентов запрашивают одну и ту же веб-страницу с сервера. Серверы служат вместилищем веб-страниц, клиенты запрашивают их через браузеры, а кеши располагаются между ними и содержат копии часто посещаемых веб-страниц. В большинстве ситуаций кеши способны удовлетворять запрос быстрее, чем домашние серверы, и распределять нагрузку между собой таким образом, чтобы кеш не был перегружен. При безуспешном обращении к кешу[17] (то есть веб-страница в кеше не найдена) кеш доставляет веб-страницу с исходного сервера. При такой конфигурации необходимо решать важную проблему, связанную с распределением веб-страниц (далее: ресурсов) между кешами (далее: узлами) с учетом следующих ниже ограничений:

- соотнесение ресурса с узлом должно быть быстрым и простым. Клиент и сервер должны быть в состоянии быстро вычислять узел, ответственный за данный ресурс;
- нагрузка на ресурсы между разными узлами должна быть примерно одинаковой, чтобы избегать возникновения горячих точек;
- соотнесение должно быть гибким в условиях частых прибытий и отбытий узлов. Как только узел отбывает (то есть происходит спонтанный отказ), его ресурсы должны эффективно переназначаться другим узлам, а при добавлении нового узла он должен получать равную порцию общей сетевой нагрузки. Все это должно происходить бесшовно, без воздействия на слишком большое число других узлов/ресурсов.

2.8.1 Типичная проблема хеширования

Исходя из первых двух требований следует, что у нас образовалась проблема с хешированием: узлы – это корзины, в которые хешируются ресурсы, и хорошая хеш-функция может обеспечивать справедливый баланс нагрузки. Наличие хеш-таблицы позволяет выяснять узел, на котором находится тот или иной ресурс. Так, при появлении поискового запроса мы хешируем ресурс и смотрим, в какой корзине (узле) он должен содержаться (рис. 2.10, слева). Это было бы прекрасно, если бы мы не находились в высокодинамичной распределенной среде, в которой узлы постоянно присоединяются и покидают (отказывают) (рис. 2.10, справа). Трудность заключается в удовлетворении требования: как переназначать ресурсы узлов, когда они покидают сеть, или как назначать ресурсы вновь прибывающему узлу, па-

[17] Англ. cache miss; син. непопадание в кеш. – *Прим. перев.*

мятуя о том, что баланс нагрузки должен оставаться примерно равным и не слишком нарушать работу сети.

Рисунок 2.10 Используя хеш-таблицу, можно ставить в соответствие ресурсы узлам и помогать находить узел, соответствующий запрашиваемому ресурсу (слева). Проблема возникает, когда узлы присоединяются к сети / покидают ее (справа)

Как мы знаем, классические хеш-таблицы можно переразмеривать путем перехеширования с использованием новой хеш-функции с другим диапазоном и копирования элементов в новую таблицу. Эта операция обходится очень дорого и, как правило, окупается, потому что выполняется только время от времени и амортизируется за счет большого числа недорогих операций. В нашем сценарии динамического веб-кеширования, в котором прибытие и отбытие узлов происходят постоянно, крайне непрактично изменять соотнесения ресурсов с узлами при каждом незначительном изменении в сети (рис. 2.11).

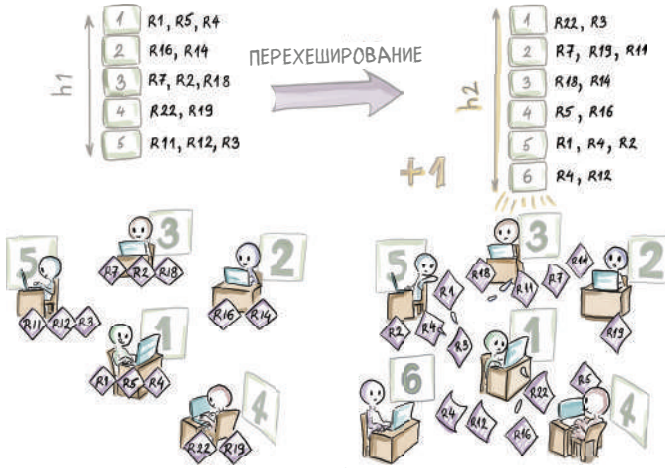


Рисунок 2.11 Перехеширование, нереализуемое в высокодинамичном контексте, поскольку присоединение/отказ одного узла инициирует повторное отведение ресурсов узлам. В этом примере изменение размера хеш-таблицы с 5 на 6 изменило отведение большинства ресурсов узлам. На рисунке справа внизу показан «промежуточный» момент, когда узлы содержат несколько устаревших и несколько новых ресурсов

В следующих далее разделах мы покажем, как последовательное хеширование помогает удовлетворять все три требования нашей задачи. Мы начнем с представления понятия хеш-кольца.

2.8.2 Хеш-кольцо

Главная идея согласованного хеширования заключается в хешировании как ресурсов, так и узлов в фиксированный диапазон $R = [0, 2^k - 1]$. Диапазон R полезно представлять визуально, распределенным по кругу, причем самая северная точка равна 0, а остальная часть диапазона равномерно распределена по кругу по часовой стрелке в порядке возрастания. Такой круг называется *хеш-кольцом*.

Каждый ресурс и узел имеют позицию в хеш-кольце, задаваемую их хешами. Имея такую конфигурацию, каждый ресурс назначается первому узлу, встречаемому по часовой стрелке в хеш-кольце. Хорошая хеш-функция должна обеспечивать, чтобы каждый узел получал достаточно эквивалентную загруженность ресурсами. Взгляните на пример, приведенный на рис. 2.12.

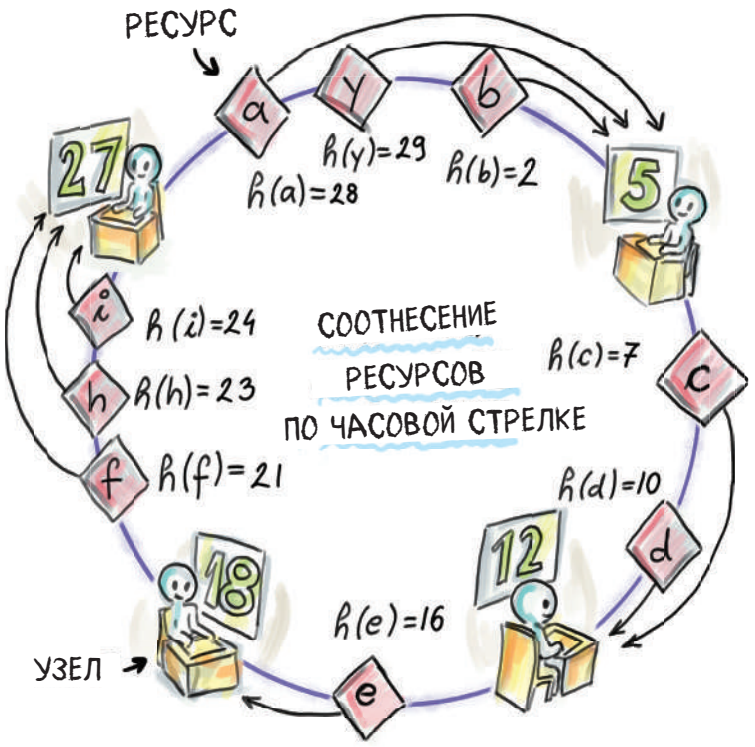


Рисунок 2.12 Соотнесение ресурсов узлам в хеш-кольце. В примере показано хеш-кольцо $R = [0, 31]$ и узлы, хеши которых равны 5, 12, 18 и 27. Ресурсы *a*, *y* и *b* назначены узлу 5, ресурсы *c* и *d* назначены узлу 12, ресурс *e* назначен узлу 18, а ресурсы *f*, *h* и *i* назначены узлу 27

Для иллюстрации принципа работы согласованного хеширования, а также прибытия и убытия узлов мы пошагово продемонстрируем простую реализацию класса `HashRing` на языке Python. Наша реализация, показанная в серии небольших фрагментов, является лишь имитацией алгоритма (фактическая реализация согласованного хеширования предусматривает сетевые вызовы между узлами и т. п.). Класс `HashRing` реализован с использованием циклического двусвязного списка узлов, в котором каждый узел хранит свои ресурсы в локальном словаре:

```
class Node:
    def __init__(self, hashValue):
        self.hashValue = hashValue
        self.resources = {}
        self.next = None
        self.previous = None

class HashRing:
    def __init__(self, k):
        self.head = None
        self.k = k
        self.min = 0
        self.max = 2**k - 1
```

В конструкторе класса `HashRing` используется параметр `k`, который инициализирует диапазон равным $[0, 2^k - 1]$. Класс `Node` имеет атрибут `hashValue`, который обозначает его позицию в кольце, и словарь `resources`, который содержит его ресурсы. Остальная часть исходного кода очень напоминает типичную реализацию циклического двусвязного списка.

Первый базовый метод описывает легальный диапазон хеш-значений ресурсов и узлов, которые мы разрешаем в хеш-кольце:

```
def legalRange(self, hashValue):
    return self.min <= hashValue <= self.max
```

Для назначения ресурсов ближайшим к ним узлам мы определяем понятие ближайшего узла в хеш-кольце, используя следующий ниже метод `distance`:

```
def distance(self, a, b):
    if a == b:
        return 0
    elif a < b:
        return b - a
    else:
        return (2**self.k) + (b - a)
```

Например, если инициализировать пустое хеш-кольцо параметром `k=5`:

```
hr = HashRing(5)
print(hr.distance(29,5))
print(hr.distance(29,12))
print(hr.distance(5,29))
```

то мы получим следующий ниже результат:

```
8
15
24
```

Расстояние в кольце от ресурса 29 до узла 5 равно 8, что короче расстояния от 29 до 12 (и фактически короче, чем до любого другого узла из нашего примера на рис. 2.6, в результате чего узлу 5 назначается ресурс 29). Следует учитывать, что порядок аргументов в этой функции имеет значение.

2.8.3 Поиск

Первая функциональность, которую необходимо реализовать в отношении класса `HashRing`, – это поиск соответствующего узла по хеш-значению ресурса. Мы двигаемся по хеш-кольцу, начиная с первого узла (с наименьшим хеш-значением) и следуя по прямым ссылкам до тех пор, пока текущий и следующий узлы не окажутся по одну сторону от ресурса. Условие цикла нарушается, когда мы собираемся проскочить через ресурс; то есть текущий узел предшествует ресурсу, а следующий узел идет сразу после ресурса, и именно этот узел нам нужно вернуть. Если ресурс присутствует, то это тот узел, который содержит ресурс. Указанная функциональность содержится в методе `lookupNode`:

```
def lookupNode(self, hashValue):
    if self.legalRange(hashValue):
        temp = self.head
        if temp is None:
            return None
        else:
            while(self.distance(temp.hashValue, hashValue) >
               ➥ self.distance(temp.next.hashValue, hashValue)):
                temp = temp.next
            if temp.hashValue == hashValue:
                return temp
            return temp.next
```

В этой реализации мы исходим из отсутствия коллизий хешей: никакие два разных узла (и никакие два разных ресурса) не будут иметь одинаковое хеш-значение. Однако может случиться так, что ресурс и узел окажутся в одной и той же позиции в хеш-кольце, и тогда ресурс с хеш-значением *i* будет назначен узлу *i*.

2.8.4 Добавление нового узла/ресурса

Когда в хеш-кольцо добавляется новый узел A, некоторые ресурсы, ранее принадлежавшие тому узлу, который теперь является преемником узла A, возможно, потребуется переназначить узлу A. Теперь расстояние этих ресурсов до A меньше, чем до ранее назначенного им узла (то есть A находится на пути по часовой стрелке к узлу, назначенному им в настоящий момент). Пример вставки узла с хеш-значением 30 приведен на рис. 2.13.

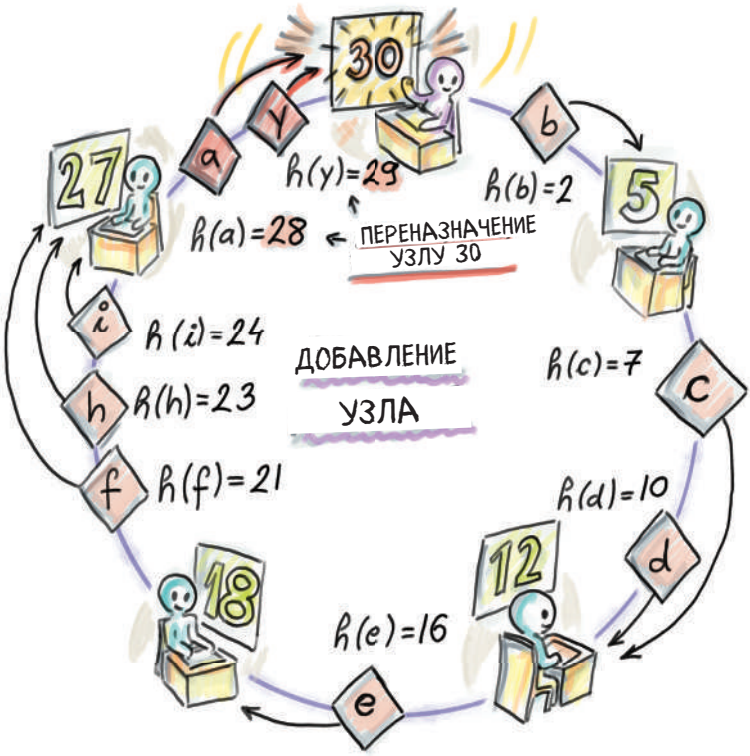


Рисунок 2.13 Прибытие нового узла. Ресурсы *a* и *y* с соответствующими хеш-значениями 28 и 29 теперь переназначаются вновь вставленному узлу с хеш-значением 30

Обратите внимание, что такой способ добавления узла конгруэнтен последнему ограничению из начала раздела: при добавлении нового узла потенциально изменяют соотнесения ресурсы только одного другого узла, а все остальные соотнесения остаются нетронутыми.

Сначала давайте посмотрим, как функциональность перемещения ресурсов реализована во вспомогательном методе `moveResources`, который также будет использоваться позже для удалений узлов:

```
def moveResources(self, dest, orig, deleteTrue):                    ❶
    delete_list = []
```

```
    for i, j in orig.resources.items():
        if (self.distance(i, dest.hashValue) < self.distance(i,
                                                             orig.hashValue)
            ➥ or deleteTrue):
            dest.resources[i] = j
            delete_list.append(i)
            print("\tПеремещение ресурса " + str(i) + " из " +
                  ➥ str(orig.hashValue) + " в " + str(dest.hashValue))
    for i in delete_list:                                              ❷
        del orig.resources[i]
```

❶ Переместить некоторые ресурсы из `orig` в `dest`
❷ Удалить назначенные ресурсы из `orig`

Особые случаи добавления узлов возникают, когда вновь добавленный узел становится головным узлом либо когда существующий список пуст. В обычном случае используется описанная ранее функция поиска, чтобы локализовывать правильное место для нового узла, а затем выполнять необходимую перекоммутацию хеш-кольца:

```
def addNode(self, hashValue):
    if self.legalRange(hashValue):
        newNode = Node(hashValue)
        if self.head is None:                                  ❶
            newNode.next = newNode
            newNode.previous = newNode
            self.head = newNode
            print("Добавление головного узла " + str(newNode.hashValue) + "...")
        else:
            temp = self.lookupNode(hashValue)                  ❷
            newNode.next = temp
            newNode.previous = temp.previous
            newNode.previous.next = newNode
            newNode.next.previous = newNode
            print("Добавление узла " + str(newNode.hashValue) +
                  ➥ ". Предыдущий: " + str(newNode.previous.hashValue) +
                  ➥ " и следующий: " + str(newNode.next.hashValue) + ".")
            self.moveResources(newNode, newNode.next, False)
            if hashValue < self.head.hashValue:                ❸
                self.head = newNode
```

❶ Пустое хеш-кольцо
❷ Преемник
❸ Изменяет указатель на голову

Теперь, когда мы знаем, как добавлять узлы, можно добавить и несколько ресурсов. Для добавления нового ресурса мы, естественно, используем метод `lookupNode` и обновляем словарь `resources` соответствующего узла

новым ресурсом. Для того чтобы добавить новый ресурс, в хеш-кольце должен быть хотя бы один узел:

```
def addResource(self, hashValueResource):
    if self.legalRange(hashValueResource):
        print("Добавление ресурса " + str(hashValueResource) + "...")
        targetNode = self.lookupNode(hashValueResource)
        if targetNode is not None:
            value = "фиктивное значение ресурса " + str(hashValueResource)
            targetNode.resources[hashValueResource] = value
        else:
            print("Не удается добавить ресурс в пустое хеш-кольцо")
```

2.8.5 Удаление узла

Удаление узла в хеш-кольце работает следующим образом: когда узел B покидает хеш-кольцо, что часто соответствует спонтанному отказу узла, тогда ресурсы, ранее принадлежавшие B, должны быть назначены тому, кто был преемником узла B в хеш-кольце (см. рис. 2.14). Опять же, это изменение затрагивает лишь малую часть ресурсов.

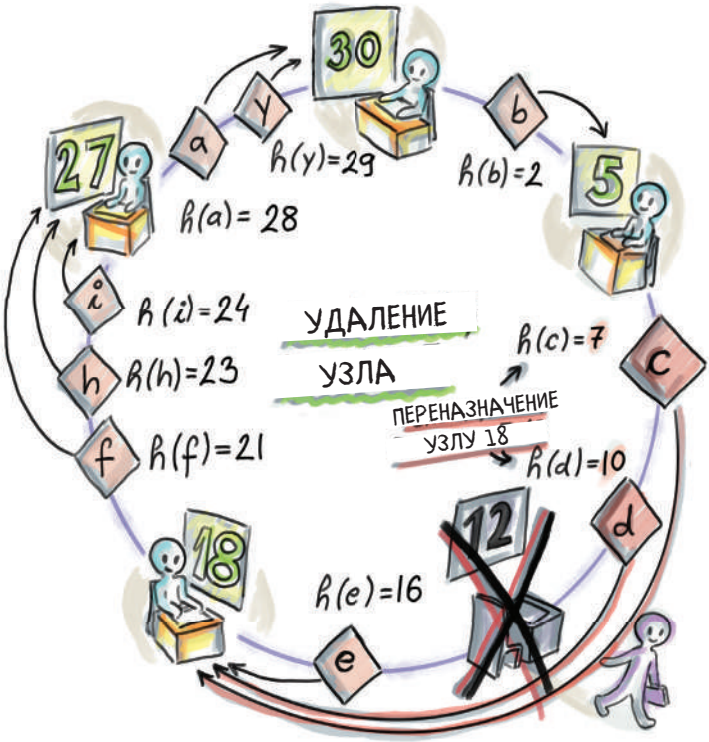


Рисунок 2.14 Удаление узла. В этом примере узел с хеш-значением 12 покидает сеть, и его ресурсы *c* и *d* с хеш-значениями 7 и 10 соответственно переназначаются узлу с хеш-значением 18, предыдущему преемнику узла 12

Реализация должна учитывать случаи пустых и одноэлементных хеш-колец и попытку удалять несуществующий узел либо удалять головной элемент, в котором указатель на голову должен быть исправлен:

```
def removeNode(self, hashValue):
    temp = self.lookupNode(hashValue)
    if temp.hashValue == hashValue:
        print("Удаление узла " + str(hashValue) + ": ")
        self.moveResources(temp.next, temp, True)
        temp.previous.next = temp.next
        temp.next.previous = temp.previous
        if self.head.hashValue == hashValue:    ❶
            self.head = temp.next
            if self.head == self.head.next:     ❷
                self.head = None
        return temp.next
    else:
        print("Удалять нечего.")                ❸
```

❶ Удаляет головной элемент
❷ Ситуация удаления из одноэлементного хеш-кольца
❸ Такого узла нет

Наконец, чтобы иметь возможность отображать содержимое хеш-кольца, мы реализуем простой метод печати, который показывает текущее состояние хеш-кольца, при этом узлы распечатываются в порядке возрастания (по часовой стрелке), начиная с самой северной точки кольца, вместе с локальными ресурсами каждого узла, хранящимися в локальной хеш-таблице:

```
def printHashRing(self):
    print("*****")
    print("Распечатка хеш-кольца по часовой стрелке:")
    temp = self.head
    if self.head is None:
        print("Пустое хеш-кольцо")
    else:
        while(True):
            print("Узел: " + str(temp.hashValue) + ", ", end=" ")
            print("Ресурсы: ", end=" ")
            if not bool(temp.resources):
                print("Пусто", end="")
            else:
                for i in temp.resources.keys():
                    print(str(i), end=" ")
            temp = temp.next
            print(" ")
```

```
            if (temp == self.head):
                break
    print("*****")
```

Имея за плечами всю эту функциональность, теперь мы готовы показать пример.

Пример

Давайте начнем с выполнения процесса, показанного на рис. 2.12 и 2.13. Сначала мы добавляем несколько узлов и ресурсов в случайном порядке и наблюдаем, как происходит переназначение ресурсов по мере добавления узлов 5, 27 и 30. Обратите внимание, что любой порядок добавления узлов и ресурсов (при условии что первый добавленный объект является узлом, а не ресурсом) должен приводить к одинаковому хеш-кольцу:

```
hr = HashRing(5)
hr.addNode(12)
hr.addNode(18)
hr.addResource(24)
hr.addResource(21)
hr.addResource(16)
hr.addResource(23)
hr.addResource(2)
hr.addResource(29)
hr.addResource(28)
hr.addResource(7)
hr.addResource(10)
hr.printHashRing()
```

В результате чего мы получаем следующий ниже результат:

```
Добавление головного узла 12...
Добавление узла 18. Предыдущий: 12 и следующий: 12.
Добавление ресурса 24...
Добавление ресурса 21...
Добавление ресурса 16...
Добавление ресурса 23...
Добавление ресурса 2...
Добавление ресурса 29...
Добавление ресурса 28...
Добавление ресурса 7...
Добавление ресурса 10...
*****
Распечатка хеш-кольца по часовой стрелке:
Узел: 12, Ресурсы:  24 21 23 2 29 28 7 10
Узел: 18, Ресурсы:  16
*****
```

Теперь мы добавим два оставшихся узла из рис. 2.12 и посмотрим, как происходит переназначение ресурсов:

```
hr.addNode(5)
hr.addNode(27)
hr.addNode(30)
hr.printHashRing()
```

Результат выглядит следующим образом:

```
Добавление узла 5. Предыдущий: 18 и следующий: 12.
    Перемещение ресурса 24 из 12 в 5
    Перемещение ресурса 21 из 12 в 5
    Перемещение ресурса 23 из 12 в 5
    Перемещение ресурса 2 из 12 в 5
    Перемещение ресурса 29 из 12 в 5
    Перемещение ресурса 28 из 12 в 5
Добавление узла 27. Предыдущий: 18 и следующий: 5.
    Перемещение ресурса 24 из 5 в 27
    Перемещение ресурса 21 из 5 в 27
    Перемещение ресурса 23 из 5 в 27
Добавление узла 30. Предыдущий: 27 и следующий: 5.
    Перемещение ресурса 29 из 5 в 30
    Перемещение ресурса 28 из 5 в 30
*****
Распечатка хеш-кольца по часовой стрелке:
Узел: 5, Ресурсы:  2
Узел: 12, Ресурсы:  7 10
Узел: 18, Ресурсы:  16
Узел: 27, Ресурсы:  24 21 23
Узел: 30, Ресурсы:  29 28
*****
```

Этот результат отражает состояние хеш-кольца на рис. 2.13. Теперь давайте удалим узел:

```
hr.removeNode(12)
hr.printHashRing()
```

Окончательное хеш-кольцо, как показано на рис. 2.14, выглядит следующим образом:

```
Удаление узла 12:
    Перемещение ресурса 7 из 12 в 18
    Перемещение ресурса 10 из 12 в 18
*****
Распечатка хеш-кольца по часовой стрелке:
Узел: 5, Ресурсы:  2
Узел: 18, Ресурсы:  16 7 10
```

```
Узел: 27, Ресурсы:  24 21 23
Узел: 30, Ресурсы:  29 28
*****
```

2.8.6 Сценарий согласованного хеширования: хордовый протокол

Протокол Chord, далее хордовый протокол, [7] – это протокол распределенного поиска для одноранговых сетей, в котором применяется согласованное хеширование. Помимо того что схема из упомянутой статьи используется в ряде одноранговых сетей, она также была перепрофилирована под высокомасштабируемое хранилище данных Amazon Dynamo, в котором хранятся различные стержневые службы платформы электронной коммерции Amazon [8].

Реализованный нами незамысловатый протокол на основе связного списка оставляет желать лучшего с точки зрения эффективности работы в реальной производственной системе. Когда мы хотим маршрутизировать запрос от ресурса, то ожидаем, что он будет проходить по линейному числу прямых указателей, и каждый такой указатель транслируется в сетевой вызов между двумя машинами. Время, необходимое для маршрутизации вызова, не будет масштабироваться в больших системах. Кроме этого, для того чтобы перенаправлять запрос, каждая машина должна поддерживать копию хеш-кольца, вследствие чего будет потреблять нетривиальный объем локальной памяти.

Хордовый протокол улучшает базовый алгоритм за счет того, что каждый узел хранит информацию только о других $O(\log n)$ узлах. Каждый узел `x` поддерживает так называемую пальцевую таблицу, которая хранит соотнесения ключ-значение для точек в хеш-кольце на экспоненциально увеличивающихся расстояниях от `x` (такие ключи называются ключами-пальцами[18]) до их узлов-преемников. Это помогает алгоритму поиска отыскивать нужный узел за логарифмическое число шагов.

В частности, для хеш-кольца с интервалом $R = [0, 2^k - 1]$ пальцевая таблица узла `x` содержит все пальцы `f_i` таким образом, что `distance(x, f_i) =` 2^{i-1} для всех $i \leqslant k$. Преемники пальцев могут быть вычислены с помощью метода `lookupNode`, который мы реализовали ранее. В качестве примера взгляните на рис. 2.15 и пальцевую таблицу для узла `x=5`.

Как пальцевые таблицы используются для ускорения поиска? Операция поиска в этой схеме работает таким образом, что если пальцевая таблица узла, откуда исходит запрос, не содержит ресурса `y`, то узел перенаправляет запрос преемнику, на который указывает палец с наименьшим расстоянием до ресурса. Пример показан на рис. 2.16, где поиск ресурса с хеш-значением 29 начинается с узла 5.

[18] Англ. key finger; син. ключ-стрелка, ключ-указатель. – *Прим. перев.*

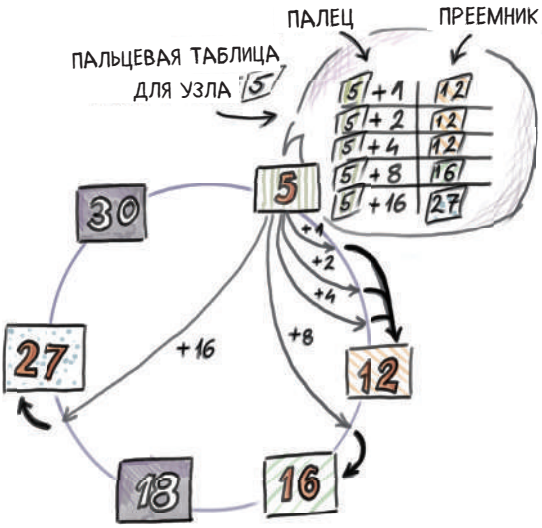


Рисунок 2.15 Пример пальцевой таблицы для узла 5 в хеш-кольце, где R = [0, 31]. В пальцевой таблице узла 5 хранится пять записей, для преемников точек 5 + 1 = 6, 5 + 2 = 7, 5 + 4 = 9, 5 + 8 = 13 и 5 + 16 = 21. Соответствующие преемники таковы: 12, 12, 12, 16 и 27

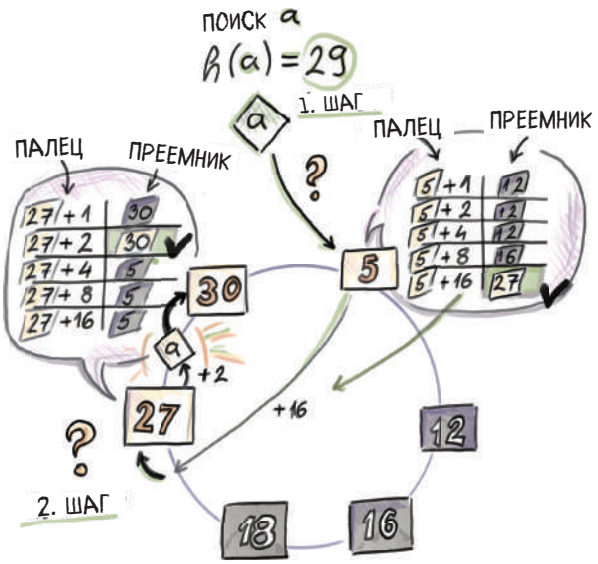


Рисунок 2.16 Процедура поиска с помощью пальцевых таблиц. Для того чтобы найти ресурс 29, начиная с узла 5, мы сначала следуем в направлении, указанном пальцем (21 = 5 + 16), так как этот палец имеет наименьшее расстояние до 29. Его преемником является 27, поэтому запрос перенаправляется к 27. В пальцевой таблице узла 27 мы берем палец 2, который дает ровно 29. Его преемником является узел 30, в котором запрос окончательно маршрутизируется (то есть если ресурс существует, то он будет найден в узле 30)

Ниже приведено несколько упражнений по программированию, чтобы проверить ваше понимание хордового протокола и пальцевых таблиц.

2.8.7 Согласованное хеширование: упражнения по программированию

Упражнение 1

С учетом исходного кода класса `HashRing` добавьте в определение класса `Node` новый атрибут `fingerTable` типа `dict`. Теперь реализуйте метод `buildFingerTables(self)` в классе `HashRing`, который создает пальцевую таблицу для каждого узла в хеш-кольце с использованием методов, которые мы уже реализовали. Наряду с парой, содержащей палец и преемника, ваша пальцевая таблица также должна хранить прямой указатель на данный узел (чтобы разрешать прямой доступ к узлу из пальцевой таблицы).

Упражнение 2

Теперь, когда каждый узел содержит свою собственную пальцевую таблицу, реализуйте более эффективный поиск в методе `chordLookup(self,hashValue)`. Затем создайте большое хеш-кольцо с несколькими тысячами узлов и ресурсов и измерьте среднее число переходов, требуемых новым методом поиска. Сравните его с реализованным нами наивным поиском, выполняемым за линейное время.

Упражнение 3

При добавлении и удалении узлов пальцевые таблицы могут устаревать и нуждаться в перестройке. Модифицируйте реализацию хеш-кольца таким образом, чтобы пальцевые таблицы всегда оставались актуальными.

Резюме

- Хеш-таблицы незаменимы в современных системах, таких как сети, базы данных, программные решения по хранению данных, приложения по обработке текста и т. д. В зависимости от приложения и рабочей нагрузки хеш-таблицы могут конструироваться с учетом различных потребностей, таких как скорость против пространства, простота против оптимизации наихудшего случая и т. д.
- Существует большое число методов урегулирования коллизий, но наиболее часто используемыми из них являются прохождение по цепочкам и линейное опробывание (раздел 2.5). Линейное опробывание имеет преимущества в части эффективности кеширования. Чем больше хеш-таблица, тем больше результат эффективности кеширования, по сравнению с эффектом числа проб на производительность.

- Большинство хеш-таблиц производственного качества, таких как `dict` в языке Python (раздел 2.6), предназначены для оптимизации общего случая и не ориентированы на решение редких патологических случаев, если они усложняют общий случай.
- MurmurHash (раздел 2.7) является примером широко используемой быстрой и простой некриптографической хеш-функции, часто используемой в хеш-ориентированных структурах данных, о которых мы узнаем из этой книги.
- Согласованное хеширование (раздел 2.8) решает задачу распределения хеш-таблиц между многочисленными машинами, как это имеет место в одноранговых средах. Согласованное хеширование было реализовано во многих одноранговых продуктах, таких как BitTorrent, а также в системах хранения данных, таких как Amazon Dynamo.

Глава 3

Приближенная принадлежность: блумовские и порционные фильтры

Эта глава охватывает следующие ниже темы:

- ознакомление с фильтрами Блума, причинами их полезности и вариантами применения;
- настройка фильтра Блума в практических условиях;
- изучение взаимодействия параметров фильтра Блума;
- обследование порционных фильтров в качестве замены фильтров Блума;
- сравнение производительности блумовского и порционного фильтров.

Фильтры Блума стали стандартом в системах обработки крупных наборов данных. Их широкое применение, в особенности в сетях и распределенных базах данных, обусловлено эффективностью, которую они демонстрируют в ситуациях, когда требуется функциональность хеш-таблицы, но при этом нет такой роскоши, как свободное пространство. Они были изобретены в 1970-х годах Бертоном Блумом (Burton Bloom) [1], но по-настоящему «расцвели» только в последние несколько десятилетий из-за растущей потребности в укрощении и сжатии больших наборов данных. Фильтры Блума также вызвали интерес у сообщества исследователей в области вычислительных наук, которое разработало множество вариантов поверх базовой структуры данных, чтобы устранить некоторые недостатки фильтров и адаптировать их под разные контексты.

Фильтры Блума можно трактовать как механизм, который поддерживает вставку и поиск таким же образом, как и хеш-таблицы, но использует очень мало пространства (в частности, 1 байт на элемент или меньше). Это значительная экономия в ситуациях, когда ключи занимают 4–8 байт. Фильтры Блума не хранят сами элементы и занимают меньше места, чем нижний

теоретический предел, необходимый для правильного хранения данных; следовательно, они демонстрируют высокую частоту ошибки. У них есть ложноположительные результаты, но у них нет ложноотрицательных, и такая односторонность ошибки используется в наших интересах. Когда фильтр Блума сообщает об элементе как `Найден/Присутствует`, есть небольшая вероятность, что он говорит неправду, но когда он сообщает об элементе как `Не найден / Не присутствует`, мы знаем, что он говорит правду. В ситуациях, когда бóльшую часть времени ожидается, что ответом на поисковый запрос будет `Не присутствует`, фильтры Блума обеспечивают высокую точность и пространственно-экономичные выгоды.

Например, фильтры Блума применяются в наиболее широко используемых системах распределенного хранения, предназначенных для обработки массивных данных, таких как Google WebTable [2] и Apache Cassandra [3]. В частности, эти системы организовывают свои данные в ряд таблиц, именуемых *таблицами сортированных строк* (SST)[19], которые размещаются на диске и структурируются как соотнесения ключ-значение (например, ключом может быть URL-адрес, а значениями – атрибуты веб-сайта или содержимое). WebTable и Cassandra одновременно обрабатывают добавление нового содержимого в таблицы и отвечают на запросы, и при поступлении поискового запроса важно локализовывать таблицу сортированных строк, содержащую запрашиваемое содержимое, без явного обращения к каждой таблице. С этой целью в оперативной памяти поддерживаются специальные фильтры Блума, по одному на таблицу, чтобы маршрутизировать поисковый запрос к нужной таблице.

В примере, показанном на рис. 3.1, 50 таблиц сортированных строк размещено на диске, 50 связанных с ними фильтров Блума – в оперативной памяти. Как только фильтр Блума сообщает `Не присутствует` на операции поиска, запрос маршрутизируется к следующему фильтру Блума. При первом сообщении фильтром Блума о `Присутствии` элемента (в данном примере фильтр Блума таблицы сортированных строк) мы переходим на диск, чтобы выполнить проверку на присутствие элемента в таблице. В случае ложной тревоги мы продолжаем маршрутизировать поисковый запрос до тех пор, пока фильтр Блума не сообщит `Присутствует` и данные также не будут найдены на диске и возвращены пользователю, как в случае с SST50.

Фильтры Блума наиболее полезны при стратегическом размещении в системах интенсивного приема данных. Например, выполнение приложением операций чтения/записи на твердотельном носителе/диске может легко снижать пропускную способность приложения с сотен тысяч операций в секунду до всего лишь пары тысяч или даже пары сотен операций в секунду. Если вместо этого в оперативной памяти размещать фильтр Блума, чтобы обслуживать операции поиска, то такого падения производительности

[19] Англ. sorted-string table (SST); это файловый формат долговременного хранения, используемый ScyllaDB, Apache Cassandra и другими базами данных NoSQL, чтобы брать хранящиеся в memtables резидентные данные, упорядочивать их с целью быстрого доступа и сохранять на диске в виде долговременного, упорядоченного, немутируемого набора файлов. – *Прим. перев.*

можно избегать и поддерживать стабильно высокую пропускную способность разных компонентов приложения. Более подробная информация о способах интеграции фильтров Блума в общую картину систем обработки потоковых данных будет приведена в главе 6.

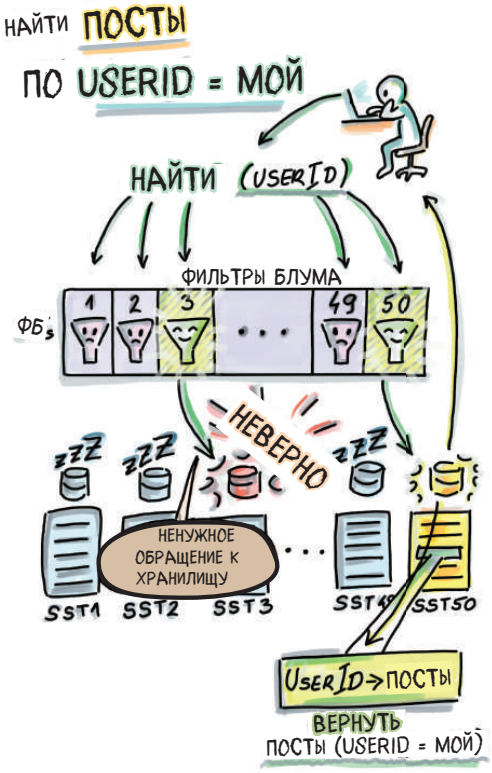


Рисунок 3.1 Фильтры Блума в системах распределенного хранения. В данном примере на диске размещено 50 таблиц сортированных строк (SST), каждая из которых имеет специальный фильтр Блума, могущий уместиться в оперативной памяти из-за его гораздо меньшего размера. Когда пользователь выполняет поиск, данная операция сначала сверяется с фильтрами Блума, что позволяет избегать дорогостоящих операций перепозиционирования головки чтения/записи на диске во время поиска

В этой главе вы познакомитесь с принципом работы фильтров Блума и ситуациями, когда их следует использовать, на примере различных практических сценариев. Вы также научитесь конфигурировать параметры фильтра Блума в своем конкретном приложении. Существует интересная взаимосвязь между пространством (m), числом элементов (n), числом хеш-функций (k) и ложноположительной частотой (f). Для читателей, более ориентированных на математику, мы потратим некоторое время на понимание того, откуда берутся формулы, связанные с важными параметрами фильтра Блума, и можно ли добиться большего, чем фильтр Блума.

В разделе 3.7 мы разведаем совершенно иную структуру данных, функционально аналогичную фильтрам Блума. *Порционный фильтр* [4] – это компактная хеш-таблица, которая обеспечивает пространственно-экономичные выгоды за счет ложноположительных результатов, но также обладает другими преимуществами. Если вы уверены в своих знаниях о фильтрах Блума, то смело можете перелистать до раздела 3.7.

3.1 Принцип работы

Фильтры Блума состоят из двух главных компонентов:

- битового массива `A[0..m-1]`, все слоты которого изначально установлены равными 0;
- k независимых хеш-функций h_1, h_2, ..., h_k, каждая из которых равномерно случайно отображает ключи в диапазон $[0, m-1]$.

3.1.1 Вставка

При вставке элемента x в фильтр Блума мы сначала вычисляем k хеш-функций на x и для каждого результирующего хеша устанавливаем соответствующий слот битового массива `A` равным 1 (см. псевдокод и рис. 3.2):

```
Bloom_insert(x):
    for i ← 1 to k
        A[hi(x)] ← 1
```

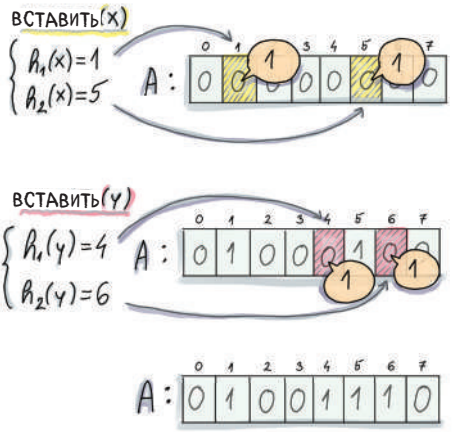


Рисунок 3.2 Пример вставки в фильтр Блума. В данном примере изначально пустой фильтр Блума имеет $m = 8$ и $k = 2$ (две хеш-функции). При вставке элемента x мы сначала вычисляем два хеша на x, первый из которых генерирует 1, а второй 5. Далее мы устанавливаем `A[1]` и `A[5]` равными 1. При вставке y мы также вычисляем хеши и аналогичным образом устанавливаем позиции `A[4]` и `A[6]` равными 1

3.1.2 Поиск

Поиск, аналогично вставке, вычисляет k хеш-функций на x, и в первый раз, когда один из соответствующих слотов битового массива A равен 0, указанная операция сообщает `Не присутствует`; в противном случае она сообщает `Присутствует`:

```
Bloom_lookup(x):
    for i ← 1 to k
        if(A[hi(x)] = 0)
            return NOT PRESENT
    return PRESENT
```

На рис. 3.3 показан пример поиска в результирующем фильтре Блума из рис. 3.2 и то, как он может генерировать истинно положительные результаты (на элементе x, который действительно был вставлен) и ложноположительные результаты (на элементе z, который не был вставлен).

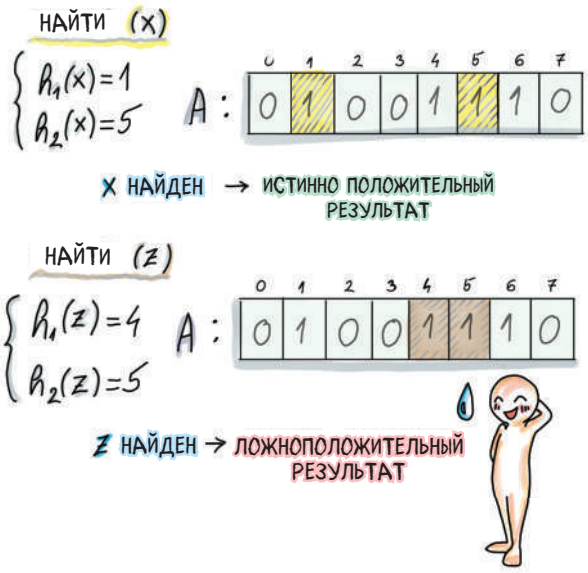


Рисунок 3.3 Пример поиска в фильтре Блума. При поиске элемента x мы вычисляем хеши (то есть такие же, как в случае вставки) и возвращаем `Найдено/Присутствует`, поскольку оба бита в соответствующих позициях равны 1. Затем выполняем поиск элемента z, который мы не вставляли, и его хеши равны соответственно 4 и 5, а биты в позициях `A[4]` и `A[5]` равны 1; следовательно, мы снова возвращаем `Найдено/Присутствует`

Как видно на рис. 3.3, ложноположительные результаты могут возникать, когда некоторые элементы вместе устанавливают биты какого-то другого элемента равными 1 (в этом примере два предыдущих элемента, x и y, установили битовые позиции элемента z равными 1).

В асимптотическом плане операция вставки в фильтре Блума стоит O(k). Учитывая, что число хеш-функций редко превышает 12, указанная операция является *постоянно-временной*. Для операции поиска тоже может потребоваться O(k), но только в случае, если операция должна проверять все биты. Однако большинство операций неуспешного поиска будут завершаться задолго до этого; мы увидим, что в среднем неуспешный поиск в хорошо сконфигурированном фильтре Блума занимает от одной до двух проб, прежде чем отказаться от продолжения, вследствие чего обеспечивается невероятно быстрая операция поиска.

3.2 Варианты использования

Во введении к главе мы рассмотрели применение фильтров Блума в системах распределенного хранения. В данном разделе мы увидим еще несколько применений фильтров Блума в распределенных сетях: сетевом прокси-сервере Squid и мобильном приложении для биткоинов.

3.2.1 Фильтры Блума в сетях: Squid

Squid – это кеш-память веб-прокси[20]. В веб-прокси кеши используются для уменьшения веб-трафика, поддерживая локальную копию недавно посещенных ссылок, по которым они могут обслуживать запросы клиентов на веб-страницы, файлы и т. д. Один из протоколов [5] предполагает, что каждый прокси также хранит сводную информацию о содержимом кеша соседних прокси и проверяет итоговые данные перед пересылкой любых поисковых запросов соседним прокси. В Squid эта функция реализована с помощью фильтров Блума, именуемых дайджестами кеша (**https://wiki.squid-cache.org/SquidFaq/AboutSquid**) (см. рис. 3.4).

Содержимое кеша быстро устаревает, и фильтры Блума эпизодически транслируются между соседями. Поскольку фильтры Блума не всегда актуальны, могут возникать ложноотрицательные результаты, когда фильтр Блума утверждает о присутствии элемента в прокси, но прокси уже не содержит ресурс.

3.2.2 Мобильное приложение для биткоинов

Фильтры Блума используются в одноранговых сетях для обмена данными, и хорошо известным примером тому является Биткойн. Важной особенностью Биткойна является обеспечение прозрачности между клиентами, которая поддерживается за счет того, что каждый узел находится в курсе всех транзакций. Однако хранить копии всех транзакций в узлах, работающих со смартфона или аналогичного устройства с ограниченной памятью и пропускной способностью, крайне непрактично, поэтому Биткойн

[20] Веб-прокси (web proxy) – это процесс, предоставляющий кеш-память для элементов, имеющихся на других серверах, доступ к которым предположительно медленнее или дороже. – *Прим. перев.*

предлагает опцию упрощенной верификации платежей[21], при которой узел может становиться *облегченным узлом*, рекламируя список транзакций, в которых он заинтересован. Это контрастирует с полными узлами, которые содержат все данные (рис. 3.5).

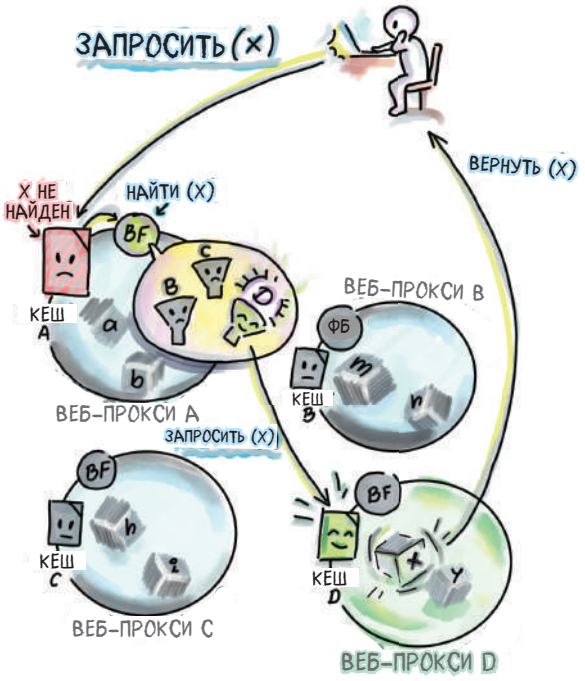


Рисунок 3.4 Фильтр Блума в веб-прокси Squid. Пользователь запрашивает веб-страницу *x*, а веб-прокси *A* не может ее найти в своем собственном кеше, поэтому он локально опрашивает блумовские фильтры *B*, *C* и *D*. Блумовский фильтр *D* сообщает Присутствует, поэтому запрос перенаправляется в *D*. Ресурс найден в *D* и возвращается пользователю

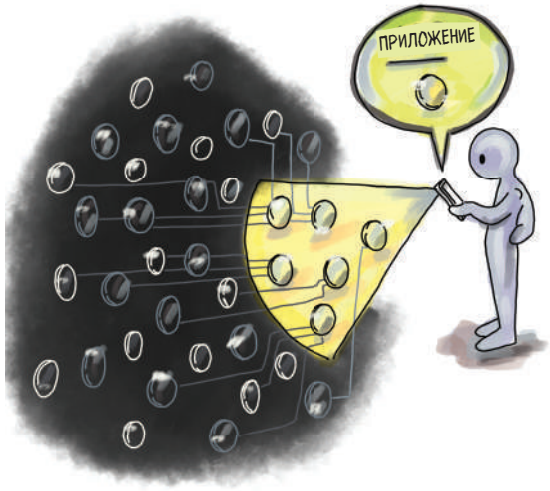


Рисунок 3.5 Облегченные клиенты в рамках технологии Биткойна могут широковещательно транслировать информацию о том, какие транзакции их интересуют, и тем самым блокировать лавину обновлений из сети

[21] Англ. simplified payment verification (SPV). – *Прим. перев.*

Облегченные узлы вычисляют и передают полным узлам блумовский фильтр списка транзакций, в которых они заинтересованы. Благодаря этому, перед тем как полный узел отправляет информацию о транзакции облегченному узлу, он сначала обращается к своему фильтру Блума, чтобы проверить заинтересованность в нем узла. Если происходит ложноположительный результат, то облегченный узел может отбросить информацию по ее прибытии [6].

В рамках технологии Биткойна совсем недавно были предложены и другие методы транзакционной фильтрации с улучшенными свойствами безопасности и конфиденциальности.

3.3 Простая реализация

Базовый фильтр Блума реализуется довольно просто. Мы покажем простую реализацию, использующую Python'овскую обертку хеш-функции MurmurHash, `mmh3`, которую мы обсуждали в главе 2. Установив *k* разных начальных позиций генератора псевдослучайных чисел, можно получить *k* разных хеш-функций. В данной реализации также используется библиотека `bitarray`, обеспечивающая возможность пространственно-эффективного кодирования фильтра, которую необходимо установить, чтобы обеспечить работу исходного кода:

```
import math
import mmh3
from bitarray import bitarray

class BloomFilter:
    def __init__(self, n, f):    ❶
        self.n = n
        self.f = f
        self.m = self.calculateM()
        self.k = self.calculateK()

        self.bit_array = bitarray(self.m)
        self.bit_array.setall(0)
        self.printParameters()

    def calculateM(self):
        return int(-math.log(self.f)*self.n/(math.log(2)**2))

    def calculateK(self):
        return int(self.m*math.log(2)/self.n)

    def printParameters(self):
        print("Параметры инициализации:")
        print(f"n = {self.n}, f = {self.f}, m = {self.m}, k = {self.k}")
```

```
    def insert(self, item):
        for i in range(self.k):
            index = mmh3.hash(item, i) % self.m
            self.bit_array[index] = 1

    def lookup(self, item):
        for i in range(self.k):
            index = mmh3.hash(item, i) % self.m
            if self.bit_array[index] == 0:
                return False

        return True
```

❶ Указать число и желаемую частоту ложноположительных результатов

Вы можете опробовать реализацию, вставив пару элементов типа `string`:

```
bf = BloomFilter(10, 0.01)
bf.insert("1")
bf.insert("2")
bf.insert("42")
print("1 {}".format(bf.lookup("1")))
print("2 {}".format(bf.lookup("2")))
print("3 {}".format(bf.lookup("3")))
print("42 {}".format(bf.lookup("42")))
print("43 {}".format(bf.lookup("43")))
```

Конструктор приведенного выше образца реализации позволяет пользователю устанавливать максимальное число элементов (n) и желаемую ложноположительную частоту (f), а сам конструктор берет на себя установку двух других параметров (m и k). Это распространенный подход, поскольку мы зачастую знаем величину набора данных, с которым имеем дело, и ложноположительную частоту, которую мы готовы допустить. Далее речь пойдет о задании остальных параметров этой реализации и конфигурировании фильтра Блума, чтобы получить максимальную отдачу. Читайте дальше.

3.4 Конфигурирование фильтра Блума

Сначала мы опишем главные формулы, связанные с важными параметрами фильтра Блума. Для четырех параметров фильтра Блума используются следующие ниже обозначения:

- n = число вставляемых элементов;
- f = частота ложноположительных результатов;
- m = число битов в фильтре Блума;
- k = число хеш-функций.

Формула, определяющая ложноположительную частоту как функцию из трех других параметров, выглядит следующим образом (уравнение 3.1):

$$f \approx \left(1 - e^{-\frac{nk}{m}}\right)^k. \qquad \text{(Уравнение 3.1)}$$

Если вы хотите понять, как эта формула выводится, то более подробная информация приведена в разделе 3.5. Прямо сейчас нас больше интересует визуальное обоснование этой формулы.

На рис. 3.6 показан график *f* как функции от *k* для разных вариантов *m*/*n* (биты в расчете на элемент). Во многих реальных приложениях важно фиксировать соотношение битов на элемент. Значения соотношения битов на элемент обычно находятся в диапазоне от 6 до 14, и такие соотношения позволяют получать довольно низкие ложноположительные частоты, как показано на рис. 3.6.

Начиная с верхней части кривой и в сторону к нижней мы имеем соответственно 6, 8, 10, 12 и 14 в качестве вариантов соотношения битов на элемент. По мере увеличения соотношения ложноположительная частота падает при том же числе хеш-функций. Кроме того, кривые демонстрируют следующий тренд: увеличение *k* до некоторой точки (слева направо) при фиксированном *m*/*n* сокращает ошибку, но после некоторой точки увеличение *k* увеличивает ошибку. Этот двоякий эффект возникает в связи с тем, что наличие большего числа хеш-функций дает операции поиска больше шансов найти 0, но устанавливает большее число битов равным 1 во время вставки. Следовательно, форма кривых указывает на то, что лучше ошибаться с большей стороны.

Кривые довольно плавные, и при *m*/*n* = 8 (то есть мы готовы потратить 1 байт на элемент), например, если мы используем где-то от 4 до 8 хеш-функций, ложноположительная частота не будет превышать 3 %, даже если оптимальный вариант *k* находится между 5 и 6.

Минимальная ложноположительная частота для каждой кривой дает оптимальное значение *k* для определенного соотношения битов на элемент (которое мы получаем, беря производную уравнения 3.1 относительно *k*; см. уравнение 3.2):

$$k_{opt} = \frac{m}{n} \ln 2. \qquad \text{(Уравнение 3.2)}$$

Например, при *m*/*n* = 8 мы имеем k_{opt} = 5.545. Эту формулу можно использовать для оптимального конфигурирования фильтра Блума, а интересным следствием выбора параметров в таком ключе является то, что в подобного рода фильтре Блума ложноположительная частота равна (уравнение 3.3):

$$f_{opt} = \left(\frac{1}{2}\right)^k. \qquad \text{(Уравнение 3.3)}$$

Уравнение 3.3 получается за счет подстановки уравнения 3.2 в уравнение 3.1. В рамках нашей реализации конструктор принимает значения *n* и *f* и использует их для вычисления *m* и *k* с помощью уравнений 3.2 и 3.3,

при этом обеспечивая, чтобы k и m были целыми числами. Если уравнение 3.2 производит нецелое число и приходится округлять в большую либо меньшую сторону, то уравнение 3.3 больше не является абсолютно точной ложноположительной частотой. Единственная правильная формула, в которую следует делать подстановку, чтобы получать точную ложноположительную частоту, приведена в уравнении 3.1, но даже если использовать уравнение 3.3, то возникающая при округлении в большую либо меньшую сторону разница будет незначительной. Нередко для сокращения объема вычислений лучше выбирать меньшее из двух возможных значений k.

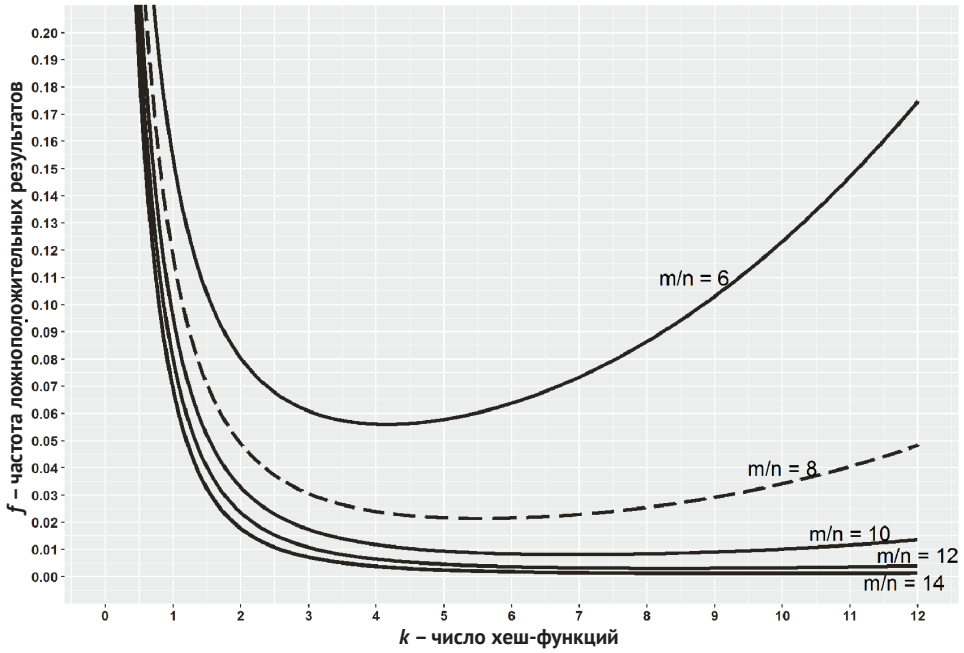


Рисунок 3.6 График, связывающий число хеш-функций (k) и ложноположительную частоту (f) в фильтре Блума. На графике показана ложноположительная частота при фиксированном соотношении битов на элемент (m/n), при этом разные кривые соответствуют разным соотношениям

Кому-то может показаться интересным выражение $(1/2)^k$ в уравнении 3.3 в связи с тем фактом, что ложноположительный результат возникает, когда поиск встречает k единиц подряд. Оптимально заполненный фильтр Блума и впрямь имеет примерно 50%-ную вероятность того, что случайный бит будет равен 1. Это просто еще один способ сказать, что если в вашем фильтре Блума слишком много нулей или единиц, то, скорее всего, он сконфигурирован неправильно.

В зависимости от предоставленных первоначальных параметров пишутся разные конструкторы фильтра Блума. Обычно k – это синтетический параметр, который вычисляется на основе других, более органических тре-

бований, таких как пространство, число элементов и ложноположительная частота. В любом случае, если вам когда-нибудь придется писать разные конструкторы фильтров Блума, вот пара примеров, которые показывают, как вычислять остальные параметры.

Пример 1: вычисление *f* из *m*, *n* и *k*

Вы пытаетесь проанализировать ложноположительную частоту уже существующего фильтра Блума, который первоначально был создан для хранения 10^6 элементов, но в итоге сохранил в 10 раз больше. Фильтр Блума дает очень слабую производительность, и нас интересует его ложноположительная частота. Емкость фильтра составляет 3 Мб, и в нем используется две хеш-функции.

Ответ

Используя уравнение 3.1, мы получаем следующее:

$$f = \left(1 - e^{-\frac{nk}{m}}\right)^k = \left(1 - e^{-\frac{2\times 10^7}{3\times 8\times 10^6}}\right)^2 = \left(1 - \left(\frac{1}{e}\right)^{\frac{5}{6}}\right)^2 \approx 32\%$$

Пример 2: вычисление *f* и *k* из *n* и *m*

Допустим, вы хотите построить фильтр Блума для $n = 10^6$ элементов, и для этого у вас есть около 1 Мб ($m = 8 * 10^6$) бит). Требуется найти оптимальную ложноположительную частоту и определить число хеш-функций.

Ответ

Из уравнения 3.2 следует, что идеальное число хеш-функций должно быть $k = ln2 * 8 * 10^6/10^6 = 5.544$. Уравнение 3.3 говорит о том, что ложноположительная частота равна $f \approx (1/2)^{5.544} \approx 0.0214$, но нам нужно легальное значение *k*. В этой ситуации мы могли бы выбрать $k = 5$ либо $k = 6$. В обоих случаях мы будем получать одинаковую 2%-ную ложноположительную частоту.

3.4.1 Работа с фильтрами Блума: мини-эксперименты

Теперь, когда у нас есть базовое представление о принципе работы фильтров Блума, ниже приводится пара мини-экспериментов, которые поднимут ваше понимание на следующий уровень.

Упражнение 1

Используйте предоставленную Python'овскую реализацию для создания фильтра Блума, где $n = 10^6$ и $f = 0.02$. Для элементов используйте случайные целые числа (без повторений) из равномерного распределения в диапазоне = $[0, 10^6]$ и конвертируйте их в строковые литералы (вставляйте их как строковые литералы). Сохраните вставленные элементы в отдельном файле.

Выполните 10^6 поисков элементов, равномерно случайно отобранных из *U*. Проследите за ложноположительной частотой и убедитесь, что она составляет ~ 2 %. Измерьте время, необходимое для выполнения операций поиска. Проверьте, чтобы в ваши измерения не было включено время, необходимое для генерации случайных чисел, связанных с выбором ключей.

Теперь выполните 10^6 поисков успешных элементов путем равномерного случайного отбора (без повторения) из файла вставленных элементов и измерьте время, необходимое для выполнения операций поиска. Проверьте, чтобы не было включено время, необходимое для чтения из файла или генерирования случайных чисел. Что занимает больше времени – операции поиска равномерных случайных элементов или операции поиска успешных элементов?

Упражнение 2

Используя предоставленную реализацию, создайте фильтр Блума, подобный приведенному в примере 2. Теперь создайте два других фильтра, один, в котором набор данных в 100 раз больше изначального, и еще один, в котором набор данных в 100 раз меньше, оставив ту же ложноположительную частоту. Что вы замечаете в размере фильтра при изменении размера набора данных?

Упражнение 3

В технической литературе описывается вариант фильтра Блума, в котором разные хеш-функции обладают «юрисдикцией» над различными частями фильтра Блума. Другими словами, *k* хеш-функций разбивают фильтр Блума на *k* идущих подряд блоков одинакового размера из *m*/*k* бит, и во время вставки *i*-я хеш-функция устанавливает биты в *i*-м блоке. Реализуйте этот вариант фильтра Блума и проверьте, сможет ли, и как, это изменение повлиять на ложноположительную частоту по сравнению с изначальным фильтром Блума.

Далее мы дадим несколько инструкций о выведении формулы ложноположительной частоты фильтра Блума и о работе нижних границ в компромиссе между пространством и ошибкой в работе структуры данных.

Следующий далее раздел носит теоретический характер и предназначен для математически ориентированных читателей. Если вы более ориентированы на практику, то можете свободно пролистать до раздела 3.6.

3.5 Немного теории

Прежде всего давайте посмотрим, откуда взята главная формула ложноположительной частоты фильтра Блума (уравнение 3.1). В приведенном ниже анализе мы исходим из допущения, что хеш-функции независимы (результаты одной хеш-функции не влияют на результаты любой другой

хеш-функции) и что каждая хеш-функция отображает ключи равномерно случайно в диапазон [0...m – 1].

Если t – это доля битов в фильтре Блума, которые по-прежнему равны 0 после всех n вставок, а k – число хеш-функций, то вероятность f ложноположительного результата равна

$$f = (1 - t)^k,$$

потому что нам нужно получить k единиц, чтобы сообщить `Присутствует`. Получение k единиц также может быть результатом успешного поиска фактически вставленного элемента; однако если мы рассматриваем запросы как равномерно случайно отбираемые из универсального множества, намного превышающего набор данных, то вероятность истинно положительного результата составляет ничтожную долю от этой величины.

Значение t невозможно узнать до того, как будут выполнены все вставки, потому что оно зависит от результата хеширования, но мы можем работать с *вероятностью* p того, что бит будет равен 0 после осуществления всех вставок (то есть p = Pr; фиксированный бит равен 0 после n вставок).

В вероятностном смысле значение p будет транслироваться в процент нулей в фильтре (t). Мы получаем значение p равным следующему ниже выражению:

$$p = \left(1 - \frac{1}{m}\right)^{nk} \approx e^{-\frac{nk}{m}}.$$

Для того чтобы понять причину справедливости этого выражения, давайте начнем с пустого фильтра Блума. Сразу после того, как первая хеш–функция h_1 установила один бит равным 1, вероятность того, что фиксированный бит в фильтре Блума будет равен 1, будет равна $1/m$, а вероятность того, что он равен 0, соответственно, равна $1 - 1/m$.

После того как все хеши первой вставки закончили устанавливать значения битов равными 1, вероятность того, что фиксированный бит по-прежнему будет равен 0, будет равна $(1 - 1/m)^k$, и после того, как мы закончили вставлять весь набор данных размера n целиком, эта вероятность будет равна $(1 - 1/m)^{nk}$. Далее аппроксимация $(1 - 1/x)^x \approx e$ дает $p \approx e^{-nk/m}$.

Возникает соблазн просто взять это выражение и заменить в нем t, обозначающее ложноположительную частоту, на p, и это даст нам уравнение 3.1. В конце концов, p описывает ожидаемое значение случайной величины, обозначая процент нулей в фильтре, но что, если фактический процент нулей может существенно варьироваться от его математического ожидания?

Используя *границы Чернова* – теорему, ограничивающую вероятность существенного отклонения случайной величины от ее среднего значения, – можно показать, что доля нулей в фильтре Блума сильно сосредоточена вокруг его среднего значения. Общая формулировка границ Чернова соблюдается для случайных величин X, представляющих собой сумму

взаимно независимых индикаторных случайных величин. Случайная величина X, которая обозначает общее число нулей в фильтре Блума, определяется как $X = \sum_{i=1}^{m} X_i$, где $X_i = 0$, если i-й бит в фильтре Блума равен 1, и $X_i = 1$ в противном случае.

Используя границы Чернова, мы покажем, что значение случайной величины X существенно не отклоняется от ее среднего значения. В нашем случае X_i не является независимой, однако она слегка отрицательно коррелирует (даже еще лучше!). Установка одного бита равным 1 немного снижает вероятность того, что другие биты будут установлены равными 1.

Общая формулировка верхней границы Чернова (что-то подобное можно сделать и для нижней границы), где μ – это среднее значение случайной величины X, выглядит следующим образом:

$$Pr[X > (1+\delta)\mu] < \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}.$$

Применительно к нашему случаю $\mu = E[X] = mp = me^{-nk/m}$. Если мы выберем $\delta = 1$ и подставим границу Чернова, то получим вероятность того, что X будет отклоняться от своего среднего значения более чем в 2 раза:

$$Pr[X > 2pm] < \left(\frac{e}{4}\right)^{me^{-\frac{nk}{m}}}.$$

Мы можем с уверенностью допустить, что $nk = \theta(m)$, и это будет дополнительно говорить об экспоненциально малой ($< (3/4)^{\theta(m)}$) вероятности того, что X будет отклоняться от среднего значения более чем в 2 раза. Следовательно, p является хорошей аппроксимацией процента нулей в фильтре Блума, t, что оправдывает замену t в формуле в начале этого раздела. На этом мы завершаем формальное выведение уравнения 3.1.

3.5.1 Можно ли добиться большего?

Фильтры Блума и впрямь хорошо упаковывают пространство, но существуют ли или могут ли существовать более оптимальные структуры данных? Другими словами, при том же объеме пространства можно ли добиться более оптимальной ложноположительной частоты? Для ответа на этот вопрос нужно формально вывести *нижнюю границу*, которая связывает объем пространства, используемого структурой данных в битах (m), с максимальной ложноположительной частотой, допускаемой структурой данных (f). Обратите внимание, что нижняя граница не зависит от какого-либо конкретного устройства структуры данных и говорит о теоретических пределах *любой* структуры данных – даже той, которая еще не была изобретена.

Структура данных представляет собой m-битовую строку и имеет в общей сложности 2^m несовпадающих кодировок. Каждая отдельная кодировка структуры данных в дополнение к сообщению `Присутствует` для некоторых n элементов также допускает $f(U - n)$ ложноположительных результатов, то есть долю f остальной части универсального множества. Из общего числа

$n + f(U - n)$ элементов, для которых структура данных сообщает `Присутствует`, мы не знаем, какие из них являются истинно положительными, а какие – ложноположительными, поэтому одна кодировка структуры данных служит для представления каждого подмножества размера n, которое мы захватываем. Таких множеств имеется $n + f(U - n)$ по n.

В целом структура данных должна уметь «охватывать» все (U по n) множеств размера n в универсальном множестве U. Вместе взятые, эти факты дают нам неравенство, показанное на рис. 3.7, которое описывает нижнюю границу.

Рисунок 3.7 Неравенство, описывающее нижнюю границу пространства-ошибки

Взяв логарифм с обеих сторон, наряду с тем фактом, что $U >> n$, и несколькими дополнительными алгебраическими манипуляциями неравенство из рис. 3.7 даст нам

$$m \geq n \log_2 \left(\frac{1}{f} \right).$$

Как это соотносится с ложноположительной частотой фильтра Блума?

Из уравнений 3.2 и 3.3 можно формально вывести взаимосвязь между m и ложноположительной частотой фильтра Блума (здесь мы будем называть его $f_{\text{ФБ}}$). Опять же, взяв логарифм и выполнив несколько алгебраических упрощений, мы получаем, что

$$m \geq n \log_2 \left(\frac{1}{f_{BF}} \right) \log_2 e.$$

Сравнив это выражение с нижней границей, мы получаем, что пространство фильтра Блума находится на расстоянии $\log_2 e \approx 1.44$ от оптимального. Некоторые структуры данных находятся ближе к нижней границе, чем фильтр Блума, но их очень трудно понять и реализовать.

3.6 Адаптации и альтернативы фильтров Блума

Базовая структура данных фильтра Блума широко используется в ряде систем, но фильтры Блума также оставляют желать лучшего, и специалисты по информатике разработали различные модифицированные версии фильтров Блума, которые устраняют некоторые из этих недостатков. Например, стандартный фильтр Блума не работает с удалениями. Существует версия фильтра Блума, именуемая *считающим фильтром Блума*[22] [7], в которой вместо отдельных битов в ячейках используются счетчики. Операция вставки в считающем фильтре Блума увеличивает соответствующие счетчики, а операция удаления их уменьшает. Считающие фильтры Блума занимают больше места (примерно в четыре раза больше) и тоже могут приводить к ложноотрицательным результатам; например, при многократном удалении одного и того же элемента, тем самым сводя счетчики какого-либо элемента к нулю.

Еще одной проблемой фильтров Блума является их неспособность эффективно изменять размер. В фильтре Блума не хранятся ни элементы, ни отпечатки, поэтому изначальные ключи необходимо возвращать из долговременного хранилища, чтобы строить новый фильтр Блума.

Кроме того, фильтры Блума уязвимы, когда запросы не делаются равномерно случайно. Запросы в реальных сценариях редко бывают равномерно случайными. Напротив, многие запросы подчиняются распределению Ципфа, где малое число элементов запрашивается большое число раз, а большое число элементов запрашивается только один или два раза. Такая закономерность запросов может увеличивать эффективную ложноположительную частоту, если один из «горячих» элементов (то есть часто запрашиваемых элементов) приводит к ложноположительному результату. Модификация фильтра Блума, именуемая *взвешенным фильтром Блума* [8], устраняет эту проблему, выделяя больше хешей «горячим» элементам, тем самым снижая возможность ложноположительного результата для этих элементов. Существуют также новые *адаптивные* модификации фильтров Блума (то есть при обнаружении ложноположительного результата они пытаются его исправить) [9].

Еще одно направление исследований было сосредоточено на конструировании структур данных, функционально аналогичных фильтру Блума, но их устройство было основано на определенных типах компактных хеш-таблиц. В следующем далее разделе мы рассмотрим одну из таких интерес-

[22] Англ. counting Bloom filter. – *Прим. перев.*

ных структур данных: *порционный фильтр*. Некоторые методы, задействованные в следующем ниже разделе, тесно связаны с конструированием хеш-таблиц для массивных наборов данных, что является темой предыдущей главы, но мы рассмотрим этот вопрос здесь, потому что главные области применения порционных фильтров функционально эквивалентны фильтрам Блума, и мы находим их применение в аналогичных контекстах.

3.7 Порционный фильтр

Порционный фильтр[23] [10] в самом простом виде представляет собой хитроумно разработанную хеш-таблицу, в которой используется линейное опробывание. Разница между порционным фильтром и обычной хеш-таблицей заключается в том, что вместо хранения ключей в слотах, как в классической хеш-таблице с использованием линейного опробывания, порционный фильтр хранит хеши (термин *отпечаток* будет использоваться взаимозаменяемо для обозначения хеша). Точнее, порционный фильтр хранит порцию каждого хеша, но, как мы увидим, он способен надежно восстанавливать весь хеш целиком.

В «спектре неискаженности» порционный фильтр находится где-то между хеш-таблицей и фильтром Блума. Если два несовпадающих ключа хешируются в один и тот же отпечаток, то порционный фильтр не сможет их отличить друг от друга так, как это сделала бы хеш-таблица. Но если два ключа хешируются в разные отпечатки, то порционный фильтр сможет их отличить; это не относится к фильтрам Блума, где запрос по ключу с уникальным набором k хешей может генерировать ложноположительный результат.

Порционный фильтр обладает функциональностью, аналогичной с фильтром Блума, но имеет совершенно иное устройство. Используя более длинные отпечатки, порционные фильтры могут снижать ложноположительную частоту, но более длинные отпечатки также могут занимать слишком много пространства.

В данном разделе мы рассмотрим разные приемы, которые используются в порционном фильтре для компактного хранения отпечатков. Возможность восстанавливать полный отпечаток приходит на помощь, когда возникает потребность в удалении элементов; да, порционные фильтры способны эффективно удалять.

В дополнение к этому порционный фильтр может сам изменять свой размер, а операция слияния двух порционных фильтров в более крупный порционный фильтр проходит бесшовно и быстро. Эффективное слияние, изменение размера и удаление – все эти функциональности, возможно, побудят вас рассмотреть возможность использования порционного фильтра в некоторых приложениях вместо фильтра Блума; эти функциональности

[23] Англ. quotient filter; син. частный фильтр; он основан на своего рода хеш-таблице, в которой записи содержат только *порцию* ключа плюс некие дополнительные биты метаданных. См. **https://en.wikipedia.org/wiki/Quotient_filter**. – *Прим. перев.*

особенно удобны в динамических распределенных системах. По правде говоря, порционные фильтры немного сложнее понять и реализовать, чем фильтры Блума, но, по нашему мнению, их изучение стоит вашего времени.

В следующих далее разделах мы проведем обследование устройства порционного фильтра, сначала познакомившись с понятием формирования частных и остатков, а затем описав применение указанной процедуры порционного фильтра вместе с битами метаданных для экономии пространства. На порционный фильтр лучше всего смотреть как на игру, в которой нужно немного сэкономить здесь или там и использовать с этой целью разные уловки. Порционный фильтр – не единственная структура данных такого рода, но некоторые приемы, которым вы здесь научитесь, будут в целом полезны для понимания аналогичных структур данных, основанных на компактных хеш-таблицах.

3.7.1 Формирование частных и остатков

Формирование частных и остатков[24] [11] подразделяет хеш каждого элемента на *частное* и *остаток*: в порционном фильтре частное используется для индексации в соответствующую корзину хеш-таблицы, а остаток – это то, что сохраняется в соответствующем слоте. Например, имея h-битовый хеш и размер таблицы $m = 2^q$, частное – это значение, определяемое крайними левыми q битами хеша, а остаток представляет оставшиеся $r = h - q$ бит. Пример на рис. 3.8 показывает разбивку отпечатков на небольшом примере, где $m = 32$ (поэтому $q = 5$) и $h = 11$.

Рисунок 3.8 Формирование частных и остатков в хеш-таблице. Элемент *y* имеет хеш `10100101101`; следовательно, остаток `101101 (35)` будет сохранен в слоте, определяемjм частным, в корзине `10100 (20)`

[24] Англ. quotienting. – *Прим. перев.*

Следующий ниже фрагмент исходного кода показывает, как после хеширования ключа и сохранения хеша в переменной `fingerprint` выполняется вставка в порционный фильтр (`filter`):

```
h = len(fingerprint)                          ❶

q = log2(m)                                   ❷

r = h - q

quotient = math.floor(fingerprint / 2**r)     ❸

remainder = fingerprint % 2**r                ❹

filter[quotient] = remainder                  ❺
```

❶ Число битов в отпечатке
❷ Предполагается, что размер хеш-таблицы равен степени двух
❸ *q* крайних левых бит отпечатка
❹ *r* крайних правых бит отпечатка
❺ Остаток сохраняется в корзине, задаваемой частным (частное не сохраняется)

Пока неплохо. Если коллизий не происходит (то есть никакие два отпечатка не имеют одинакового частного), каждый остаток занимает свою собственную корзину *b*. Полный отпечаток легко восстанавливается путем конкатенации двоичного представления номера корзины *b* с двоичным представлением остатка, хранящимся в корзине *b*. Даже в этом небольшом примере за счет формирования частных и остатков нам удалось сэкономить $q = 5$ бит на каждый слот.

Важно помнить, что отпечатки в порционном фильтре можно реконструировать, а это помогает при изменении размера и слиянии, но невозможно реконструировать изначальные элементы. Опять же, мы находимся где-то между хеш-таблицами, которые содержат фактические ключи, и фильтрами Блума, которые не могут реконструировать информацию о том, у какого элемента были какие хеши.

Однако коллизии в хеш-таблицах довольно распространены, и порционные фильтры урегулируют коллизии, используя вариант линейного опробывания. Мы уже чувствуем неладное, потому что вследствие линейного опробывания некоторые остатки будут задвигаться вниз от их изначальной корзины, что будет приводить к потере ассоциации между частным и остатком. Для реконструкции полного отпечатка в порционном фильтре используется три дополнительных бита метаданных на каждый слот. Три бита – это малая цена за экономию ~20–30 бит в каждом слоте на частном в крупных хеш-таблицах. В следующем далее разделе мы объясним, как биты метаданных облегчают операции в порционном фильтре.

3.7.2 Понятие битов метаданных

Прежде чем познакомиться со смыслом битов метаданных, необходимо немного разобраться в терминологии. Если вы запутались в терминах этого раздела, то не слишком волнуйтесь, поскольку все должно проясниться по мере того, как мы будем работать с примерами, посмотрим на предназначение отрезков и кластеров и на роль, которую играет каждый бит метаданных при урегулировании коллизий во время вставки или поиска.

Отрезок[25] – это непрерывная последовательность слотов порционного фильтра, занятых остатками (то есть отпечатками) с одинаковым частным (все отпечатки, которые вступили в коллизию в одной конкретной корзине). Все остатки с одинаковым частным сохраняются в фильтре поочередно и в отсортированном порядке остатков. Из-за коллизий и проталкивания остатков вниз при возникновении коллизий отрезок может начинаться сколь угодно далеко от соответствующей ему корзины.

Кластер – это последовательность из одного или нескольких отрезков. Это поочередная последовательность[26] слотов порционного фильтра, занятых остатками, где первый остаток хранится в его изначально хешированном слоте (такой слот называется *якорным*). Конец кластера обозначается либо пустым слотом, либо началом нового кластера.

При выполнении поиска в порционном фильтре необходимо декодировать остатки вместе с соответствующими битами метаданных, чтобы получить полные отпечатки. Декодирование всегда начинается в начале содержащего кластера (то есть в якорном слоте) и проходит в нисходящем направлении. Декодировать кластер помогают три бита метаданных в каждом слоте, которые приведены ниже[27]:

`bucket_occupied` – сообщает о том, был ли какой-либо ключ ранее хеширован в данную корзину. Его значение равно 1, если какой-либо ключ был хеширован в корзину, и 0 в противном случае. Этот бит сообщает о всех возможных частных в кластере;

`run_continued` – сообщает о наличии у остатка, хранящегося в данный момент в этом слоте, того же частного, что и у остатка прямо над ним. Другими словами, этот бит равен 0, если остаток идет первым в своем отрезке, и в противном случае равен 1. Этот бит сообщает о том, где начинается и заканчивается каждый отрезок в кластере;

`is_shifted` – сообщает о том, что остаток, хранящийся в данный момент в слоте, находится в своей изначальной предполагаемой позиции, либо о том, что он был сдвинут. Этот бит помогает локализовывать начало кластера. Он устанавливается равным 0 только в якорном слоте и в противном случае устанавливается равным 1.

---

[25] Англ. run; син. пробег. – *Прим. перев.*

[26] Англ. consecutive sequence; син. непрерывная последовательность. – *Прим. перев.*

[27] В порядке появления: корзина_занята, отрезок_продолжается и есть_сдвиг. – *Прим. перев.*

3.7.3 Вставка в порционный фильтр: пример

Теперь давайте проработаем вставку элементов *v*, *w* и *x*, которые можно увидеть на рис. 3.9.

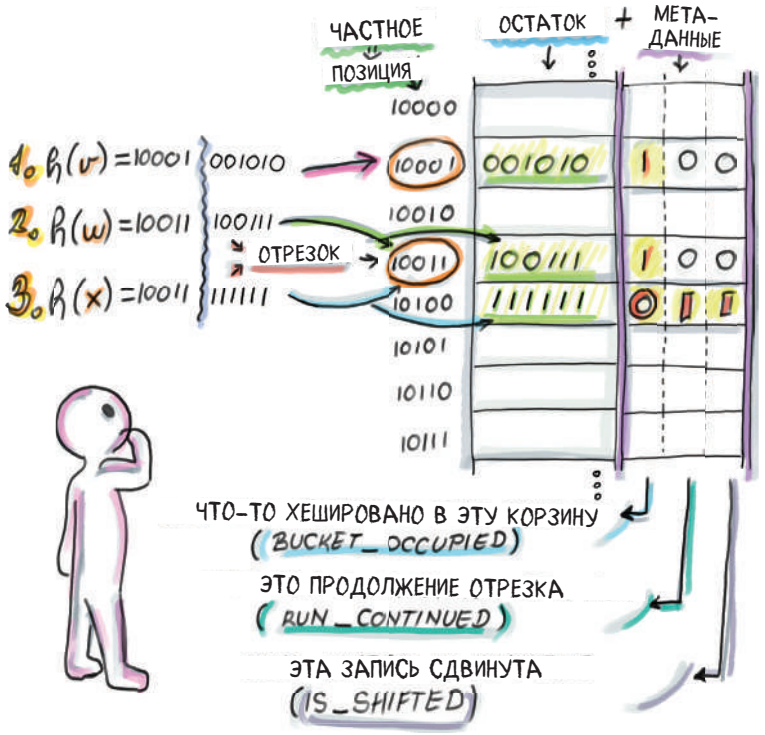


Рисунок 3.9 Вставка в порционный фильтр и биты метаданных

Мы начинаем с изначально пустого порционного фильтра из $32 = 2^5$ слотов, где каждый слот, а также все три бита метаданных изначально установлены равными 0.

1. Вставить *v*: *h*(*v*) = `10001001010`. Корзина, заданная частным `10001`, ранее не была занята. Мы устанавливаем бит `bucket_occupied` равным 1 и сохраняем остаток в слоте, заданном его частным. Обратите внимание, что нам не нужно никаких дополнительных действий с другими битами, так как этот элемент в настоящее время является началом отрезка и кластера.
2. Вставить *w*: *h*(*w*) = `10011100111`. Опять же, мы устанавливаем бит `bucket_occupied` равным 1 в `10011` и сохраняем остаток в соответствующем слоте, поскольку он доступен, и не изменяем никакие другие биты.
3. Вставить *x*: *h*(*x*) = `100111111111`. Пытаясь установить корзину равной 1, мы видим, что корзина `10011` уже занята, поэтому, где бы мы ни сохраняли остаток, нам нужно будет установить `run_continued` этого

слота равным 1. Слот в хешированной корзине занят, поэтому, где бы мы ни сохраняли остаток, нам также нужно будет установить бит `is_shifted` этого слота равным 1. Учитывая, что мы находимся в начале кластера, у которого есть только один отрезок (частное которого равно частному от x), мы сканируем вниз, чтобы найти первый доступный слот внутри отрезка в корзине `10100`. Мы сохраняем остаток и устанавливаем биты `run_continued` и `is_shifted` равными 1.

В настоящее время порционный фильтр имеет два кластера, каждый из которых имеет по одному отрезку. Теперь мы вставим еще несколько элементов (как показано на рис. 3.10).

- Вставить y: $h(x)$ = 10100101101. Бит `bucket_occupied` в 10100 был равен 0, и мы установили его равным 1 (`run_continued` в окончательном слоте остатка будет установлен равным 0), и слот в хешированной корзине будет занят (`is_shifted` в окончательном слоте остатка будет установлен равным 1). Мы ищем первое место начиная с начала кластера, чтобы сохранить новый отрезок. Первый доступный слот находится в корзине 10101, поэтому мы сохраняем остаток и соответствующим образом устанавливаем биты метаданных.
- Вставить z: $h(x)$ = 10100111110. Бит `bucket_occupied` в корзине z уже установлен равным 1 (`run_continued` в конечном слоте будет равен 1), и исходный слот занят (`is_shifted` в окончательном слоте будет равен 1). Мы декодируем с начала кластера и находим местоположение подходящего отрезка. Мы сканируем отрезок до корзины 10110, чтобы сохранить остаток z и соответствующим образом установить биты.

Наша последовательность вставок довольно упрощена, поскольку вставки поступают в отсортированном порядке значений отпечатков. Сценарий отсортированного порядка вставок в порционный фильтр получил широкое распространение из-за способа слияния и изменения размера порционных фильтров, аналогично слиянию сортированных списков при сортировке слиянием, но нам также нужно уметь работать со сценариями, когда вставляемые отпечатки поступают в произвольном порядке.

Когда элементы вставляются вне сортированного порядка отпечатков, то после того, как будет найден правильный отрезок для вставляемого туда остатка, это может привести к удалению нескольких элементов этого отрезка и других элементов в этом кластере. Рассмотрим пример вставки элемента a, $h(a)$ = 101000000000 в результирующий порционный фильтр, показанный на рис. 3.10. Элемент будет принадлежать второму отрезку второго кластера, который в настоящее время занимает слоты, заданные корзинами `10101` и `10110`. Элемент a переместил бы весь отрезок на один слот ниже, чтобы быть сохраненным в слоте `10101`, потому что его остаток является первым в возрастающем сортированном порядке в том отрезке, и, следовательно, также инициирует изменения в битах метаданных.

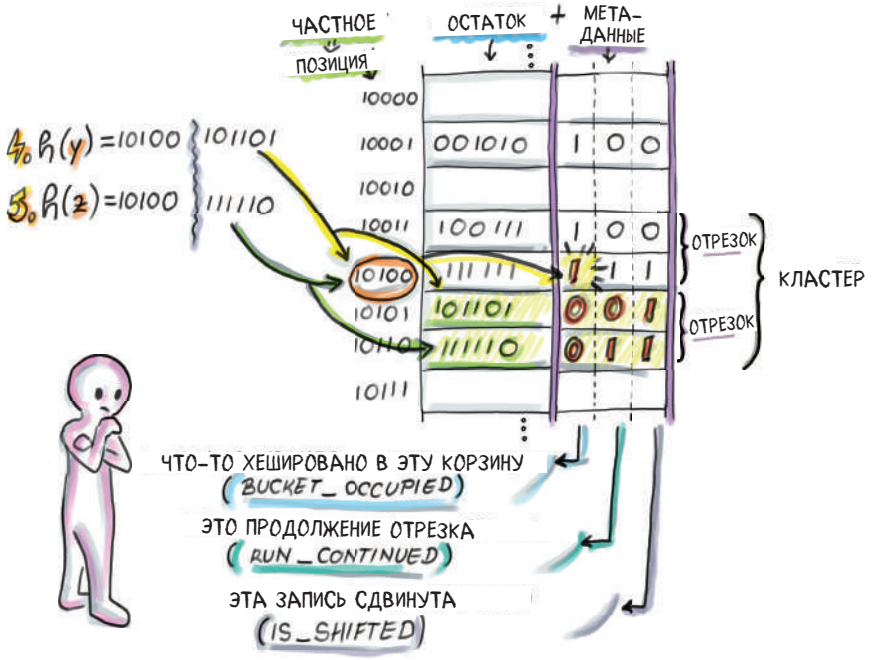


Рисунок 3.10 Вставка *y* и *z* в порционный фильтр

Зачем нужно иметь сортированные остатки внутри отрезка (и отрезки между кластерами)? Ответ на этот вопрос придет, когда мы поговорим об эффективном изменении размера и слиянии.

С хеш-таблицей связано еще одно важное замечание, заключающееся в том, что необходимость потенциального перемещения целого кластера элементов при вставке/удалении и декодировании целого кластера при выполнении поиска подчеркивает важность малых размеров кластера. Чем больше пустого пространства мы оставляем, тем меньше вероятность получения крупных кластеров, которые операции вставки и поиска должны сканировать и декодировать. Как и в случае с обычными хеш-таблицами с использованием линейного опробывания, порционные фильтры работают быстрее, когда коэффициент загруженности поддерживается на уровне 75–90 %, чем когда он выше.

3.7.4 Исходный код Python для поиска

Теперь, когда мы концептуально понимаем процедуру вставки, давайте посмотрим на работу процедуры поиска с использованием исходного кода на Python. Для понимания лежащей в основе логики на минутку отложим в сторону механику компактного хранения структуры данных, которая предусматривает множество операций побитовой распаковки и сдвига. Наша реализация класса `Slot` и класса `QuotientFilter` является «разреженной» в том смысле, что бит метаданных представляет собой целую булеву

переменную, поэтому он занимает более одного бита памяти. Наш исходный Python'овский код основан на псевдокоде из оригинальной статьи:

```
import math

class Slot:
    def __init__(self):
        self.remainder = 0
        self.bucket_occupied = False
        self.run_continued = False
        self.is_shifted = False

class QuotientFilter:
    def __init__(self, q, r):
        self.q = q
        self.r = r
        self.size = 2**q                                          ❶
        self.filter = [Slot() for _ in range(self.size)]

    def lookup(self, fingerprint):
        quotient = math.floor(fingerprint / 2**self.r)
        remainder = fingerprint % 2**self.r
        if not self.filter[quotient].bucket_occupied:
            return False                                          ❷
        b = quotient
        while(self.filter[b].is_shifted):                         ❸
            b = b - 1
        s = b

        while b != quotient:                                      ❹
            s = s + 1
            while self.filter[s].run_continued:                   ❺
                s = s + 1
            b = b + 1
            while not self.filter[b].bucket_occupied:             ❻
                b = b + 1
        while self.filter[s].remainder != remainder:              ❼
            s = s + 1
            if not self.filter[s].run_continued:
                return False
        return True
```

❶ Размер фильтра
❷ Ни один элемент никогда не хешировался в эту корзину
❸ Подняться к началу кластера
❹ b отслеживает занятые корзины, а s – соответствующие отрезки
❺ Спуститься вниз по отрезку и увеличить номер корзины
❻ Пропустить пустые корзины
❼ Теперь s указывает на начало отрезка, где может содержаться наш элемент

Процедура поиска начинается с отыскания начала кластера, в котором может содержаться искомый элемент. Другими словами, для того чтобы узнать о присутствии элемента, нужно декодировать весь кластер.

После того как мы нашли начало кластера, содержащего корзину отпечатка, для каждой занятой корзины мы находим соответствующий ему отрезок и пробегаем по этому отрезку до тех пор, пока не найдем корзину, которая равна частному отпечатка и началу его отрезка (если этого ни разу не произойдет, то мы возвращаем `False`). Найдя соответствующий отрезок, процедура поиска выполняет поиск остатка отпечатка внутри отрезка.

В исходном коде вставки также используется модифицированная версия процедуры поиска, которая возвращает не булево значение, а позицию запрашиваемого элемента. Исходный код начинается с пометки соответствующей корзины как занятой. Затем в нем используется алгоритм поиска, чтобы найти подходящее место для вставки остатка, что может потребовать сдвига других остатков вниз до тех пор, пока не будет достигнут пустой слот. Удаления работают аналогичным образом, когда им, возможно, потребуется сдвигать элементы вверх, чтобы заполнять дыру, образовавшуюся в результате удаления. Как вариант иногда удаления реализуются путем размещения сигнального элемента в указанной позиции, тем самым устраняя необходимость в перемещении остатков вверх-вниз. Если удаленный элемент является единственным элементом в своем отрезке, то нам также нужно снять пометку с бита `bucket_occupied`. По достижении конца таблицы все операции в порционном фильтре продолжаются с начала таблицы точно так же, как в обычных хеш-таблицах с использованием линейного опробывания.

Хранение порционного фильтра

Важной деталью реализации порционного фильтра, как и многих других типов компактных хеш-таблиц, является то, как данные располагаются в памяти. В частности, размер слота обычно не равен размерам единиц адресуемой памяти (например, байтам), поэтому границы байта и слота зачастую не совпадают. Например, предположим, что у нас порционный фильтр с остатком длины $r = 7$ бит, а также три бита метаданных (всего по 10 бит на слот). На рис. 3.11 показано расположение слотов порционного фильтра в памяти.

Например, при чтении бита `run_continued` слота 3 нужно обратиться к пятому биту байта 4. При декодировании кластера нужно выполнить множество операций побитового сдвига и побитовой распаковки, поэтому реализации порционного фильтра (чаще всего написанные на C) выменивают малое пространство на дополнительные затраты на центральный процессор, что, как мы увидим в разделе 3.8, влияет на производительность вставки и поиска внутри оперативной памяти для высоких коэффициентов загруженности в порционном фильтре. В отличие от вставки в рамках фильтра Блума, которая элегантно прыгает туда-сюда, устанавливая биты

равными 1, порционный фильтр бывает очень процессорно-интенсивным. Однако бывает, что это не представляет особой проблемы, поскольку операции порционного фильтра перемещаются по таблице последовательно, тогда как фильтр Блума страдает от слабой пространственной локализации.

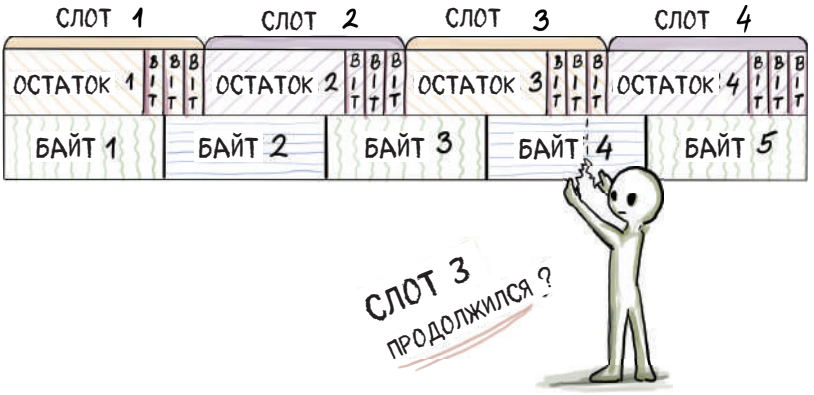


Рисунок 3.11 Схема расположения слотов порционного фильтра в памяти

3.7.5 Изменение размера и слияние

Если мы хотим удвоить размер порционного фильтра, то достаточно извлечь отпечаток, скорректировать размер частного и остатка, похитив один бит из остатка и передав его частному, и вставить новый отпечаток в порционный фильтр вдвое большего размера. В общем случае изменение размера выполняется путем обхода порционного фильтра в сортированном порядке, декодирования отпечатков по мере продвижения и вставки их в этом сортированном порядке в новый порционный фильтр. Быстрая операция добавления позволяет проноситься по порционному фильтру молниеносно, поскольку вставка в сортированном порядке не требует большого объема декодирования и перемещения остатков вверх-вниз. Простой пример изменения размера малого порционного фильтра размера 4 показан на рис. 3.12.

Слияние двух порционных фильтров выполняется аналогично процедуре изменения размера, в быстром линейно-временном стиле, как это делается с двумя сортированными списками в рамках сортировки слиянием, снова позволяя быстро добавлять в более крупный порционный фильтр.

Напомним, что выполнять слияние или изменение размера в фильтре Блума нет так просто, поскольку мы не консервируем ни изначальные элементы, ни отпечатки. Изменение размера фильтра Блума предусматривает сохранение изначального набора ключей и перезагрузку его в память, чтобы создавать новый фильтр Блума, что неосуществимо в быстро движущихся потоках и базах данных с интенсивным приемом данных.

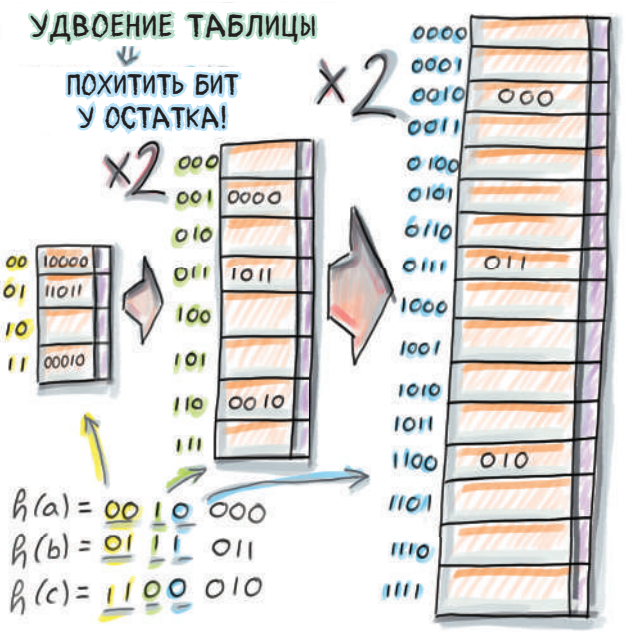


Рисунок 3.12 Изменение размера порционного фильтра размера 4, содержащего три элемента. Для простоты мы исходим из того, что между этими тремя элементами не произошло никаких коллизий и каждый остаток сохраняется в своей изначальной корзине. Отпечаток `110010`, который хешировался в корзину `11` (с остатком `0010`) в изначальном порционном фильтре, сохраняется в корзине `110` второго порционного фильтра (с остатком `0010`) и в корзине `1100` окончательного порционного фильтра (с остатком `010`)

3.7.6 Соображения по поводу частоты ложноположительных результатов и пространства

Ложноположительные результаты в порционном фильтре возникают в ситуациях, когда два несовпадающих ключа генерируют один и тот же отпечаток. Анализ [12] показывает, что при наличии таблицы из 2^q слотов и отпечатка длиной $p = q + r$ вероятность ложноположительного результата сопоставима с $1/2_r$. Объем пространства, требуемый хеш-таблицей порционного фильтра, составляет $2^q(r + 3)$ бит. Число вставляемых в порционный фильтр элементов составляет $n = \alpha 2^q$, где коэффициент загруженности α оказывает существенное влияние на производительность операций вставки, поиска и удаления.

Также существует вариант порционного фильтра, в котором используется только два бита метаданных и $2^q(r + 2)$ бит для хранения, без ущерба для ложноположительной частоты, но этот вариант существенно усложняет шаг декодирования, делая обычные операции слишком процессорно-интенсивными на более длинных кластерах.

С практической точки зрения, из-за дополнительного пространства, необходимого для линейно опробываемой таблицы, порционные фильтры, как правило, занимают немного больше места, чем фильтры Блума при обычных ложноположительных частотах. При чрезвычайно низкой ложноположительной частоте порционные фильтры более экономны по занимаемому пространству, чем фильтры Блума.

Для запросов о принадлежности существуют и другие лаконичные структуры данных, основанные на хеш-таблицах, которые мы не будем изучать в этой главе (например, кукушечный фильтр [13], основанный на кукушечном хешировании). Однако будем надеяться, что идеи, которые вы почерпнете, изучая фильтры Блума и попробовав порционные фильтры, а также увидев их производительности в сравнении в следующем далее разделе, дадут вам правильный взгляд на существующие аналогичные структуры данных.

3.8 Сравнение блумовских и порционных фильтров

В этом разделе мы подытожим различия в производительности между фильтрами Блума и порционными фильтрами. Спойлер: различия в производительности не столь существенны ни в ту, ни в другую сторону. Однако больший интерес вызывает другое – поведенческие различия между двумя структурами данных, которые проистекают из того, как они были сконструированы, и которые могли бы помочь нам лучше понять природу этих структур данных. Наш анализ основан на экспериментах, ранее проведенных в оригинальной статье, посвященной порционным фильтрам; на рис. 3.13 представлено грубое резюме некоторых их результатов.

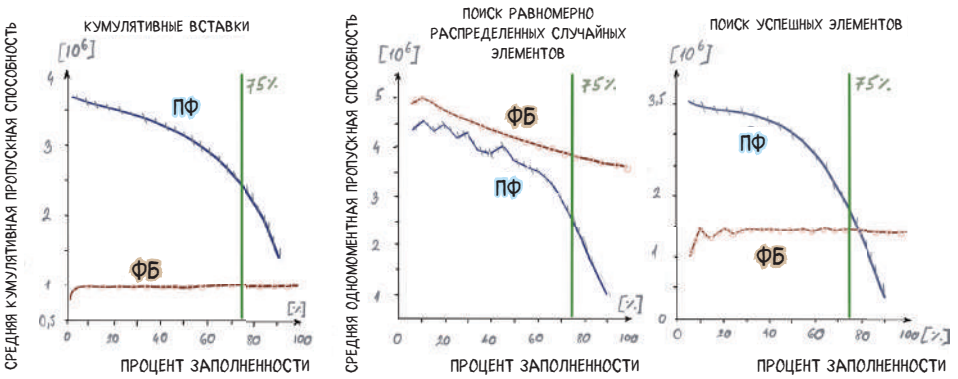


Рисунок 3.13 Сравнение производительности фильтров Блума и порционных фильтров соответственно при вставках, поиске равномерно распределенных случайных элементов и поиске успешных элементов

На всех трех графиках ось *x* представляет долю полноты структуры данных, а ось *y* – кумулятивную пропускную способность по мере заполнения структуры данных. В этом конкретном эксперименте обеим структурам данных был предоставлен одинаковый объем пространства (2 Гб), и они были заполнены как можно большим числом элементов без ухудшения ложноположительной частоты, которая была установлена равной 1/64 для обеих структур данных. Производительность в случае порционных фильтров становится очень низкой после 90%-ной занятости, поэтому порционные фильтры заполняются на 90 %.

Вставка равномерно распределенных случайных элементов

Как видно на рис. 3.13 (слева), вставки в порционных фильтрах выполняются значительно быстрее, чем в фильтрах Блума. Если порционные фильтры при 75%-ной занятости имеют кумулятивную пропускную способность ~3 млн вставок в секунду, то фильтры Блума имеют совокупную пропускную способность ~1.5 млн вставок в секунду. Фильтрам Блума требуется *k* операций случайной записи для каждой вставки, тогда как порционным фильтрам обычно требуется только 1 операция случайной записи, и это основная причина разницы в производительности. В случае более высоких ложноположительных частот фильтрам Блума может потребоваться больше хеш-функций, что может еще больше ухудшить производительность вставок. В фильтрах Блума производительность вставок остается плоской линией по мере заполнения структуры данных, тогда как в порционных фильтрах вставки замедляются по мере заполнения, поскольку им приходится декодировать все более крупные кластеры, и значительно падает после $\alpha = 0.8$.

Поиск равномерно распределенных случайных элементов

Поиск равномерно распределенных случайных элементов выполняется немного быстрее в фильтрах Блума, чем в порционных фильтрах (рис. 3.13, в центре); разница становится заметнее после того, как структуры данных достигают 70%-ной занятости и порционные фильтры начинают декодировать все более крупные кластеры. В фильтрах Блума поиск равномерно распределенных случайных элементов, как правило, выполняется быстрее, чем вставка. При достаточно большом универсальном множестве большинство операций поиска равномерно распределенных случайных элементов неуспешны, а при оптимальном заполнении фильтры Блума отклоняют поиск неуспешных элементов в среднем после одной-двух проб; побить чтение всего одного-двух бит с точки зрения производительности очень трудно.

Вспомните пример маршрутизации запросов в Google WebTable в начале главы: неуспешный поиск является обычным явлением, поэтому быстрое выполнение этой операции является весьма благоприятным для фильтров Блума. Производительность поиска равномерно распределенных случай-

ных элементов снижается лишь незначительно по мере заполнения фильтра Блума, потому что доля одного бита увеличивается, а вместе с ней и число битов, которые поиск должен проверять, прежде чем он прекратит попытки. Порционным фильтрам требуется выполнять немного больше работы, чем фильтрам Блума, при декодировании содержащего кластера, но это все равно сводится к одной операции произвольного чтения плюс дополнительной побитовой распаковке.

Поиск успешных элементов

Поиски успешных элементов демонстрируют те же тренды в производительности, что и вставки (рис. 3.13, справа), и порционные фильтры снова превосходят фильтры Блума, за исключением очень высоких коэффициентов загруженности. Фильтр Блума должен проверять k случайных бит на поиске успешных элементов (его производительность не зависит от заполненности структуры данных), тогда как производительность порционного фильтра деградирует с ростом занятости и увеличением размера кластеров.

Экспериментальные результаты не указывают на явного победителя среди фильтров Блума и порционных фильтров, но другие их особенности могут указывать на более оптимальную пригодность в конкретных условиях работы: фильтр Блума проще реализуется и создает меньшую нагрузку на центральный процессор. Порционный фильтр поддерживает операцию удаления и очень хорошо работает в распределенных средах, в которых важную роль играет эффективное слияние и изменение размера. Варианты порционного фильтра, адаптированные под твердотельный накопитель/диск, также превосходят варианты фильтра Блума, предназначенные для тех же целей, благодаря быстрому последовательному слиянию и малому числу операций произвольного чтения/записи в порционных фильтрах. Подробнее об адаптированных под твердотельный накопитель/диск версиях фильтров Блума и порционных фильтров можно узнать, обратившись к соответствующему обзору [14].

Резюме

- Фильтры Блума широко применяются в контексте распределенных баз данных, сетей, биоинформатики и других областей, где обычные хеш-таблицы занимают слишком много места.
- Фильтры Блума обменивают точность на экономию пространства, и в нем существует взаимосвязь между пространством, ложноположительной частотой, числом элементов и числом хеш-функций.
- Фильтры Блума не удовлетворяют нижней границе соотношения пространства и точности, но их проще реализовывать, чем более пространственно-эффективные альтернативы, и со временем они были адаптированы под работу с операциями удаления, различными распределениями запросов и т. д.

- Порционные фильтры основаны на компактных хеш-таблицах с использованием линейного опробывания и функционально эквивалентны фильтрам Блума, с отдельным преимуществом пространственной локализации: возможностью эффективного удаления, слияния и изменения размера.
- Порционные фильтры основаны на пространственно-экономичном методе формирования частных и остатков, а также дополнительных битах метаданных, которые позволяют полностью реконструировать цифровой отпечаток.
- Порционные фильтры обеспечивают более высокую производительность, чем фильтры Блума, в операциях вставки равномерно распределенных случайных элементов и операциях поиска успешных элементов, тогда как фильтры Блума выигрывают в операциях поиска равномерно распределенных случайных (неуспешных) элементов. Производительность порционных фильтров зависит от коэффициента загруженности (чем меньше коэффициент загруженности, тем лучше), а производительность фильтров Блума зависит от числа хеш-функций (чем меньше хеш-функций, тем лучше).

Глава 4

Оценивание частоты и набросок count-min

Эта глава охватывает следующие ниже темы:

- обследование практических примеров использования в ситуациях, когда возникают оценки частот, и помощь в этом наброска count-min;
- принцип работы наброска count-min;
- обследование вариантов использования в приложениях по обработке сенсорных данных и естественного языка;
- компромисс между ошибкой и пространством в наброске count-min;
- понятие диадических диапазонов и выполнение диапазонных запросов с помощью наброска count-min.

В современных приложениях с интенсивным использованием данных распространена задача анализа популярности, в том числе составление списка бестселлеров на сайте электронной коммерции, вычисление k верхних лидирующих запросов в поисковой системе или составление отчетов о частых парах IP-адресов источник–местоназначение в сети. Задача обнаружения аномалий (то есть мониторинг изменений в системах, которые работают в круглосуточном режиме 24/7, таких как сенсорные сети или камеры наблюдения) подпадает под тот же алгоритмический подход, что и задача измерения популярности. Обнаружение аномалий зачастую наблюдается по внезапному скачку значения определенного параметра, такого как температура или изменение местоположения в датчике, появление объекта в кадре или число пунктов, на которые выросли или упали акции компании за определенный промежуток времени.

В сущности, обе задачи сводятся к измерению частоты: для уникального ключа каждого элемента построить структуру данных, которая сможет сообщать о числе раз, когда мы с ним сталкивались. Это задача, в которой словари ключ-значение подходят как перчатка, а объем занимаемого сло-

варем пространства пропорционален не общей сумме частот встречаемости (N), а общему числу несовпадающих элементов, частоты которых мы хотели бы измерять (n). Однако это число бывает очень большим. В данной главе рассматриваются альтернативные решения задачи о частоте каждого элемента, когда число несовпадающих элементов слишком велико.

Когда мы сталкиваемся с большим числом несовпадающих элементов? Допустим, мы хотим подсчитать число разных товаров, проданных на сайте электронной коммерции. Если мы посмотрим на распределение продаж по товарам, то нередко сможем обнаружить, что малое число несовпадающих товаров составляет большинство продаж, а крупное число товаров было продано малое число раз. Такого рода распределение (так называемый закон Ципфа) наблюдалось в самых разных областях. Сложность измерения частоты при подобном типе распределения заключается в том, что, с одной стороны, существует законная необходимость решения этой задачи из-за больших вариаций в частоте, а с другой – трудность масштабирования, которая притаилась за углом из-за множества низкочастотных элементов.

Еще одна практическая ситуация, в которой могут возникать проблемы с масштабируемостью, – это когда ключи представляют собой пары элементов, например пары IP-адресов источник–местоназначение или пары слов во фрагменте текста, где n может увеличиваться квадратично по отношению к общему числу несовпадающих IP-адресов (или слов), которое само по себе может быть довольно высоким.

Кроме того, эти трудности связаны не только с ограничениями, накладываемыми на пространство. В этой главе нас будет интересовать измерение частоты в быстро движущихся потоках, которые накладывают ряд сложных ограничений на наш выбор структуры данных и алгоритма. Например, наш алгоритм сможет видеть каждый элемент только один раз (или малое постоянное число раз), прежде чем мы его отбросим и перейдем к следующему. В этих условиях жестких ограниченний решение таких задач, как запросы на получение k верхних элементов и тяжеловесов (или влиятельных игроков), зачастую невозможно. Следовательно, нам нужно прибегнуть к приближенным решениям.

Мы узнаем, как вероятностная лаконичная структура данных, набросок count-min, поможет нам измерять частоту приближенно и решать связанные с этим задачи, достигая при этом огромной экономии пространства. Мы начнем с задачи о тяжеловесах и обсудим причину, по которой для правильного решения этой задачи крайне необходимо линейное пространство, если мы хотим решить ее за один проход. Затем представим набросок count-min и его устройство, а также воспользуемся сценариями его применения в контексте обработки сенсорных данных и естественного языка. Ближе к концу главы мы также обсудим компромисс между ошибкой и пространством в наброске count-min и способы применения наброска count-min для ответа на приближенные диапазонные запросы.

4.1 Преобладающий элемент

Давайте начнем с простой задачи: при наличии массива из N элементов и условия, что указанный массив содержит элемент, который встречается по меньшей мере $\lfloor N/2 + 1 \rfloor$ раз (то есть преобладающий элемент), задание состоит в том, чтобы показать этот элемент.

Упражнение 1

Прежде чем двигаться дальше, попробуйте сконструировать и реализовать для задачи о преобладающем элементе линейно-временной алгоритм с постоянным пространством (помимо массива в качестве хранилища).

Эта задача решается с помощью алгоритма однократного прохода по массиву [1] (еще известного как алгоритм определения большинства голосов Бойера–Мора[28]), в котором используются лишь две дополнительные переменные, как показано в следующем ниже исходном коде Python:

```
def majority(A):                    ❶
    index = 0
    count = 1
    for i in range(len(A)):
        if A[i] == A[index]:
            count += 1
        else:
            count -= 1
        if count == 0:
            index = i
            count = 1
    return A[index]
```

❶ Если в A есть преобладающий элемент, то эта функция его возвращает

Функция `majority` отслеживает текущего претендента на титул преобладающего элемента и сбрасывает лидера, как только число появлений (одного или нескольких) других элементов его уравновешивает. При условии что предоставленный список содержит преобладающий элемент, приведенный выше алгоритм его покажет; в противном случае он выведет произвольный элемент. Если мы не уверены в наличии преобладающего элемента в массиве, то можно выполнить еще один виток по массиву, обеспечив, чтобы возвращенное значение действительно было преобладающим. Мы покажем работу алгоритма на двух примерах – списке с преобладающим элементом и списке с элементом, который почти преобладает:

```
A = [4, 5, 5, 4, 6, 4, 4]
print(majority(A))

C = [3, 3, 4, 4, 4, 5]
```

[28] Англ. Boyer-More majority vote algorithm. – *Прим. перев.*

```
print(majority(C))
```

Результат будет таким:

```
4
5
```

Существует также более наглядный взгляд на эту задачу: взять случайную пару смежных чисел в массиве, которые не равны друг другу, выбросить их и сжать отверстие, образовавшееся в результате удаления двух элементов. Продолжать выхватывать пары разных элементов в любом месте массива до тех пор, пока у вас не останется один несовпадающий элемент, возможно, его несколько копий; этот элемент будет преобладающим. Рисунок 4.1 иллюстрирует алгоритм, показывая коз, соперничающих за свое место на мосту, причем коза типа 2 побеждает как «преобладающая коза».

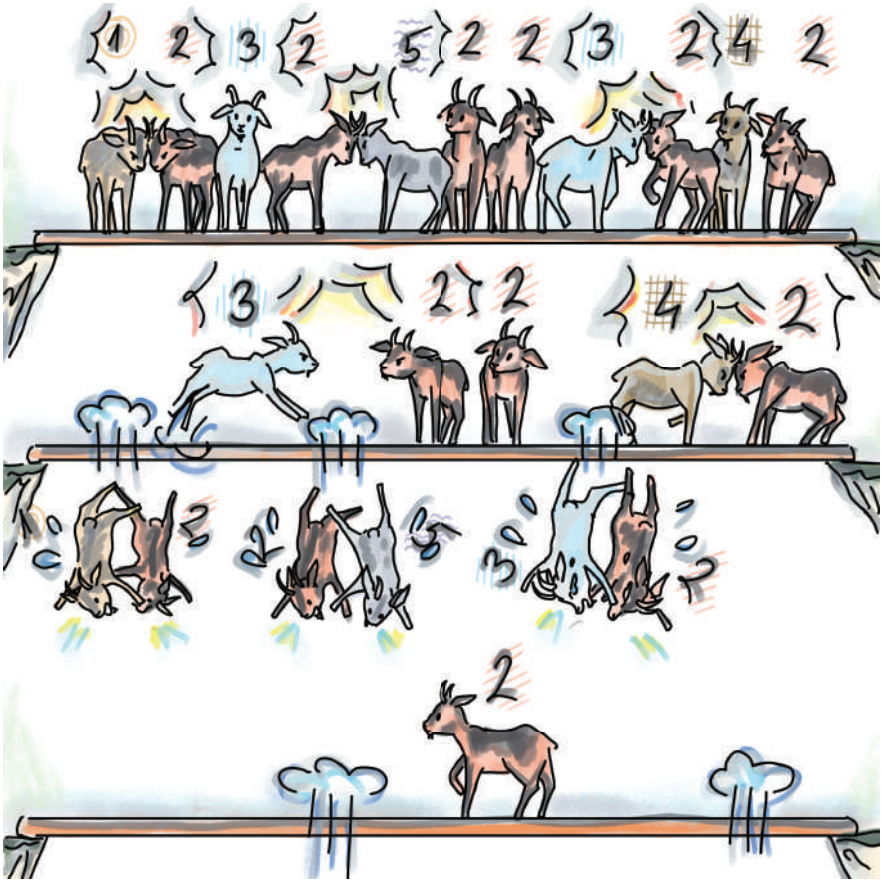


Рисунок 4.1 Мы находим преобладающий элемент в массиве, устроив так, чтобы разные соседние элементы выбрасывали друг друга. В этом примере, после того как будут выброшены (1,2), (2,5), (3,2) и затем (3,2) и (4,2), у нас останется преобладающий элемент 2

Оба алгоритма иллюстрируют, что эта задача может быть решена довольно просто за линейное время с постоянными издержками на память. Но можно ли распространить этот подход на общую задачу о тяжеловесах?

4.1.1 Общая задача о тяжеловесах

Общая задача о тяжеловесах с параметром k требует, чтобы алгоритм выводил все элементы в массиве из N элементов, которые встречаются более N/k раз (преобладающим элементом является простой экземпляр, где $k = 2$). Тяжеловесов может быть не более $k - 1$, но их также может быть меньше или вообще не быть.

Поскольку тяжеловесов может быть много, простое применение нашего алгоритма отбора претендентов подразумевает, что мы должны поддерживать большое число кандидатов на титул тяжеловесов конкурентно.

В качестве иллюстрации трудности, возникающей в такой обстановке, давайте рассмотрим следующий экстремальный случай, наблюдаемый [2] при $k = N$: допустим, мы наблюдаем длинный поток данных, в котором все обнаруженные к настоящему моменту элементы отличались бы друг от друга и однократное повторение элемента определило бы этот элемент как тяжеловеса. Для того чтобы идентифицировать потенциального тяжеловеса, нам нужно сохранять каждый новый входящий несовпадающий элемент, поскольку мы не знаем, какой из них будет повторяться.

Этот игрушечный пример имеет небольшую подковырку; его цель состоит не столько в том, чтобы проиллюстрировать практически встречающийся пример задачи, сколько в том, чтобы убедить вас в росте потребления памяти и алгоритмической сложности при более крупном k, если мы хотим найти точное решение задачи, и в необходимости обратиться к приближенному ее решению.

Что будет аппроксимироваться в задаче о тяжеловесах? Задача о (ε, k) тяжеловесах просит сообщать обо всех элементах, которые встречаются не менее $N/k - \varepsilon N$ раз, то есть обо всех тяжеловесах и обо всех элементах, которые не дотягивают не более εN, чтобы стать тяжеловесами, для некоторого ранее установленного значения ε. Другими словами, структура данных, которая будет хранить частоты, будет завышать оценку частот некоторых элементов на долю ε от общей суммы частот N.

В этой главе мы сосредоточимся на структуре данных, позволяющей регистрировать приближенные частоты, именуемой наброском count-min (набросок CM)[29,30], разработанной Кормодом (Cormode) и Мутукришнаном (Muthukrishnan) в 2005 году [3]. *Набросок count-min* похож на молодого, перспективного родственника фильтра Блума. Аналогично тому, как фильтр Блума дает приближенные ответы на запросы о принадлежности, занимая меньше места, чем хеш-таблицы, набросок count-min оценивает *частоты*

[29] Англ. count-min sketch (CMS). – *Прим. перев.*

[30] Это простой метод резюмирования частотных данных больших объемов. Расчетное число определяется наименьшим значением в таблице для i, а именно $\hat{a}_i = \min_j count[j, h_j(i)]$, где *count* – это таблица, отсюда и название count-min. – *Прим. перев.*

элементов в меньшем пространстве, чем хеш-таблица или любой линейно-пространственный словарь ключ-значение. Еще одно важное сходство заключается в том, что набросок count-min основан на хешировании, поэтому мы продолжим в том же духе, используя хеширование, чтобы создавать компактные и приближенные наброски данных. Далее мы рассмотрим принцип работы наброска count-min.

4.2 Набросок count-min: принцип работы

Набросок count-min поддерживает две главные операции: обновление как эквивалент вставки и оценивание как эквивалент поиска. Для входной пары (a_t, c_t) во временно́м слоте t операция обновления увеличивает частоту элемента a_t на величину c_t (в случае если в конкретном приложении $c_t = 1$, иными словами, числа появлений не имеют особого смысла, то обновление можно переопределить, чтобы использовать только аргумент a_t). Операция оценивания возвращает частотную оценку a_t. Возвращаемая оценка может быть завышенной по отношению к фактической частоте, но никогда не заниженной (и это не случайное сходство с ложноположительными результатами в фильтре Блума).

Набросок count-min представляется матрицей (`CMS`) целочисленных счетчиков с d строками и w столбцами (`CMS[1..d][1..w]`), при этом все счетчики первоначально установлены равными 0, и d независимыми хеш-функциями ($h_1, h_2, \ldots, h_d$). Каждая хеш-функция имеет диапазон [1..w], а j-я хеш-функция предназначена для i-й строки матрицы `CMS`, $1 \leqslant j \leqslant d$.

4.2.1 Обновление

Операция обновления добавляет еще один экземпляр (или c_t экземпляров) элемента в набор данных. Используя d хеш-функций, операция обновления вычисляет d хешей на a_t, и для каждого хеш-значения $h_j(a_t)$, $1 \leqslant j \leqslant d$, соответствующая позиция в j-й строке увеличивается на c_t:

```
CMS_UPDATE(at,ct):
    for j ← 1 to d
        CMS[j][hj(at)] += ct
```

Пример работы операции обновления показан на рис. 4.2, где мы начинаем с пустого наброска count-min и выполняем обновления элементов x, y и z соответственно величинами/частотами 2, 1 и 3.

4.2.2 Оценивание

Операция оценивания сообщает приближенную частоту запрашиваемого элемента. Так же, как и операция обновления, операция оценивания вычисляет d хешей и возвращает минимум среди d счетчиков в d разных строках, где позиция счетчика в j-й строке задается хешем $h_j(a_t)$, $1 \leqslant j \leqslant d$:

```
CMS_ESTIMATE(at):
    min = CMS[1][h1(at)]
    for j ← 2 to d
        if(CMS[j][hj(at)] < min)
            min = CMS[j][hj(at)]
    return min
```

Пример работы операции оценивания показан на рис. 4.3, где в результате оценивания элемента *y* возвращается правильный ответ, тогда как в результате оценивания элемента *x* возвращается завышенная оценка. Как видно из примера, набросок count-min может завышать фактическую частоту из-за коллизий хешей разных элементов и внесения ими своего вклада в частоты других элементов; однако завышение частоты происходит только в том случае, если коллизия была в каждой строке.

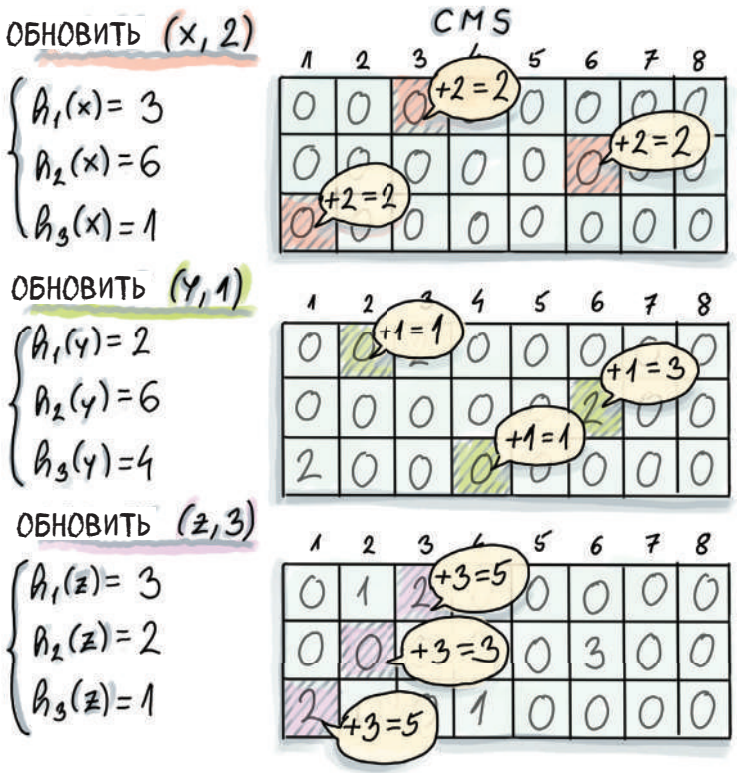


Рисунок 4.2 Три операции обновления элементов *x*, *y* и *z*, выполненные на первоначально пустом наброске count-min размера 3×8. Вычисленные хеши указывают на столбцы наброска count-min, в соответствующих строках которых необходимо делать обновления

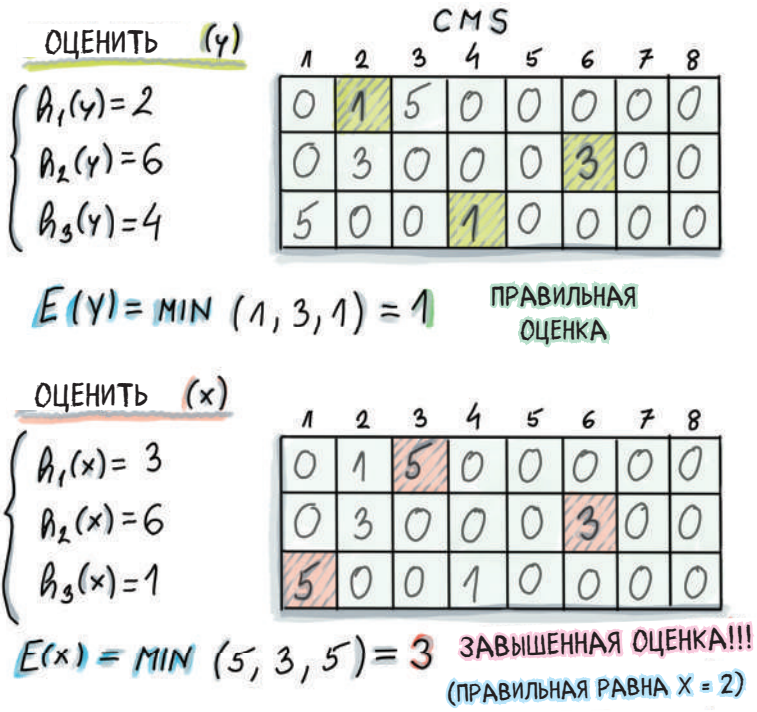


Рисунок 4.3 Пример операции оценивания для наброска count-min из рис. 4.2. В случае элемента *y*, истинная частота которого равна 1, набросок count-min сообщает правильный ответ, равный 1 (минимум из 1, 3 и 1). Однако в случае элемента *x*, истинная частота которого равна 2, набросок count-min сообщает 3 (минимум из 5, 3 и 5). Обратитесь к рис. 4.2, чтобы убедиться, что во время предыдущих операций обновления элементы *y* и *z* вместе увеличивали все счетчики, используемые элементом *x*, что привело к завышенной оценке частоты элемента *x*

4.3 Варианты использования

Теперь мы переходим к практическому применению наброска count-min в двух разных областях: приложении по обработке сенсорных данных умных кроватей и приложении по обработке естественного языка[31].

4.3.1 *k* верхних беспокойно спящих пользователей

Качество сна в течение длительного времени было связано с результатами психического и физического здоровья человека. Однако только недавно, благодаря широкому доступу к новым технологиям и возможности обрабатывать огромные массивы данных, мы смогли собирать очень подробные данные, связанные со сном, у большого числа людей. Например,

[31] Англ. natural language–processing (NLP). – *Прим. перев.*

умные кровати, оснащенные сотнями датчиков, способными регистрировать разные параметры во время сна, такие как движение, давление, температура и т. д., помогают получать новое представление о характере сна людей. Основываясь на сенсорных данных, различные компоненты кровати могут адаптироваться и видоизменяться в реальном времени – части кровати могут подниматься, подогреваться, охлаждаться и т. д.

Рассмотрим компанию, производящую умные кровати, которая собирает данные от своих пользователей и хранит их в одной центральной базе данных. Данные отправляются ежесекундно миллионами пользователей и датчиков; следовательно, объем данных быстро становится слишком большим, чтобы его можно было обрабатывать и анализировать простым способом. Давайте допустим, что одна умная кровать оснащена 100 датчиками и что этот тип умной кровати используется 10^8 потребителями; тогда наша гипотетическая компания ежедневно собирает в общей сложности 10^8 (потребителей) × 3600 (секунд в час) × 24 (часов в сутки) × 100 (датчиков) = 8.6×10^{14} кортежей данных, что приводит к ежедневному объему хранения, исчисляющемуся терабайтами. Этот конкретный пример является гипотетическим, но объем собранных данных и связанная с ними задача, которую мы изучим, таковыми не являются.

С каждой покупкой умной кровати поставляется мобильное приложение SleepQuality, которое позволяет пользователям кроватей отслеживать свой сон во временной динамике. Одно из новых приложений отслеживает беспокойство спящих и уведомляет самых беспокойных спящих пользователей о том, что характер их сна отличается от характера сна остальных спящих. Реализуя эту функциональность, приложение учитывает показания разных датчиков и сводит свои результаты в один балл по шкале качества сна. Из-за огромного объема поступающих данных инженеры компании решили попробовать хранить получаемые от датчиков пользователей величины в наброске count-min.

Как показано на рис. 4.4, данные поступают с высокой частотой, и на каждом временном шаге пара (`user-id`, `amount`)[32] обновляет набросок count-min на указанную величину. В нашем упрощенном примере ключи наброска count-min соответствуют индивидуальным пользователям; в целях более тонкого анализа изменения показаний конкретных датчиков ключом должна быть пара (`user-id`, `sensor-id`)[33].

Обновляя набросок count-min таким образом, можно генерировать приближенную оценку частоты любого запрошенного пользователя. Но для того, чтобы поддерживать список k верхних беспокойных спящих пользователей, нам придется сделать немного больше, чем просто поддерживать набросок count-min. Вспомните, что набросок count-min – это всего лишь матрица счетчиков, и в ней не хранится никакой информации о разных идентификаторах пользователей или упорядочении соответствующих частот.

[32] То есть ИД пользователя, величина. – *Прим. перев.*

[33] То есть ИД пользователя, ИД датчика. – *Прим. перев.*

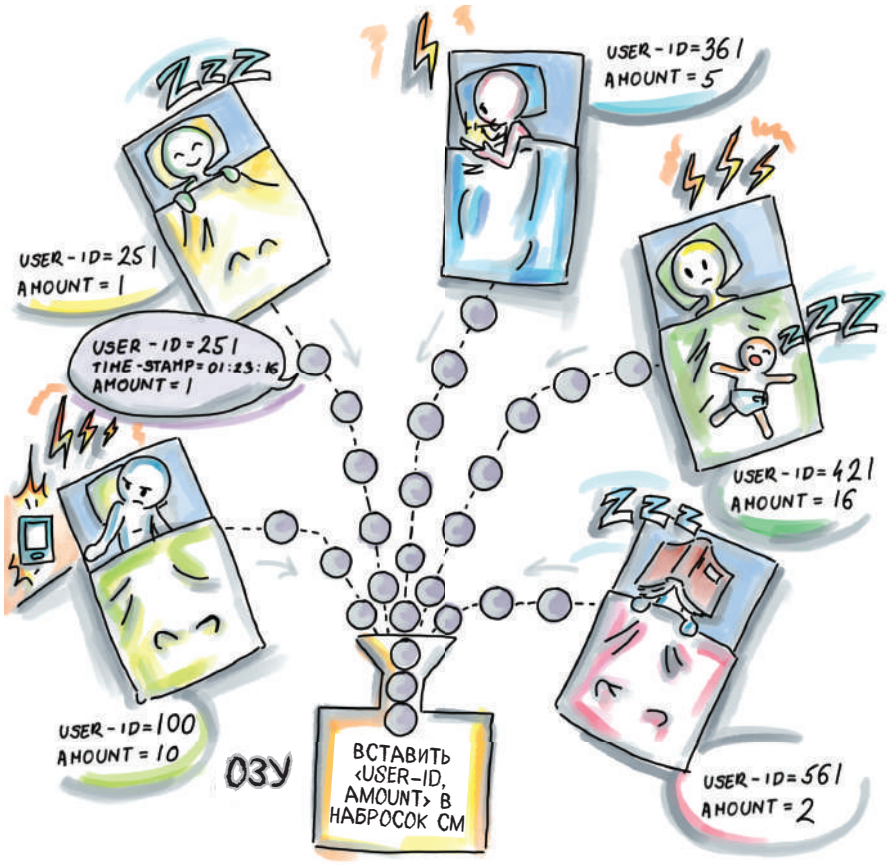


Рисунок 4.4 Все данные о состоянии сна отправляются в центральный архив, но перед этим они вводятся в резидентный набросок count-min для последующего анализа. Пара (`user-id`, `amount`) подается на вход структуры данных, в которой частота идентификатора `user-id` увеличивается на величину `amount`. Хешируемым ключом является `user-id`

Упражнение 2

Прежде чем перейти к решению, подумайте о том, какой могла бы быть правильная структура данных, которую можно было бы использовать вместе с наброском count-min для хранения k верхних беспокойных спящих пользователей в каждый момент времени, используя лишь $O(k)$ дополнительного пространства.

В одном решении применяется *минимум-ориентированная куча*[34], как показано на рис. 4.5. Минимум-ориентированная куча поддерживает текущих k «победителей» конкурса на беспокойность с возможностью обновления всякий раз, когда поступает новое обновление наброска count-min.

[34] Англ. min-heap; минимум-ориентированная куча – это структура данных на основе двоичного дерева, в которой значение каждого узла меньше или равно его дочерним узлам. – *Прим. перев.*

В частности, набросок count-min обновляется по прибытии нового элемента, после чего мы выполняем операцию оценивания этого конкретного элемента. Если сообщаемая частота превышает минимальный элемент в минимум-ориентированной куче (легкодоступный за O(1)), то минимум кучи удаляется и вставляется новый элемент. Также обратите внимание, что при каждом обновлении уже находящегося в куче элемента обновленная частота должна отражаться в позиции элемента кучи.

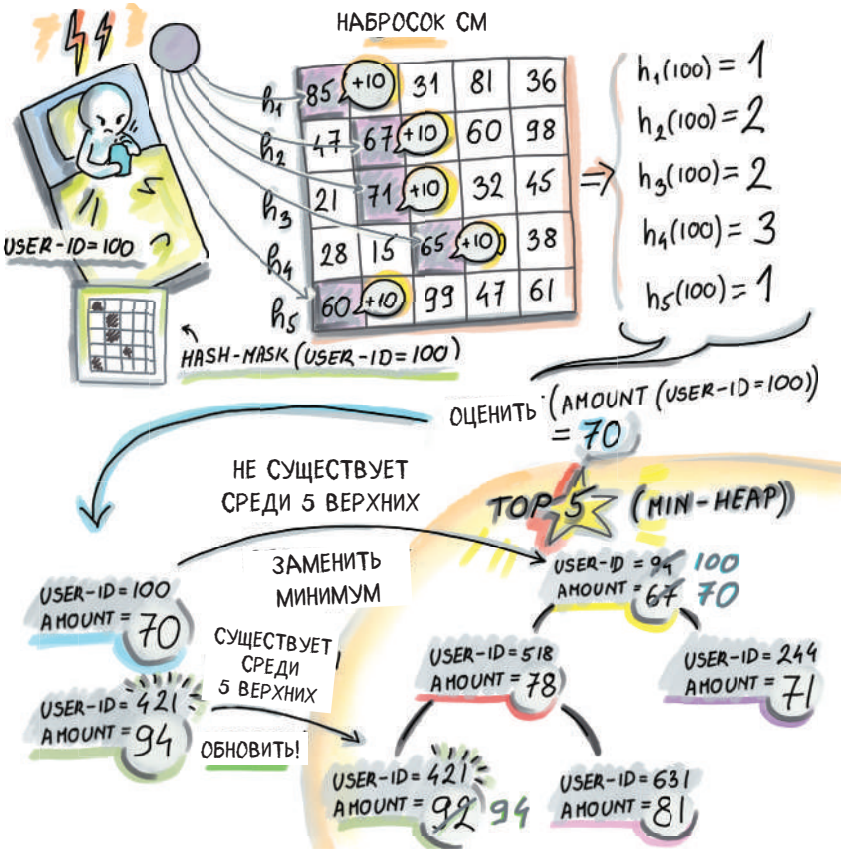


Рисунок 4.5 Совместное использование минимум-ориентированной кучи и наброска count-min для отыскания *k* верхних беспокойных спящих пользователей. Для того чтобы поддерживать правильный список *k* верхних беспокойно спящих пользователей при каждом обновлении наброска count-min парой `(user-id, amount)`, в данном примере `(100, 10)` мы оцениваем частоту недавно обновленного идентификатора `user-id`. В нашем случае частотная оценка идентификатора `user-id` 100 будет равна 70. Затем если идентификатор `user-id` в куче отсутствует и имеет значение выше минимума (как это происходит в данном примере), то мы извлечем минимальное значение и вставим новую пару `(user-id, amount)` в кучу. Если пара уже присутствовала, то ее значение необходимо обновить, удалив и вставив пару повторно с новым, обновленным (более высоким) значением

В этом примере мы показали, что набросок count-min можно использовать вместо обычного словаря ключ-значение, чтобы хранить информацию о частотах для приложения SleepQuality, тем самым экономя много пространства (хотя мы еще не анализировали потребности в пространстве). В то же время мы использовали минимум-ориентированную кучу размера O(*k*) для хранения информации о *k* верхних беспокойно спящих пользователях. Минимум-ориентированная куча поддерживает актуальные оценки частот на каждый момент времени, поэтому всякий раз, когда мы хотим отправить уведомление таким пользователям, в нашем распоряжении есть вся необходимая информация.

4.3.2 Масштабирование распределительного сходства между словами

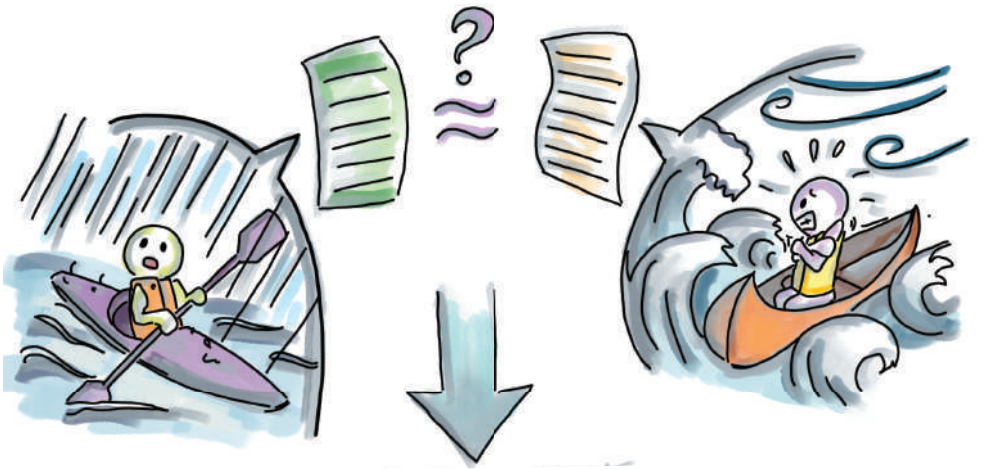
Задача семантического сходства между словами на основании их распределения, или *распределительного сходства*, требует, чтобы при наличии большого текстового корпуса мы находили пары слов, схожих по смыслу в зависимости от контекстов, в которых они встречаются (или, как выразился лингвист Джон Р. Ферт (John R. Firth), «Вы узнаете слово по компании его друзей»). Например, слова *каяк* и *каноэ* будут окружены похожими словами, такими как *вода*, *спорт*, *погода*, *река* и т. д. В качестве контекста данного слова мы выбираем окно размера *k* (например, *k* = 3), которое включает в себя *k* слов до и *k* слов после данного слова в тексте, или меньше, если мы пересекаем границу предложения.

Для измерения распределительного сходства заданной пары слово–контекст широко используется метод *поточечной взаимной информации*[35] [4]. Формула поточечной взаимной информации для слов *A* и *B* выглядит следующим образом:

$$\textit{Поточечная взаимная информация}(A, B) = \log_2 \frac{Prob(A \cap B)}{Prob(A)Prob(B)},$$

где *Prob*(*A*) обозначает вероятность появления *A*, то есть число появлений *A* в корпусе, деленное на общее число слов в корпусе. Это причудливый способ сказать, что поточечная взаимная информация служит мерой вероятности совместного появления *A* и *B* в корпусе по сравнению с частотой их появления, если бы они были независимыми. Чем выше поточечная взаимная информация, тем больше слова похожи. Как правило, перед вычислением этой метрики для всех интересующих пар слово–контекст или конкретных пар слово–контекст корпус предобрабатывается, чтобы произвести матрицу наподобие той, которая показана на рис. 4.6, содержащую все частоты пар слово–контекст.

[35] Англ. pointwise mutual information (PMI). – *Прим. перев.*



КОНТЕКСТ / СЛОВА	ВОДА	ЛЕД	МЕБЕЛЬ	СТАРО-МОДНОСТЬ	ДУБ	ПОГОДА	Σ СЛОВА
КАЯК	3	0	0	0	0	1	4
КАНОЭ	2	1	0	0	0	1	4
СТУЛ	0	0	2	1	0	0	3
СТОЛ	0	0	1	2	1	0	4
Σ КОНТЕКСТ	5	1	3	3	1	2	ВСЕГО 15

Рисунок 4.6 Создание матрицы *M*, в которой запись `M[A][B]` содержит число появлений слова *A* в контексте *B*, является одним из способов предобработки текстового массива для вычисления поточечной взаимной информации. Например, *каяк* появляется три раза в контексте *воды* и ноль раз в контексте *мебели*. Кроме того, мы дополнительно подсчитываем каждое слово (последний столбец матрицы) и каждый контекст (последняя строка матрицы), а также общее число слов (правый нижний угол)

Более высокие баллы ассоциативности между словами зависят от объема текста – чем больше текста, тем лучше, – но при увеличении корпуса, даже если число несовпадающих слов является достаточно разумным по размеру, число пар слово–контекст быстро выходит из-под контроля.

Например, в исследовательской работе, в которой измерялось распределительное сходство с использованием методов формирования набросков [5], применялся набор данных Gigaword, полученный из англоязычных

текстовых новостных источников, содержащий 9.8 Гб текста и около 56 млн предложений. В результате получилось 3.35 млрд токенов пар слово–контекст и 215 млн уникальных пар слово–контекст; простое хранение этих пар с их частотами занимает 4.6 Гб. Решение состоит в преобразовании матрицы таким образом, чтобы частоты пар слово–контекст хранились в наброске count-min, и поскольку число несовпадающих слов не слишком велико, можно позволить себе хранить слова с их частотами в их собственной хеш-таблице (см. последний столбец матрицы), а контексты с их частотами – в их собственной хеш-таблице (см. последнюю строку матрицы). Это преобразование показано на рис. 4.7.

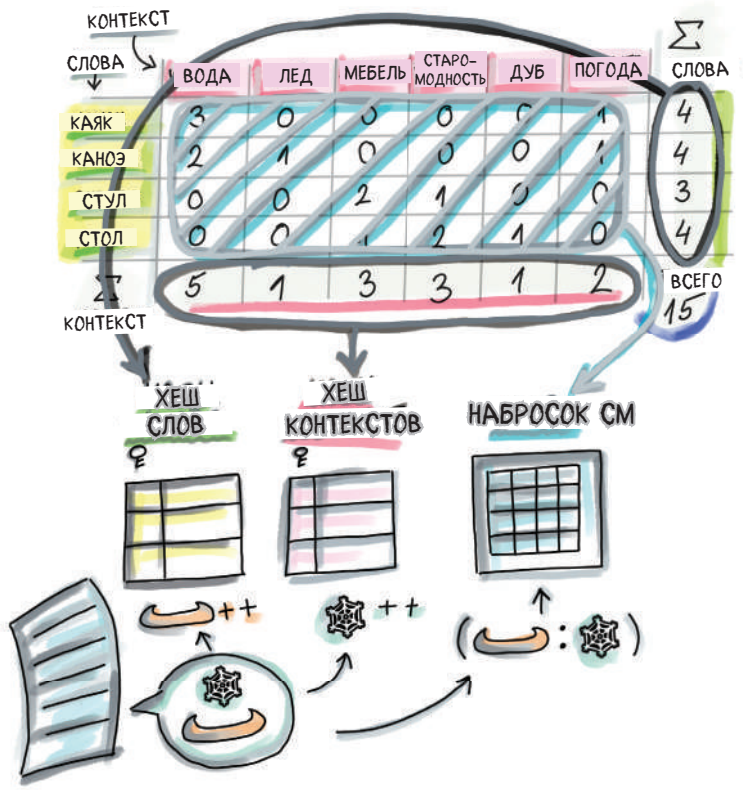


Рисунок 4.7 Преобразование матрицы из рис. 4.6 для экономии пространства: пары слово–контекст, хранящиеся в главном теле матрицы, заменяются наброском count-min, в котором хранятся частоты пар слово–контекст. Поскольку число несовпадающих слов (и контекстов) не так велико, каждое из них можно хранить в их собственной хеш-таблице с соответствующими частотами. Другими словами, когда мы сталкиваемся с новой парой (слово, контекст), мы увеличиваем частоту пары в наброске count-min и увеличиваем соответствующие частоты в хеш-таблице слов и хеш-таблице контекстов. При расчете поточечной взаимной информации для пары слово–контекст мы выполняем запрос к наброску count-min и находим соответствующие частоты слова и контекста в соответствующих хеш-таблицах

Экономия пространства, достигнутая в этом примере использования наброска count-min, составила более 100 раз. Авторы данного исследования сообщают, что набросок размера 40 Мб дает результаты, сопоставимые с другими методами вычисления распределительного сходства, которые используют гораздо больше пространства. Генерирование этого наброска count-min и двух хеш-таблиц занимает всего один проход по предобработанным и очищенным данным, что является большим благом для обработки потоковых данных. Метрики поточечной взаимной информации *k* верхних можно было бы произвести с помощью дополнительного витка по данным.

Возможно, вы задаетесь вопросом о том, как конфигурировать набросок count-min (то есть как задавать его размеры) и какова взаимосвязь между завышением частоты и размером наброска count-min. Набросок count-min имеет два параметра ошибки, ε и δ, и их значения используются для определения размеров наброска. В следующем далее разделе мы рассмотрим взаимосвязь между ошибкой и потребностями в пространстве подробнее.

4.4 Ошибка и пространство в наброске count-min

Набросок count-min демонстрирует два типа ошибок: ε (эпсилон), которая регулирует диапазон завышения частоты, и δ (дельту), вероятность неуспеха[36]. Если для потока *S,* достигшего временного слота *t*,

$$S = (a_1, c_1), (a_2, c_2), \ldots, (a_k, c_k)$$

через N обозначить общую сумму наблюдаемых в потоке частот, $N = \sum_i^t = 1C_t$, то ошибку ε завышения частоты можно выразить как процент от N, на который можно превышать фактическую частоту любого элемента. Другими словами, для элемента x и его истинной частоты f_x набросок count-min оценивает частоту как f_{est}

$$f_x \leqslant f_{est} \leqslant f_x + \varepsilon N$$

с вероятностью не менее 1 – δ. Обычно значение δ получает малое значение (например, 0.01), вследствие чего можно рассчитывать, что ошибка завышения частоты будет с высокой вероятностью оставаться в обещанной полосе. Существует малая вероятность, δ, что завышение частоты в наброске count-min будет неограниченным.

Так же, как и в случае с фильтром Блума, набросок count-min можно отрегулировать на более высокую точность, но это будет стоить пространства. Какими бы ни были значения (ε, δ), которые мы хотим иметь в приложении, для достижения сформулированных границ необходимо сконфигурировать размеры наброска count-min, установив их равными $w = e/\varepsilon$ и $d = \ln(1/\delta)$.

[36] Англ. failure probability; определяется как вероятность превышения предельного состояния. – *Прим. перев.*

Следовательно, требуемое для наброска count-min пространство, выраженное в числе целочисленных счетчиков, будет равно (уравнение 4.1)

$$O\left(\frac{e\ \ln\left(\frac{1}{\sigma}\right)}{\varepsilon}\right). \qquad \text{(Уравнение 4.1)}$$

Обратите внимание, что набросок count-min, как правило, невелик, даже при использовании с большими наборами данных. Набросок count-min часто хвалят за его потребности в пространстве – они не зависят от размера набора данных, – но это верно только в том случае, если вы хотите, чтобы ошибка составляла фиксированный процент от размера набора данных. Например, поддержание допустимой полосы ошибки фиксировано на уровне 0.3 % от N и не потребует увеличения размера наброска count-min, даже если удвоить значение N, но фактическая абсолютная полоса завышения частоты удвоится. Можно было бы возразить, что при вдвое большем N приложение должно быть в состоянии допускать ошибку завышения частоты, которая в два раза больше.

Однако в части свойств ошибки наброска count-min оставляет желать лучшего то, что ошибка завышения частоты чувствительна только к общей сумме частот N, а не к частоте отдельного элемента. Следовательно, полоса ошибки может сильно варьироваться, если наблюдать ее по отношению к индивидуальной частоте элемента: если максимальная завышенная оценка частоты равна $\varepsilon N = 200$, то мы в равной степени можем ожидать, что она будет завышенной оценкой для элемента с частотой 10 000 и для элемента, частота которого равна 10. В последнем случае оценка частоты может проскакивать мимо истинной частоты в 20 раз.

4.5 Простая реализация наброска count-min

Теперь мы готовы посмотреть на минимальную реализацию наброска count-min. Как и в случае с фильтрами Блума, мы используем обертку хеш-функции MurmurHash, `mmh3`, для d хеш-функций:

```
import numpy as np
import mmh3
from math import log, e, ceil

class CountMinSketch:
    def __init__(self, eps, delta):
        self.eps = eps
        self.delta = delta
        self.w = int(ceil(e/eps))                    ❶
        self.d = int(ceil(log(1. / delta)))          ❷
        self.sketch = np.zeros((self.d, self.w))
```

```
    def update(self, item, freq=1):
        for i in range(self.d):
            index = mmh3.hash(item, i) % self.w
            self.sketch[i][index] += freq

    def estimate(self, item):
        return min(self.sketch[i][mmh3.hash(item, i) % self.w] for i
        ➥ in range(self.d))
```

❶ Устанавливает ширину
❷ Устанавливает глубину

Попробуйте приведенный ниже исходный код, который показывает использование класса `CountMinSketch`. Поэкспериментируйте с обновлениями и посмотрите на изменение оценок частот:

```
cms = CountMinSketch(0.0001, 0.01)
for i in range(100000):
    cms.update(f'{i}', 1)
print(cms.estimate('0'))
```

В разделе 4.5.1 мы приводим несколько упражнений, чтобы проверить ваше понимание процедуры конфигурирования наброска count-min. Раздел 4.5.2 посвящен интуитивному пониманию, вытекающему из формального выведения частот ошибки в наброске count-min, и носит более теоретический характер. И поэтому указанный раздел в первую очередь предназначен для читателей, интересующихся математическими основами обсуждаемой структуры данных, в противном случае его можно просто пролистать.

4.5.1 Упражнения

Следующие ниже упражнения предназначены для проверки вашего понимания наброска count-min, его устройства и влияния его формы и размера на частоту ошибки.

Упражнение 3

Имея $N = 10^8$, $\varepsilon = 10^{-6}$ и $\delta = 0.1$, определите свойства наброска count-min, касающиеся ошибки.

Упражнение 4

Рассчитайте потребности в пространстве для наброска count-min из упражнения 3.

Упражнение 5

Рассмотрите, что происходит с размером (и, конкретнее, формой) наброска count-min, если мы хотим получить фиксированную абсолютную

ошибку (εN) при увеличении N. Например, предположим, что мы хотим поддерживать завышенную оценку частоты на уровне 100 или меньше, как в упражнении 3, но для N, которое в два раза больше.

Упражнение 6

Сможете ли вы сконструировать два наброска count-min, которые потребляют одинаковый объем пространства, но имеют очень отличающиеся характеристики производительности (по отношению к их ошибкам)? Каково практическое ограничение, лимитирующее глубину наброска count-min низкими значениями (< 30)?

Упражнение 7

Как бы вы решили упомянутую в начале главы задачу о приближенных k тяжеловесах с помощью наброска count-min? В частности, как бы вы установили ε, чтобы облегчить решение этой задачи? Напомним, что в задаче о приближенных k тяжеловесах мы хотели бы сообщать обо всех тяжеловесах и, возможно, о тех, кто не является тяжеловесом.

4.5.2 Вытекающий из формулы интуитивный вывод: немного математики

Как мы наблюдали в простой реализации наброска count-min, для достижения завышенной частоты не более εN с вероятностью не менее $1 - \delta$ ширина наброска count-min w должна быть установлена равной e/ε, а глубина наброска count-min d должна быть установлена равной $\ln(1/\delta)$. Почему w связано с ε, а d связано с δ?

В целях понимания причины давайте рассмотрим процесс выполнения обновлений в наброске count-min и сосредоточим особое внимание на первой строке. К тому времени, когда мы выполним все обновления наброска count-min, сумма счетчиков в первой строке (и любой отдельной строке) будет равна N. Исходя из допущения, что хеш-функции распределяют обновления равномерно случайно по ячейкам, случайная величина X, описывающая значение, хранящееся в любой фиксированной ячейке C в первой строке, после того как были выполнены все обновления, имеет среднее (или ожидаемое) значение $E[X] = N/w$.

Это также означает, что при оценивании конкретного элемента, счетчик которого в первой строке находится в ячейке C, другие элементы могут вносить вклад в его счетчик в среднем не более чем на N/w. Для того чтобы среднее завышение в одной строке не превышало εN, можно установить $w = 1/\varepsilon$. Очевидно, что величина завышенной оценки частоты связана с шириной структуры данных.

$E[X]$ говорит о поведении в среднем, но X может значительно отклоняться от своего математического ожидания: в некоторых ячейках значения могут быть намного выше, чем в других. Завышение частоты в одной стро-

ке можно в каком-то смысле умеренно ограничивать, используя неравенство Маркова, которое говорит, что если X – неотрицательная случайная величина, а $c > 1$, то

$$Pr\left(X \geq c \times E[X]\right) \leq \frac{1}{c}.$$

Применив неравенство Маркова к нашему случаю, мы получаем следующее:

$$Pr\left(X \geq \frac{eN}{w}\right) \leq \frac{1}{e}.$$

Другими словами, вероятность того, что конкретная ячейка в первой строке будет иметь значение $\varepsilon N/w$ или больше, не превышает $1/\varepsilon$. Но этого недостаточно: для того чтобы связать вероятность завышенных оценок частот выше, чем εN, мы рассматриваем все d строк. Вспомните, что для того чтобы сообщать о завышенной частотной оценке элемента q в наброске count-min, соответствующие ячейки в *каждой* строке должны иметь завышенную оценку, как минимум равную q. Если применить вытекающую из неравенства Маркова вероятность ко всем уровням (обратите внимание, что результаты хеш-функций для разных уровней взаимно независимы), то мы обнаружим, что

$$Pr\left(\textit{завышенная оценка в каждой строке равна как минимум } \frac{eN}{w}\right) \leq \left(\frac{1}{e}\right)^d.$$

Установив $w = e/\varepsilon$ и $d = \ln(1/\delta)$, мы обнаружим, что вероятность того, что завышенная оценка частоты больше εN, составляет не более δ.

4.6 Диапазонные запросы с помощью наброска count-min

В качестве заключительного примера применения наброска count-min в этой главе мы обсудим вопрос о том, как сообщать оценки частот не для отдельных точек, а для диапазонов. Отчетность о диапазонах имеет огромное значение в базах данных, где запросы часто задаются для выявления свойств групп и категорий, а не отдельных точек данных; естественно, такие запросы, как «показать всех сотрудников, проработавших в компании от a до b лет или имеющих зарплату от x до y», естественным образом транслируются в диапазонные запросы. Еще одним примером диапазонов являются временные ряды; например, «сколько книг было продано на Amazon.com между 20 декабря и 10 января этого года?».

Для навигации по диапазонам очень хорошо подходят структуры данных в виде сбалансированных деревьев двоичного поиска, поскольку их элементы расположены в лексикографическом порядке, и поэтому стоимость

диапазонного запроса после отыскания начальной точки пропорциональна простой стоимости вывода результатов запроса внутри диапазона; это отличается от хеш-таблиц, которые рассеивают данные по всей таблице и где для диапазонного запроса может потребоваться полное сканирование таблицы, даже если в отчете будет сообщено ноль элементов. Как нетрудно себе представить, это не рисует многообещающей картины для обследования диапазонов с использованием наброска на основе хеша.

Простое использование наброска count-min для получения оценок частот на диапазонах состоит в преобразовании диапазонного запроса для диапазона $[x, y]$ в $y - x + 1$ точечных запросов для каждой точки на интервале запроса и не дает желаемых результатов. В дополнение ко времени запроса, увеличивающегося линейно вместе с размером диапазона, вместе с ним линейно увеличивается и ошибка, поэтому вместо того, чтобы обещать завышение частоты не более εN с вероятностью не менее $1 - \delta$, можно обещать не более $(y - x + 1)\ \varepsilon N$, тогда для крупных диапазонов можно будет расценивать структуру данных как абсолютно бесполезную. Например, если построить набросок count-min с максимальным завышением частоты, $\varepsilon N = 7$, то диапазонный запрос с размером интервала 10 000 может привести к завышению оценки частоты до 70 000.

4.6.1 Диадические интервалы

Во избежание линейно растущей ошибки необходимо найти способ разложения произвольного диапазона на малое число поддиапазонов. Благодаря этому мы сможем получить более устойчивые оценки частот, суммируя оценки частот меньших диапазонов без накопления существенных ошибок [6].

Главная идея состоит в подразделении диапазона на малое число так называемых *диадических диапазонов*. Имея полный универсальный интервал $U = [1, n]$, мы определяем множество диадических диапазонов на $\log_2 n + 1$ разных уровнях: диадические диапазоны уровня i, $0 \leqslant i \leqslant \log_2 n$ имеют длину 2^i и могут быть выражены как $[j2^i + 1, (j + 1)2^i]$, где $0 \leqslant j \leqslant n/2^i - 1$ (см. рис. 4.8, демонстрирующий множество диадических диапазонов для универсального интервала $U = [1,16]$).

Интересным свойством диадических диапазонов является то, что любой произвольный диапазон может быть разложен не более чем на $2\log U$ диадических диапазонов. Позже в этом разделе мы покажем исходный код Python, который может раскладывать произвольный диапазон на множество диадических диапазонов, но сначала, в качестве примера, рассмотрим на рис. 4.8 малое универсальное множество и взглянем на несколько примеров подразделения диапазонов на наименьшее множество диадических диапазонов:

- диапазон [5,14] может быть подразделен на три диадических диапазона: [5,8], [9,12], [13,14];

- диапазон [2,16] может быть подразделен на четыре диадических диапазона: [2,2], [3,4], [5,8], [9,16];
- диапазон [9,13] может быть подразделен на два диадических диапазона: [9,13], [13,13].

Возможность сообщать частоту диапазона как сумму частот диадических диапазонов предусматривает поддержание информации о частоте каждого диадического диапазона по мере обновления. С этой целью можно использовать один набросок count-min, который будет обслуживать все обновления диадических диапазонов одного уровня (диадические диапазоны одинакового размера), в общей сложности O(log *n*) набросков count-min. Далее мы опишем эту схему подробнее.

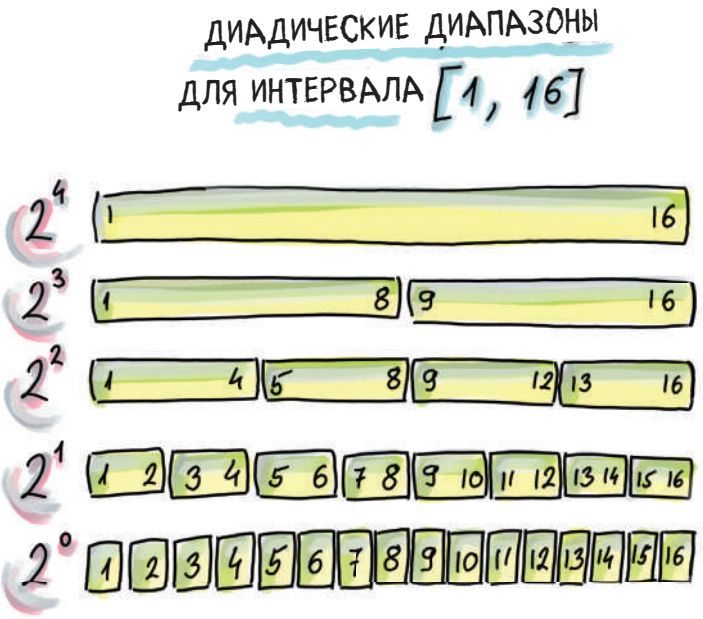


Рисунок 4.8 Диадические диапазоны для интервала = [1,16]. Диадические диапазоны уровня 0 находятся ниже всех, с диапазонами размера 1; затем уровнем выше находятся диапазоны уровня 1, с диапазонами размера 2; а общие диадические диапазоны на уровне *i* имеют размер 2^i. Диадические диапазоны на разных уровнях взаимно выровнены

4.6.2 Фаза обновления

Учитывая, что теперь элементными единицами являются диадические диапазоны, нам нужно конвертировать обновление одного поступающего в нашу систему элемента в обновление каждого диадического диапазона, в котором этот элемент содержится. Например, при обновлении частоты

элемента 5 в примере универсального множества [1,16] мы будем обновлять частоту следующих диадических диапазонов: [5], [5,6], [5,8], [1,8] и [1,16]. Диапазоны могут хешироваться точно так же, как и обычные элементы, поэтому нет никаких препятствий, для того чтобы диадический диапазон формата [*l*, *r*] рассматривался как один элемент.

Это достигается с помощью O(log *n*) набросков count-min за счет сборки одного наброска count-min для каждого уровня диадического диапазона; элементы, подлежащие обновлению/оцениванию в наброске count-min на уровне *i*, будут диадическими диапазонами этого уровня. На рис. 4.9 показан процесс обновления нового элемента: прибывающий новый элемент будет обновляться в каждом наброске count-min путем обновления содержащего его диапазона в соответствующем наброске count-min (CMS).

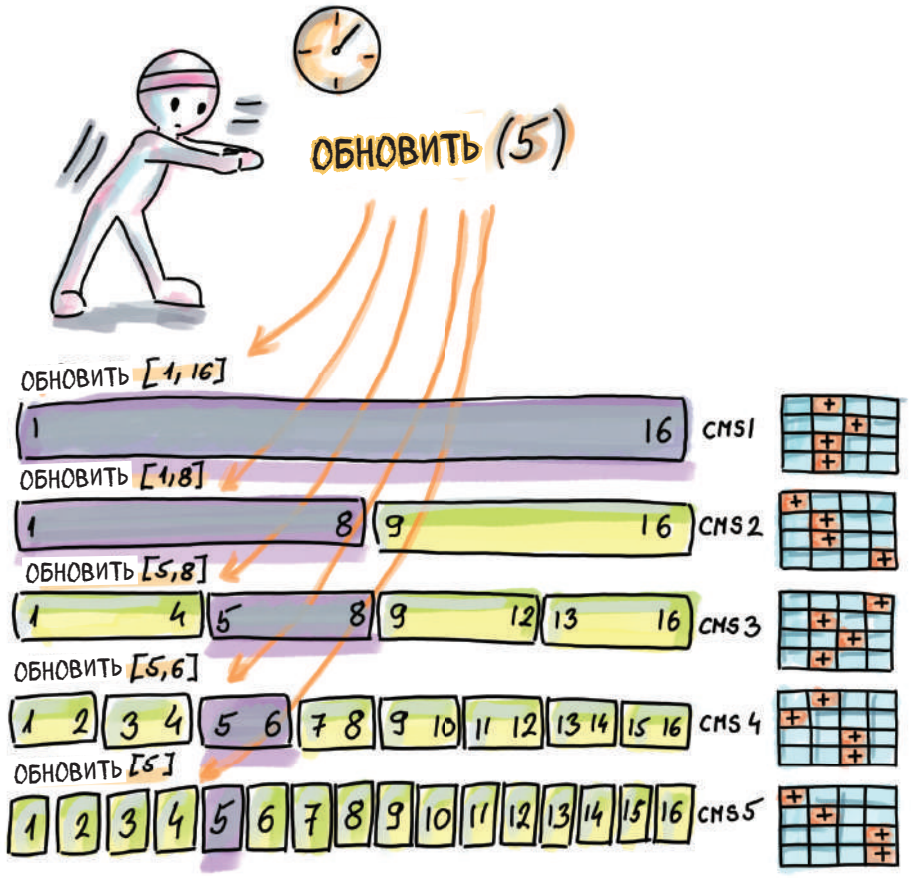


Рисунок 4.9 Обновление одного элемента преобразовывается в одно обновление в расчете на каждый уровень. Например, если мы обновляем 5, то фактически обновляем [1,16] в CMS1, [1,8] в CMS2, [5,8] в CMS3, [5,6] в CMS4 и [5] в CMS5. Вместо обновления элемента мы обновляем соответствующий диапазон, к которому элемент принадлежит в соответствующем наброске count-min

4.6.3 Фаза оценивания

Теперь мы готовы выполнить оценку определенного диапазона, используя диадические диапазоны. Сначала мы подразделяем диапазон запроса на его собственное множество диадических диапазонов. Для каждого диадического диапазона мы выполняем оценку в наброске count-min, который находится на его уровне (каждый запрос может иметь не более двух диадических диапазонов на одном уровне). Окончательный результат получается в результате суммирования всех оценок. На рис. 4.10 показана процедура оценивания диапазона для [3,13], частотную оценку которого мы получаем, оценивая следующие диадические диапазоны в соответствующих набросках count-min и их суммируя: [3,4], [5,8], [9,12] и [13].

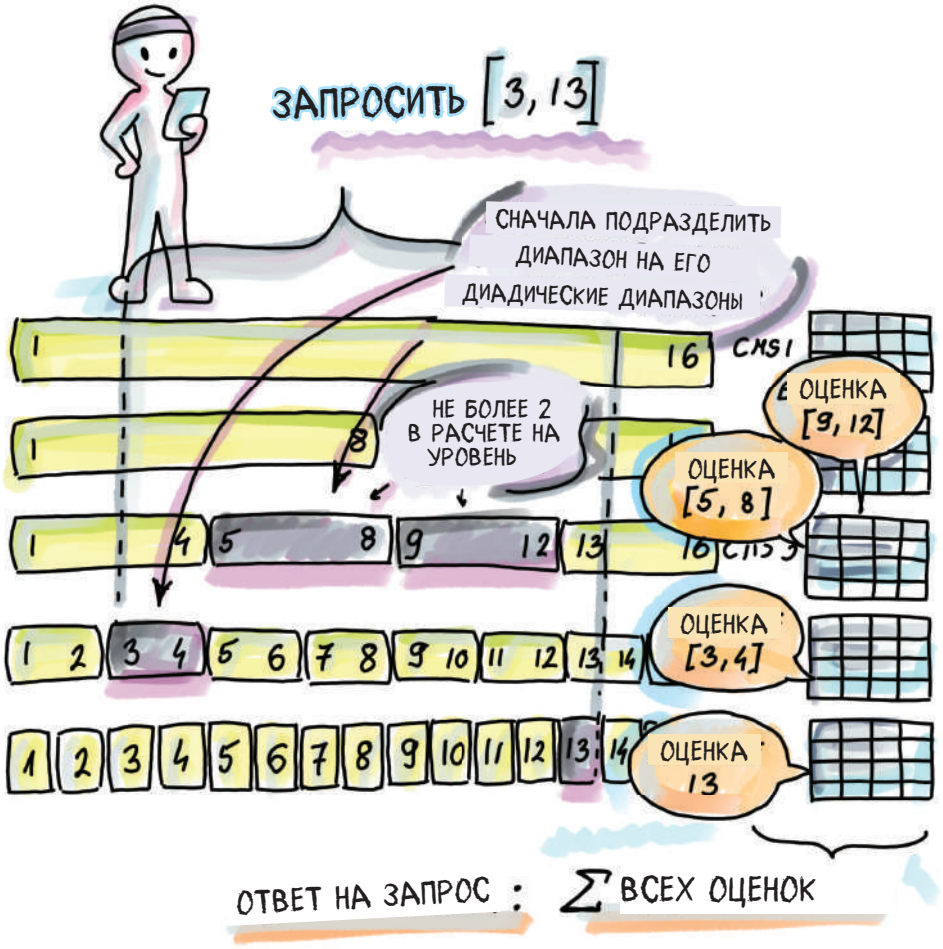


Рисунок 4.10 В этом примере диапазон [3,13] запроса подразделен на [3,4] ∪ [5,8] ∪ [9,12] ∪ [13], и мы получим частотную оценку для [3,13], получив частотные оценки для упомянутых диапазонов и их просуммировав

Полезно знать, что каждый диапазон может быть подразделен не более чем на 2log n диадических диапазонов (не более двух на уровень). Время выполнения обновления и оценивания является логарифмическим, и ошибка растет только логарифмически. Ошибку можно сделать такой же, как в изначальном наброске count-min, сделав отдельные наброски count-min в этой схеме шире на логарифмический коэффициент, чтобы логарифмы взаимоуравновешивались.

4.6.4 Вычисление диадических интервалов

Приведенный ниже исходный код Python дает разложение интервала I на диадические интервалы, где имеется большое универсальное множество U и диапазон $1 \subseteq U$. Сначала мы строим полное дерево двоичного поиска на основе универсального интервала, аналогично рис. 4.8, где каждый уровень соответствует уровню диадических диапазонов и каждый узел соответствует уникальному диадическому диапазону. Например, корневой узел представляет диапазон [1, n], его левый дочерний узел представляет диапазон [1, n/2], его правый дочерний узел представляет диапазон [n/2 + 1, n] и т. д. Листья представляют диапазоны размера 1, и всего их n. Мы строим такое дерево из универсального интервала:

```
from collections import deque

class Node:                                          ❶
    def __init__(self, lower, upper):
        self.data = (lower, upper)
        self.left = None
        self.right = None
        self.marked = False

def intervalToBST(left, right):                      ❷
    if left == right:
        root = Node(left, right)
        return root
    if abs(right - left) >= 1:
        root = Node(left, right)
        mid = int((left + right) / 2)
        root.left = intervalToBST(left, mid)
        root.right = intervalToBST(mid + 1, right)
        return root
```

❶ Каждый узел представляет диадический диапазон
❷ Преобразовывает интервал [left, right] в дерево двоичного поиска

Имея тот или иной диапазон, теперь мы вычисляем множество его диадических диапазонов, используя построенное нами дерево двоичного поиска. В каждом узле также используется атрибут `marked`. Узлы, которые в

конечном итоге будут иметь значение атрибута `marked`, равное `True`, будут узлами, представляющими диадические поддиапазоны диапазона запроса. Алгоритм работает, сначала помечая каждый лист, который является поддиапазоном интервала *I*. Затем он работает поуровнево, поднимаясь вверх по дереву, и если у узла отмечены оба дочерних элемента, то этот узел помечается, и с дочерних элементов пометка снимается. Алгоритм останавливается после обработки корневого узла.

Рассмотрим простой интервал I = [1,5] в универсальном интервале U = [1,16]. На нижнем уровне дерева мы отмечаем узлы, представляющие следующие интервалы: [1,1], [2,2], [3,3], [4,4] и [5,5]. Затем мы поднимаемся на один уровень вверх и обнаруживаем, что у узла [1,2] отмечены оба его дочерних элемента, [1,1] и [2,2], поэтому мы отмечаем [1,2] (и снимаем пометки с [1,1] и [2,2]). Аналогичным образом помечается [3,4], потому что [3,3] и [4,4] помечены, и снимаются пометки с [3,3] и [4,4]. На третьем уровне снизу мы помечаем [1,4], потому что [1,2] и [3,4] помечены, а [1,2] и [3,4] не помечены. Таким же образом обрабатываются узлы со всех других уровней до самого корня, но мы больше не встречаем узлов, оба дочерних элемента которых помечены как `True`. Следовательно, осталось два помеченных узла, и они соответствуют поддиапазонам [1,4] и [5,5], и мы сообщаем о них как о диадических диапазонах. Эта функциональность проиллюстрирована в следующем ниже исходном коде:

```
def markNodes(root, lower, upper):
    if root is None:                                          ❶
        return
    queue = [root]
    stack = deque()
    while(len(queue) > 0):
        stack.append(queue[0])
        node = queue.pop(0)
        if node.left is not None:
            queue.append(node.left)
        if node.right is not None:
            queue.append(node.right)

    while(len(stack) > 0):                                    ❷
        i = stack.pop()
        if i.data[0] >= lower and i.data[1] <= upper and
               ➥ i.left is None and i.right is None:         ❸
            i.marked = True

        if i.left is not None and i.right is not None:
            if i.left.marked and i.right.marked:              ❹
                i.left.marked = False
                i.right.marked = False
                i.marked = True
```

```
def inorderMarked(root):                                    ❺
    if root is None:
        return
    inorderMarked(root.left)
    if root.marked:
        print(root.data)
    inorderMarked(root.right)
```

❶ Сначала пройти по узлам в поуровневом порядке (обход сперва в ширину)
❷ Узлы в стеке хранятся в поуровневом порядке, начиная с листьев
❸ Каждый лист внутри интервала помечен
❹ Помечать внутренние узлы, оба дочерних элемента которых были помечены, и снимать пометку с дочерних узлов
❺ Распечатать диадические диапазоны

Вот как эта реализация работает на примере универсального интервала $U = [1,16]$ и интервала $I = [3,13]$:

```
k = 4
root = intervalToBST(1, 2**k)
markNodes(root, 3, 13)
inorderMarked(root)
```

Выходные диадические интервалы таковы:

```
(3, 4)
(5, 8)
(9, 12)
(13, 13)
```

Время работы алгоритма в наихудшем случае равно времени, асимптотически требуемому алгоритму поиска сперва в ширину в дереве универсального множества, следовательно, $O(n)$.

Резюме

- Задачи, связанные с оцениванием частот, обычно возникают при анализе больших данных, в особенности в наборах, содержащих большое число появлений очень малого числа элементов и малое число появлений большого числа элементов. Несмотря на то что в условиях стандартной оперативной памяти задача оценивания частот решается просто в линейном пространстве, ее решение становится очень сложным в контексте обработки потоковых данных, где разрешен только один проход по данным и сублинейное пространство.
- Набросок count-min хорошо подходит для решения задачи о приближенных тяжеловесах, а также многих других задач в области обработки сенсорных данных и естественного языка.

- Набросок count-min очень пространственно экономичен и имеет два параметра ошибки: ε (контролирующий диапазон завышения оценки частоты) и δ (контролирующий вероятность неуспеха), которые можно настраивать и которые определяют размеры наброска. Если допустимая полоса ошибки завышенной оценки частоты поддерживается в виде фиксированного процента от общего числа данных N, то объем пространства в наброске count-min не зависит от размера набора данных.
- Используя набросок count-min для диапазонных запросов, можно давать довольно точные оценки частот, раскладывая диапазон на множество диадических диапазонов и используя O(log n) набросков count-min.

Глава 5

Оценивание кардинального числа и алгоритм HyperLogLog

Эта глава охватывает следующие ниже темы:

- примеры практического применения пространственно-эффективных алгоритмов оценивания кардинального числа;
- постепенное развитие идей, подводящих к алгоритму HyperLogLog, таких как вероятностный подсчет и алгоритм LogLog;
- принцип работы алгоритма HyperLogLog, его требования к пространству и ошибке, а также области его применения;
- экспериментальная симуляция с целью анализа поведения разных оценок кардинального числа на крупных данных;
- понимание практических реализаций алгоритма HyperLogLog.

Задача определения кардинального числа мультимножества (множества с дубликатами) получила широкое распространение и возникает во всех областях разработки программного обеспечения, в особенности в приложениях, связанных с базами данных, сетевым трафиком и т. д. Однако с расширением интернет-служб, в которых миллиарды кликов, поисковых запросов и покупок совершаются ежедневно гораздо меньшим числом несовпадающих пользователей, интерес к этой фундаментальной задаче возрос с новой силой. В частности, существует большой интерес к разработке алгоритмов и структур данных, которые способны оценивать кардинальное число мультимножества за одно сканирование данных и в объеме пространства, существенно меньшем, чем число несовпадающих элементов.

Сегодня процедуры оценивания кардинального числа используются для определения числа несовпадающих посетителей, заинтересованных в конкретном товаре, числа разных пользователей, использующих те или иные функциональные возможности веб-приложения, и способов обнаружения внезапных изменений в числе несовпадающих IP-адресов источник–местоназначение, проходящих через маршрутизатор (потенциально

указывающих на DoS-атаку[37]). Информация во Всемирной паутине реплицируется снова и снова таким образом, что измерение кардинального числа также помогает определять, со сколькими несовпадающими частями контента мы имеем дело; например, числом несовпадающих новостных статей или копий контента определенного веб-сайта.

В связи с сегодняшними крупными наборами данных растет интерес к разработке алгоритмов, которые могут точно аппроксимировать кардинальное число множества в объеме пространства, существенно меньшем, чем само множество. В этой главе будет рассмотрен один из таких алгоритмов, именуемый HyperLogLog, но сначала давайте углубимся в одно классическое применение измерения кардинального числа, чтобы понять причину, по которой классические алгоритмические решения по измерению кардинального числа не соответствуют требованиям.

5.1 Подсчет числа несовпадающих элементов в базах данных

Возможно, один из наиболее знакомых примеров измерения кардинального числа проистекает из баз данных и того, как в SQL используется ключевое слово `DISTINCT`. Если применить операцию `SELECT DISTINCT` к одному столбцу таблицы, то она вернет все несовпадающие элементы в этом столбце, тогда как операция `SELECT COUNT DISTINCT` вернет число несовпадающих элементов в данном столбце.

Запросы с операцией `COUNT DISTINCT` очень распространены, в особенности в электронной коммерции, когда мы хотим получить статистику использования веб-сайта. Данные о посещениях пользователей нередко заносятся в таблицу `DAILY_VISITS`, которая, как правило, становится очень большой, имея в своем составе такие атрибуты, как `session_id`, `timestamp`, `product_id`, `user_ip_address`, `visit_duration` и др. Выполнив операцию `SELECT`

```
SELECT COUNT (DISTINCT user_ip_address) WHERE product_id = 9873947
FROM DAILY_VISITS
```

мы получим число несовпадающих IP-адресов (то есть пользователей), обратившихся к товару с ИД `9873947` в определенный день. На оживленном веб-сайте таблица ежедневных посещений может вырастать до нескольких миллиардов строк, и этот конкретный запрос может занимать какое-то время.

Задержка в основном связана с операцией сортировки, которую классическая операция `COUNT DISTINCT` выполняет в большинстве баз данных (например, в Azure SQL/SQL Server), если только столбец не был предварительно упорядочен. После сортировки столбца все дубликаты окажутся рядом друг с другом, и одного последовательного сканирования будет достаточно, чтобы идентифицировать и подсчитать число несовпадающих элемен-

[37] Англ. denial-of-service (DoS); отказ в обслуживании. – *Прим. перев.*

тов. Операция сортировки стоит $O(n \log_2 n)$ в таблице с n строками и плохо масштабируется даже на несколько миллионов, не говоря уже о нескольких миллиардах строк. Что еще хуже, даже простые запросы выполняют большое число операций `COUNT DISTINCT` и `GROUP BY` на разных столбцах, и сортировка одного столбца не помогает снижать сложность сортировки другого. Можно было бы задействовать хеш-таблицу, чтобы ускорить процесс, но хеш-таблица по-прежнему требует линейного пространства в числе несовпадающих элементов k. Поскольку k может доходить до n, мы не сможем позволить себе использовать и хеширование.

Сложность остается, даже когда требуется узнать лишь число несовпадающих элементов, содержащихся в мультимножестве, и не требуется перечислять сами несовпадающие элементы. Для того чтобы в этом убедиться, рассмотрим задачу об уникальности элементов[38], в которой, имея массив из n элементов, необходимо определить, все ли элементы в нем уникальны; эта задача имеет нижнюю границу $\Omega(n \log_2 n)$ [1].

Для урегулирования трудностей масштабирования разработчики новых версий систем управления базами данных и складов данных обращаются к оценкам кардинального числа: в SQL Server 2019 есть операция `APPROX_COUNT_DISTINCT` (**http://mng.bz/QWjm**), которая занимает очень мало пространства и работает быстро. Google BigQuery идет еще дальше, и этот приближенный и вероятностный подход используется в операции `COUNT_DISTINCT` по умолчанию, оставляя операцию `EXACT_COUNT_DISTINCT` для ситуаций, когда необходим абсолютно точный ответ (**http:// mng.bz/y4PJ**). В основе этих оценок лежит алгоритм под названием HyperLogLog, первоначально изобретенный Флажоле (Flajolet) и соавт. [2], который обеспечивает поразительную экономию пространства (например, в килобайтах) при обработке наборов данных размером в триллион и низкую частоту ошибки – порядка $O(1/\sqrt{m})$, где m обозначает число ячеек памяти шириной 5 или 6 бит. Одним из распространенных вариантов для m является 2^{14}.

В этой книге мы рассмотрели ряд примеров экономии пространства в обмен на отказ от некоторой точности; однако алгоритм HyperLogLog придает совершенно новый смысл экономичному использованию пространства, почти всегда оставаясь в пределах нескольких килобайтов, при этом получая истинное кардинальное число с малой частотой ошибки (например, ±2%) в среднем.

Следующий далее раздел посвящен постепенному изложению идей, подводящих к алгоритму HyperLogLog. Мы представим изначальный алгоритм и несколько примеров, симуляции и математическую интуицию, вытекающую из него, а также упомянем несколько способов реализации и оптимизации алгоритма HyperLogLog такими компаниями, как Redis, Google, Facebook и др.

Является ли HyperLogLog структурой данных или же это алгоритм (и имеет ли это значение)? Изначально HyperLogLog назывался алгоритмом, и мы будем называть его алгоритмом, когда сосредоточимся на процедуре, ко-

[38] Англ. element-distinctness problem. – *Прим. перев.*

торая выполняется на входных данных. Однако HyperLogLog также должен хранить массив со значениями, которые вычисляются на входных данных, и эта структура нередко хранится для дальнейшего использования, как мы увидим в примере агрегирования в разделе 5.5. В таком контексте мы также поговорим о HyperLogLog как о структуре данных.

5.2 Постепенное конструирование алгоритма HyperLogLog

Суть идеи алгоритма HyperLogLog (HLL) состоит в использовании вероятностных и статистических свойств равномерно распределенных случайных битовых строк для угадывания кардинального числа мультимножества. С этой целью элементы первоначально хешируются в битовые строки: в оригинальной реализации алгоритма HyperLogLog используются 32-битовые хеши, а в более поздней реинкарнации Google под названием HyperLogLog++ [3] и реализации Redis (**http://antirez.com/news/75**) используются 64-битовые хеши, чтобы вмещать сколь угодно большие кардинальные числа. Хеши не являются случайными, и невозможно получить случайные данные из неслучайных; однако для наших целей они достаточно хорошо имитируют случайность (то есть *выглядят* случайными).

Имея мультимножество $M = \{a_1, a_2, \ldots, a_n\}$ с n элементами и k несовпадающими элементами (мы не знаем k) и используя хеш-функцию $h: U \rightarrow \{0,1\}^L$, мы генерируем хешированное множество $h(M) = \{h_1, h_2, \ldots, h_n\}$, где $h_i = h(a_i)$ с длиной хеша $L = |h_i|$. Для достаточно большого L (например, $L = 64$) каждый несовпадающий элемент будет соотнесен с несовпадающим хешем с высокой вероятностью, так что число несовпадающих хешей тоже будет равно k или очень близко. Хеширование само по себе пока не помогает нам оценивать кардинальное число, но теперь мы перешли от оценивания числа несовпадающих входных элементов к оцениванию числа несовпадающих хешей.

Мы решили, что лучше всего продемонстрировать принцип работы алгоритма HyperLogLog, постепенно наращивая простейшие алгоритмы, выявляя их недостатки и переходя ко все более сложным алгоритмам. Для того чтобы помочь в понимании, мы опускаем некоторые технические детали, показывая Python-подобный псевдокод, а не сам исходный код.

Другими словами, мы попытаемся поставить себя на место изобретателей алгоритма HyperLogLog и начать с чего-то простого и постепенно это улучшать. Будем надеяться, что вы не только научитесь разбираться в конечном продукте, но и в итеративном процессе конструирования алгоритма и внесения небольших улучшений на каждой стадии. В какой-то момент подразделы 5.2.1–5.2.4, возможно, покажутся слегка громоздкими с математической точки зрения. Но не волнуйтесь; раздел 5.4 содержит эксперимент, который тестирует три версии алгоритма, ведущего к алгоритму HyperLogLog, и который должен помочь вам понять идеи, лежащие в основе алгоритмов, и необходимость соответствующих улучшений.

5.2.1 Первая примерка: вероятностный подсчет

Самая грубая оценка, именуемая *вероятностным подсчетом*[39] [4], учитывает битовые регулярности в хеше за счет вычисления ρ_i для каждого хеша h_i таким образом, что

$$\rho_i = (\textit{число замыкающих нулей в } h_i) + 1.$$

То есть ρ_i будет обозначать позицию первой единицы, встречающейся справа (если хеш не содержит никаких единиц, то $\rho_i = L + 1$). Без потери общности в этом и других местах данной главы мы будем использовать правую сторону вместо левой. Например, для $h_1 = 1100$, $h_2 = 0111$ и $h_3 = 0000$ соответствующие значения ρ_i равны $\rho_1 = 3$, $\rho_2 = 1$ и $\rho_3 = 5$. Оценка кардинального числа E будет зависеть от $\rho_{\max} = \max(\rho_1, \rho_2, \ldots, \rho_n)$, и она равна

$$E = 2^{\rho_{\max}}.$$

Ниже приведена идея вероятностного подсчета, выраженная в Python-подобном псевдокоде:

```
p_max = 0
for a in M                                    ❶
    h = hash(a)
    p = num_trailing_zeros(h) + 1
    if(p > p_max)
        p_max = p
return 2**p_max
```

❶ M – это мультимножество, кардинальное число которого мы хотим измерить

Пример 1

На рис. 5.1 показан вероятностный подсчет в действии, где $n = 12$, $k = 7$ и окончательная оценка $2^5 = 32$, причем элемент *лимон* существенно влияет на оценку.

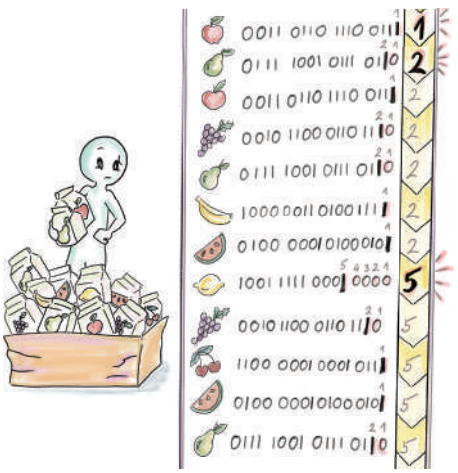


Рисунок 5.1 Набор данных из 12 элементов хешируется в 16-битовые хеши. При сканировании набора данных мы поддерживаем текущий максимум ρ_i. В этом примере элемент *лимон*, хеш которого равен `1001 1111 0001 0000`, содержит максимальное значение $\rho_{\max} = 5$, а наша оценка кардинального числа равна $E = 2^{\rho_{\max}} = 32$, но истинное число несовпадающих элементов равно $k = 7$

[39] Англ. probabilistic counting. – *Прим. перев.*

Это не так близко к истине, как нам бы хотелось, но мы только начинаем. Грубый интуитивный вывод, вытекающий из вероятностного подсчета, заключается в следующем: если бы нам удалось получить необычный хеш (то есть хеш со многими замыкающими нулями), то это было бы показателем наличия многих других хешей в множестве. Давайте посмотрим, почему это так, но перед этим следует учитывать, что с данной оценкой мы все еще далеки от истины, поэтому не привязывайтесь к этому методу и точно так же не ожидайте, что математическое объяснение, которое воспоследует, будет истиной, выбитой в камне; мы говорим приближенно.

Давайте наденем вероятностные шляпы: в равномерно случайно сгенерированном наборе из k битовых строк в среднем около $k/2$ битовых строк имеют 0 в качестве последней цифры, а остальные $k/2$ имеют 1. Из первых $k/2$ в среднем половина (то есть $k/4$) имеют 00 в качестве двух последних своих цифр, другие $k/4$ имеют 10 и т. д. В конечном счете $k/2$ элементов в среднем имеют свои последние i цифр в качестве одних нулей, а другие имеют свои последние i цифр в форме 10^{i-1}.

Соответственно, вероятность генерирования хеша, где $\rho_i = 1$ (хеш заканчивается на 1), равна 1/2, вероятность хеша, где $\rho_i = 2$ (хеш заканчивается на 10), равна 1/4, а вероятность хеша, где $\rho_i = i$ (заканчивается на 10^{i-1}), равна $1/2^i$. Для события, которое происходит с вероятностью $1/2^i$, в среднем нужно 2^i повторений, чтобы оно произошло, таким образом, двигаясь в обратном направлении, наличие элемента с $\rho_i = \rho_{\max}$ в среднем подразумевает кардинальное число $2^{\rho_{\max}}$, которое соответствует оценке в результате вероятностного подсчета.

Однако это всего лишь *усредненное* поведение случайных величин (то есть математическое ожидание), а эмпирическая величина зачастую далека от среднего. Рассмотрим набор данных с двумя точками данных, 0 и 100; среднее значение равно 50, мало что говоря о фактических значениях в наборе. Аналогичные вещи происходят и со случайными величинами, где будут иметь место отклонения от этого среднего значения, и даже малое отклонение может существенно влиять на оценку, если учитывать, что $\rho_{\max}$ находится в экспоненте. В общем и целом мы наблюдаем оценочную ошибку алгоритма HyperLogLog как *относительную ошибку* – долю истинного кардинального числа, на которую оценка отклоняется в любое направление ($\pm\frac{E-k}{k}$); для малых кардинальных чисел эта доля может быть очень большой.

5.2.2 Стохастическое усреднение, или «Когда жизнь преподносит вам лимоны»

С нашим первым примерочным решением есть пара проблем: даже при отсутствии влияния выбросов на оценку все оценки являются степенями 2, что для многих значений кардинального числа делает невозможным приблизиться к правильному ответу. В целях урегулирования проблемы выбросов мы прибегнем к методу, именуемому *стохастическим усреднением*, который разбивает множество хешей равномерно случайно на $m = 2^b$

подмножеств примерно одинакового размера, бросая хеши в корзины, задаваемые первыми b битами каждого хеша. После того как каждый хеш был назначен корзине, мы выполняем вероятностный подсчет по каждой корзине в отдельности: вместо 1-го оценщика ρ_{max} у нас будет m оценщиков $\rho_{i,max}$, $1 \leqslant i \leqslant m$, где $\rho_{i,max}$ представляет собой ρ_{max} хешей из i-й корзины.

Деление на подмножества можно трактовать как хеширование «для бедных» всего множества m раз и получение m оценок, которые в дальнейшем можно скомбинировать. На самом деле мы не можем себе позволить m хеш-функций и вычислительные затраты на хеширование каждого элемента m раз.

Теперь, когда у нас есть m оценок, мы сначала вычислим их среднее арифметическое

$$A = \frac{\sum_{i=1}^{m} \rho_{i,max}}{m}$$

и применим его, чтобы получить среднюю оценку по корзинам

$$E_{\text{корзина}} = 2^A,$$

эквивалент среднего геометрического значения оценок, полученных в результате вероятностного подсчета, для индивидуальных корзин. Получение совокупной оценки E предусматривает учет всех m корзин:

$$E = m \times E_{bucket} = m \times 2^{\frac{\sum_{i=1}^{m} \rho_{i,max}}{m}}.$$

Пример 1 (продолжение)

Давайте посмотрим, как это работает при $b = 2$, и, следовательно, существует $m = 4$ корзин. На рис. 5.2 показано содержимое и $\rho_{i,max}$ для каждой корзины. Вычисляя оценку, мы сначала вычисляем $A = (2 + 2 + 5 + 1) / 4 = 2.5$. Отсюда мы получаем, что $E_{\text{корзина}} = 2^A = 22.5 \approx 5.66$, а $E = m \times E_{\text{корзина}} = 4 \times 5.66 = 22.64$, что точнее, чем наша предыдущая оценка, равная 32. Предельное значение, к которому мы стремимся, равно 7.

Следующий ниже Python-подобный псевдокод показывает принцип работы стохастического усреднения:

```
m = 2**b                                    ❶
S = 0                                       ❷

for a in M                                  ❸
  h = hash(a)
  p = num_trailing_zeros(h) + 1
  bucket = first_bits(h, b)  ....           ❹
  if(p > S[bucket])
      S[bucket] = p
```

```
    sum = 0
    for item in S
        sum += item

    arit_avg = sum / m
    return m * 2**avg
```

❶ m представляет число корзин; b – число битов, используемых для индексации в корзине
❷ S – это список/массив из m записей, хранящий максимальные замыкающие нули в каждой корзине
❸ Прокручивает мультимножество M в цикле
❹ Целое число, описываемое первыми b битами из h

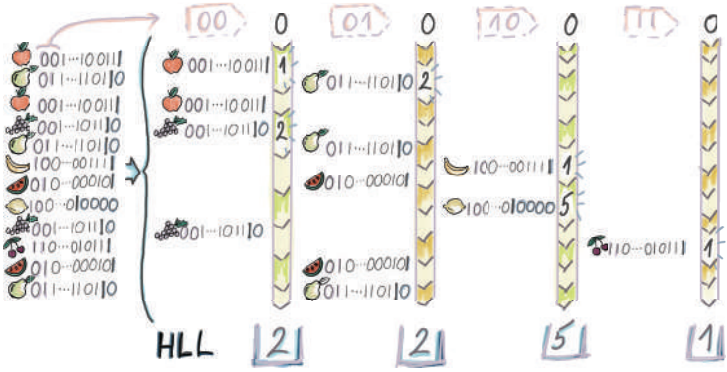


Рисунок 5.2 В этом примере каждый хеш соотносится с корзиной на основе его первых двух бит (например, хеш, соответствующий элементу *виноград*, соотносится с корзиной `00`, тогда как хеш, соответствующий элементу *груша*, соотносится с корзиной `01`. При выполнении этого процесса на крупных наборах данных мы ожидаем, что каждая корзина будет получать одно и то же число несовпадающих хешей. Каждая корзина вычисляет свою величину $\rho_{i,max}$, что в данном случае приводит к значениям 2, 2, 5 и 1 корзины. Теперь хеш элемента *лимон* влияет только на значение, хранящееся в корзине `10`

5.2.3 Алгоритм LogLog

В алгоритме LogLog стохастическое усреднение используется в сочетании с нормализующей константой $\tilde{a}_m$, которая вводится для устранения смещения из-за систематического завышения оценки, возникающего при оценивании кардинального числа с помощью случайной величины $\rho_{i,max}$ (максимума геометрических величин параметра 1/2). Следовательно, мы меняем изначальную оценку на следующую ниже формулу:

$$E = \tilde{a}_m \times m \times 2^{\frac{\sum_{i=1}^{m} \rho_{i,max}}{m}},$$

где константа $\tilde{a}_m$ параметризуется величиной m и равна

$$\tilde{a}_m \sim 0.39701 - \frac{2\pi^2 + (\ln 2)^2}{48m}.$$

Для большинства практических целей (в частности, при $m \geqslant 64$) можно использовать $\tilde{a}_m = 0.39701$. Более подробная информация о формальном выведении выражения для $\tilde{a}_m$ находится в оригинальной статье об алгоритме LogLog [5].

Пример 1 (продолжение)

При получении оценки алгоритмом LogLog для текущего примера (на рис. 5.2) мы вычисляем $\tilde{a}_4$, приближенно равное 0.292; таким образом, оценка алгоритма LogLog равна $0.292 \times 22.6 \approx 6.6$, которая находится чрезвычайно близко к истинному кардинальному числу 7!

Соображения по поводу ошибки и пространства в алгоритме LogLog

На основе статистического анализа было обнаружено, что относительная ошибка в алгоритме LogLog может близко аппроксимироваться величиной $1.3/\sqrt{m}$. Если посмотреть на это в перспективе, то во многих современных реализациях значение m часто устанавливается равным 2^{14}, и можно ожидать, что относительная ошибка будет составлять

$$1.3/\sqrt{2^{14}} = 1.01\ \%,$$

независимо от размера набора данных. Если принять во внимание, что 2^{14} 8-байтовых целочисленных ячеек занимают всего около 130 Кб, алгоритм LogLog может показаться каким-то волшебством!

Тем не менее важно понимать, что для корзинных счетчиков не нужно 8 байт. На самом деле нужно пять или шесть бит, в зависимости от величины оцениваемых кардинальных чисел. Если верхним пределом кардинального числа набора данных является k_{max}, то нам нужно, чтобы $O(\log_2 k_{max})$ было длиной хеша, дабы продифференцировать вплоть до этого кардинального числа, а затем нужно $O(m \log_2 \log_2 k_{max})$ бит, чтобы сохранить максимальное значение в корзине (отсюда и LogLog). Безопасный верхний предел кардинального числа равен $k_{max} = 2^{64}$, поэтому для одной корзины требуется шесть бит. Общие потребности в хранилище в рамках алгоритма LogLog составляют:

$$O(m \log_2 \log_2 k_{max}).$$

Подставив $m = 2^{14}$ в качестве общего значения, оказывается, что для хранения структуры данных LogLog требуется примерно 12 Кб.

Если быть точнее, мы ожидаем, что максимальное кардинальное число внутри одной корзины будет ближе к k_{max}/m (k_{max} – это наихудший случай), что сокращает потребность в пространстве до

$$O\left(m \log_2 \log_2 \left(\frac{k_{max}}{m}\right)\right).$$

Однако в нашем примере, где $k_{max}/m = 2^{50}$, это не помогает, так как логарифмы округляются вверх до их целочисленных значений (в данном случае логарифм из 50 будет округлен вверх до 6).

Алгоритм SuperLogLog

Частоту ошибки в алгоритме LogLog можно попробовать снизить за счет поддержания только процента θ самых низких корзинных значений и основывать оценку на этих $m_\theta = \theta m$ корзинах. Это называется *правилом усечения*. В аналогичном подходе, именуемом *правилом ограничения*, используются только те корзинные значения, которые не превышают $\lceil \log_2 k_{max}/m + 3 \rceil$, что устраняет выбросы, но и позволяет использовать корзины шириной в $\lceil \log_2 \lceil \log_2 k_{max}/m + 3 \rceil \rceil$ бит. Существуют экспериментальные подтверждения, что при задействовании правил усечения и ограничения частота ошибки падает до $1.05/\sqrt{m}$.

Даже если это лучше по сравнению с базовым подходом на основе вероятностного подсчета, среднее арифметическое в экспоненте все равно может уводить окончательную оценку сколь угодно далеко от среднего, поскольку среднее арифметическое очень чувствительно к выбросам. В трехмерном контексте это аналогично центроиду (трехмерной версии среднего арифметического), который может оказываться сколь угодно далеко от центра масс из-за того, что одна точка будет находиться далеко от всех остальных. В нашем последнем усовершенствовании, алгоритме HyperLogLog, для вычисления оценки будет использоваться среднее гармоническое из корзинных значений.

5.2.4 Алгоритм HyperLogLog: стохастическое усреднение вместе с гармоническим средним

Формула гармонического среднего применительно к корзинам, которая представляет наше новое среднее корзинное значение, выглядит следующим образом:

$$E_{\text{корзина}} = \frac{m}{\sum_{i=1}^{m} 2^{-\rho_{i,max}}}.$$

В окончательной оценке мы применим соответствующий коэффициент поправки смещения, α_m, и учтем все m сегментов:

$$E = a_m \times m \times E_{\text{корзина}} = \frac{a_m m^2}{\sum_{i=1}^{m} 2^{-\rho_{i,max}}}.$$

Коэффициент поправки смещения отличается от такого же коэффициента в алгоритме LogLog, и его можно аппроксимировать следующим образом:

$$a_m = \frac{1}{2 \ln 2 \left(1 + \frac{1}{m}(3 \ln 2 - 1) + O(m^{-2})\right)}.$$

Для очень больших значений m значение $\alpha_m = 1/(2\ln 2) = 0.72134$ – неплохая аппроксимация, вместе с тем в исходный код также полезно встраивать несколько типичных значений α_m:

$$\alpha_{16} = 0.673;$$
$$\alpha_{32} = 0.697;$$
$$\alpha_{64} = 0.709;$$
$$\alpha_m = 0.723 / (1 + 1.079/m) \text{ для } m \geqslant 128.$$

Пример 1 (продолжение)

Применив среднее гармоническое значение к текущему примеру из рис. 5.2, мы получим

$$E_{\text{корзина}} = \frac{4}{\left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^5 + \left(\frac{1}{2}\right)^1} = \frac{4}{\frac{33}{32}} \approx 3.88.$$

Мы также получим $\alpha_4 = 0.541$ из формулы α_m, которая далее дает следующую ниже оценку:

$$E = 0.541 \times 4 \times 3.88 = 8.39.$$

Эта оценка еще дальше от истины, чем предыдущая оценка алгоритма LogLog (6.6), но по мере увеличения наборов данных, как мы увидим в симуляциях раздела 5.4, алгоритм HyperLogLog является менее смещенным оценщиком и имеет меньшую относительную ошибку. Статистический анализ показывает, что относительная ошибка в алгоритме HyperLogLog составляет до $1.04/\sqrt{m}$. Более подробная информация о вычислении частоты ошибки в алгоритме HyperLogLog находится в оригинальной статье об алгоритме HyperLogLog [6].

Вот мы и закончили наш рассказ о процедуре получения сырой оценки алгоритмом HyperLogLog, чей Python-подобный псевдокод приведен ниже (первая часть фрагмента псевдокода, исключая установку альфа-параметров, идентична приведенному ранее фрагменту псевдокода). После получения сырой оценки есть несколько незначительных уточнений, в частности когда вычисляемое кардинальное число стало слишком малым или слишком большим:

```
alpha16 = 0.673                                    ❶
alpha32 = 0.697
alpha64 = 0.709
alpha_m = 0.7213/(1 + 1.079/m) for m>= 128

m = 2**b
S = 0

for a in M
```

```
    h = hash(a)
    p = num_trailing_zeros(h) + 1
    bucket = first_bits(h, b)  ....
    if(p > S[bucket])
        S[bucket] = p

sum = 0
for item in S
   sum += 2**(-1*item)

harmonic_avg = m / sum
E = alpha_m * m * harmonic_avg          ❷

                                        ❸
if E <= 5*m/2                           ❹
    V = num_registers_zero()            ❺
    if V != 0
        E_final = mlog(m/V)
    else
        E_final = E

if E <= 2**32 / 30                      ❻
    E_final = E
if E > 2**32 / 30                       ❼
    E_final = -2**32 * log(1 - E/2**32)

return E_final                          ❽
```

❶ Устанавливает альфа для разных значений m
❷ Сырая оценка
❸ Вычисляет исправленную оценку
❹ Поправка в малом диапазоне
❺ Обозначим через V число регистров, равное 0
❻ Промежуточный диапазон, без поправки
❼ Поправка в большом диапазоне
❽ Исправленная оценка с относительной ошибкой ±1.04/sqrt(m)

В случае очень малых кардинальных чисел (по отношению к числу корзин) многие корзины останутся пустыми, и тогда мы будем прибегать к вероятностному методу, именуемому *линейным подсчетом*, чтобы определиться с истинным кардинальным числом. Этот подход следует логике игры в шары и корзины, в которой, если равномерно случайно бросать n шаров в m корзин, то, основываясь на числе оставшихся пустыми корзин, можно оценить общее число шаров. Более подробная информация о линейном подсчете находится в статье [7], посвященной указанной теме.

Интересным результатом использования линейного подсчета является то, что непосредственно в точке пересечения, когда кардинальное число стано-

вится достаточно большим, чтобы переключиться на оценку HyperLogLog, наблюдается большой всплеск в смещении. Авторы алгоритма HyperLogLog++ попытались смягчить эту проблему, экспериментально установив средние величины смещения для каждого кардинального числа вокруг этой точки, а затем возвращая оценку по этой величине смещения. В реализации Redis используется полиномиальная регрессия, которая аппроксимирует кривую смещения, а затем возвращает оценки по этой предсказанной величине.

Учитывая, что наш псевдокод отражает реализацию алгоритма HyperLogLog из оригинальной статьи и то, как 32-битовые хеши используются в ней, может возникнуть одна проблема: при очень больших кардинальных числах начинают появляться коллизии хешей, поэтому мы начинаем терять точность даже на уровне хеширования, и, следовательно, требуется поправка в оценке. Однако это не проблема, когда используется 64-битовый хеш, если учесть, как он используется во всех современных реализациях Google, Redis, Facebook (**http://mng.bz/M2z2**) и др.

Соображения по поводу ошибки и пространства в алгоритме HyperLogLog

Статистические доказательства показывают, что относительная погрешность алгоритма HyperLogLog составляет около $1.04/\sqrt{m}$. Потребление пространства такое же, как и в алгоритме LogLog:

$$O(m \log_2 \log_2 k_{max}).$$

И точно так же, как в алгоритме LogLog, можно использовать шестибитовые поля для корзин. Не скупимся ли мы, настаивая на конкретно-прикладных шестибитовых полях, в отличие от стандартных восьмибитовых полей для алгоритма, который и так занимает очень мало места в памяти, и не жертвуем ли мы ценным временем центрального процессора, распаковывая эти биты? Ответы на эти вопросы во многом зависят от того или иного приложения. Например, при встраивании алгоритмов в аппаратное обеспечение или при агрегировании большого числа массивов HyperLogLog в один такие различия суммируются, и каждый трюк с оптимизацией пространства того стоит.

Прежде чем экспериментально протестировать особенности представленных в этом разделе структур данных/алгоритмов, мы завершим техническое обсуждение примером контекста, в котором можно использовать алгоритм HyperLogLog (HLL).

5.3 Пример использования: ловля червей с помощью алгоритма HyperLogLog

Приложения и системы обнаружения вторжений, служащие для мониторинга сетевого трафика, отслеживают изменения различных сетевых параметров, которые могут выявлять надвигающиеся нарушения безопасности, например в сети организации. Один из индикаторов работоспособности

сети связан с парами IP-адресов источник–местоназначение, имеющихся в пакетных заголовках, проходящих через маршрутизатор.

Стабильный сетевой трафик характеризуется (потенциально большим) числом пакетов, которыми обменивается гораздо меньшее число пар компьютеров. Наличие у одного источника большого числа соединений с (иногда случайными) местоназначениями за короткий промежуток времени или просто значительное увеличение числа уникальных пар IP-адресов источник–местоназначение может указывать на наличие вируса [8] (см. рис. 5.3 и 5.4).

Рисунок 5.3 Исправный сетевой поток.
Довольно большое число пакетов, но малое число уникальных потоков

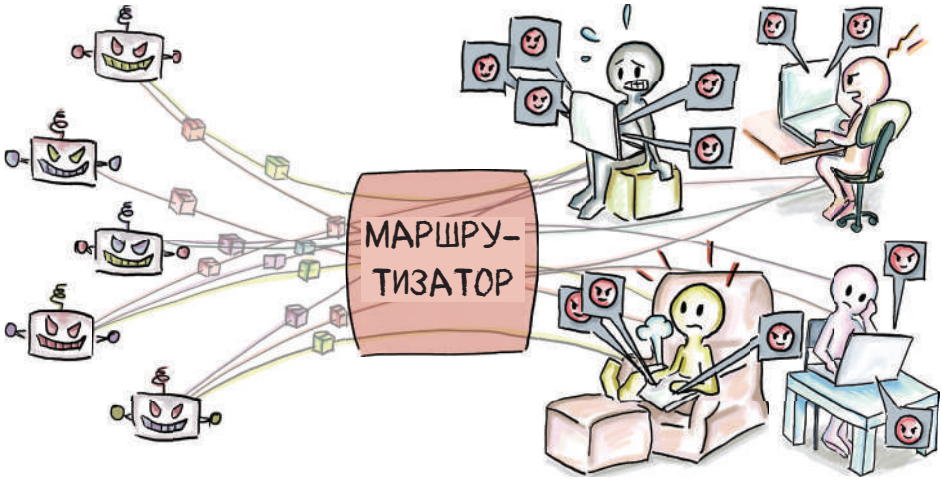


Рисунок 5.4 Подозрительный поток – наличие большого числа пар источник–местоназначение, и один источник открывает большое число уникальных соединений за короткий промежуток времени

И следовательно, бывает очень полезно встраивать алгоритм HyperLogLog в подключенное к маршрутизатору программное обеспечение, в особенности из-за необходимости быстрого выполнения вычислений и малого объема потребляемой памяти. Еще одним хорошим местом для стратегического размещения структуры данных/алгоритма HyperLogLog и других структур данных/алгоритмов, которые помогают анализировать оживленный сетевой трафик с малыми потребностями в пространстве и времени, является точка входа в сеть организации, как показано на рис. 5.5.

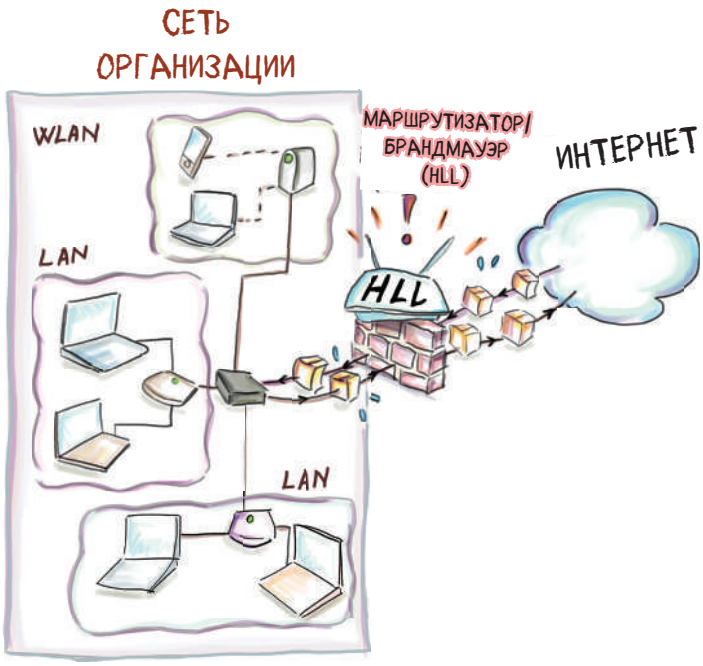


Рисунок 5.5 Размещение алгоритма HyperLogLog в точке входа внутрь организации помогает собирать ценную статистику о сетевом трафике этой организации

5.4 Но как это работает? Мини-эксперимент

В этом разделе мы выполним симуляции, чтобы получить интуитивное представление о том, как различные оценки – вероятностный подсчет, LogLog и HyperLogLog – соотносятся с точки зрения смещения и точности при выполнении на наборе данных разумного размера. Мы сконструируем эксперимент, чтобы увидеть, насколько хорошо границы ошибки, полученные в результате вероятностного анализа, соответствуют числам из практического контекста. Нас также интересует величина, на которую нормализующие коэффициенты $\tilde{a}_m$ (в алгоритме LogLog) и α_m (в алгоритме HyperLogLog) повышают точность, а также влияние числа корзин в HyperLogLog на точность и ширину распределения.

Данные всех графиков в этом разделе получены в результате выполнения следующего эксперимента 1000 раз: мы генерируем $N = 2^{16} = 65\,536$ 32-битовых строк, где каждый бит выбирается равномерно случайно. Мы начинаем с равномерно распределенных случайных строк, которые действуют как хеши (и в дальнейшем будут называться хешами), потому что мы заинтересованы в получении 1000 наборов хешей с одинаковым (или почти одинаковым) кардинальным числом. Учитывая, что может быть 2^{32} хешей, а размер набора хешей равен 2^{16}, в большинстве экспериментов мы не будем встречать коллизий хешей, и общее число несовпадающих хешей/элементов будет равно размеру набора данных, $N = k = 65\,536$; эпизодически будет происходить коллизия хешей, но число k несовпадающих элементов никогда не опускается ниже 65 531, что указывает на ничтожную разницу в кардинальных числах между разными экспериментами. Мы разработали эксперимент без дубликатов, потому что они не влияют на оценки наших методов, поэтому данный эксперимент также мог бы послужить для демонстрации поведения даже гораздо более крупных наборов хешей, чем 2^{16}, но с 2^{16} несовпадающими элементами.

На первом показанном на рис. 5.6 графике мы сравниваем следующие ниже методы:

- вероятностный подсчет;
- стохастическое усреднение с ненормализованным средним арифметическим ($m = 64$);
- стохастическое усреднение с ненормализованным гармоническим средним ($m = 64$).

Ось x показывает логарифм по основанию 2 кардинального числа; на графике указывается позиция истинного кардинального числа (в 16). Ось y показывает число проведенных экспериментов.

График показывает, что вероятностный подсчет имеет наибольшее отклонение из трех методов, при этом некоторые экземпляры эксперимента отличаются друг от друга на целых 12 единиц $\log_2$ кардинального числа и 384 экземпляра эксперимента (более трети) с $\log_2$ кардинального числа 18 и более. За вероятностным подсчетом следует стохастическое усреднение с ненормализованным средним арифметическим с разбросом примерно на 1.5 единицы $\log_2$ кардинального числа, а стохастическое усреднение с ненормализованным средним гармоническим является самым узким из трех.

Метод гармонического среднего наиболее близок к истинной оценке в среднем. В этом эксперименте средними $\log_2$ кардинального числа являются 17.31 (вероятностный подсчет), 17.32 (стохастическое усреднение с ненормализованным средним арифметическим) и 16.47 (стохастическое усреднение с ненормализованным средним гармоническим).

После нормализации среднеарифметической и среднегармонической оценок соответствующими константами $\tilde{a}_m = 0.39701$ и $\alpha_{64} = 0.709$ средние $\log_2$ кардинальных чисел снижаются соответственно до 15.97 (LogLog) и

15.93 (HyperLogLog), со средним смещением от истинного кардинального числа в обоих случаях в размере около 13 %. Получен довольно неплохой результат, если учитывать, что оценочная частота ошибки в обоих случаях составляет приблизительно 12.5 %.

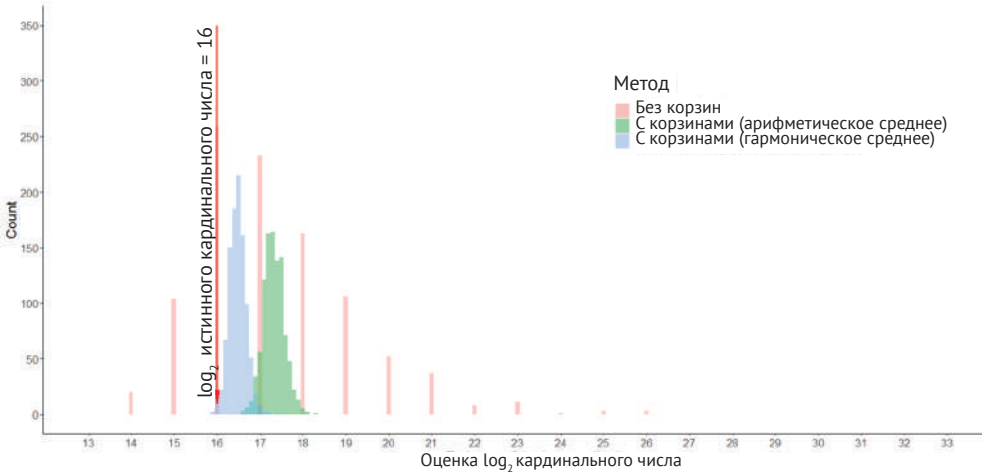


Рисунок 5.6 График показывает сравнение вероятностного подсчета (без корзин), стохастического усреднения со средним арифметическим (ненормализованное; с корзинами; среднее арифметическое) и стохастического усреднения со средним гармоническим (ненормализованное; с корзинами; среднее гармоническое). Все сырые оценки показывают систематическое смещение из-за завышения; однако наименьшее смещение в среднем показано методом гармонического среднего, за которым следует метод среднего арифметического среднего и вероятностный подсчет. Наибольшее отклонение в оценках (причем разные эксперименты варьируются в оценке в 2^{12} раз) проявляется при вероятностном подсчете, чьи оценки являются только степенями 2, за которыми следует метод среднего арифметического, а затем метод гармонического среднего

5.4.1 Влияние числа корзин (*m*)

Здесь мы показываем эксперимент с теми же наборами хешей, что и раньше, но на этот раз измеряем эффект от использования трех разных вариантов числа корзин в алгоритме HyperLogLog: $m = 16$, $m = 64$ и $m = 256$. Как и ожидалось, на рис. 5.7 показано, что чем больше корзин, тем с меньшей дисперсией мы сталкиваемся в полученных оценках.

Поскольку скорректированное на смещение среднегармоническое из алгоритма HyperLogLog очень близко к истине, на рис. 5.8 мы показываем тот же график, но построенный как смещение от фактического кардинального числа в каждом эксперименте (теперь ось x – это истинное кардинальное число, а не логарифм).

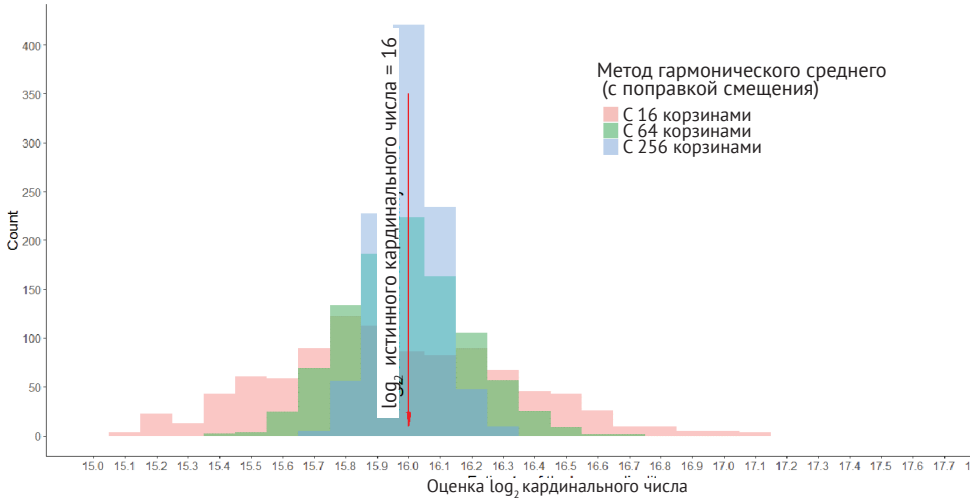


Рисунок 5.7 Влияние разных значений *m* на точность оценки $\log_2$ кардинального числа в алгоритме HyperLogLog. Чем больше число корзин, тем меньше отклонение от истинного кардинального числа. В целом метод гармонического среднего после поправки смещения редко приводит к завышению/занижению более чем на одну единицу $\log_2$ во всех трех случаях

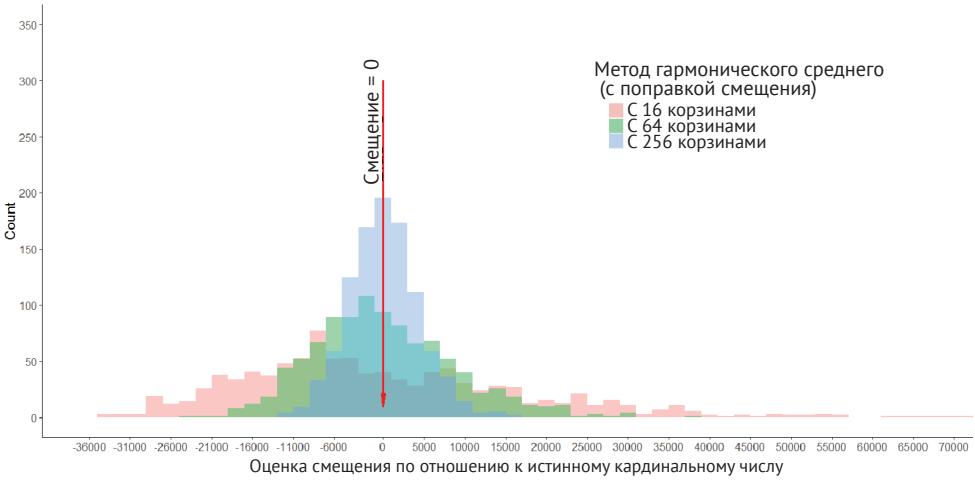


Рисунок 5.8 Влияние корзин на оценку кардинального числа в алгоритме HyperLogLog. Из большего значения *m* следует меньшее смещение и распределение, более похожее на гауссово

Как отмечалось в оригинальной статье, распределение кардинальных чисел выглядит гауссовым, с более короткими хвостами при большем *m*. Приближенное гауссово распределение помогает сделать следующий ниже практический вывод:

С учетом стандартной ошибки (или относительной ошибки) алгоритма HyperLogLog в размере $1.04/\sqrt{m}$ *соответственно около 65 %, 95 % и 99 % значений (где значение – это оценка кардинального числа для одного набора данных) будут находиться в пределах* σ, 2σ *и* 3σ *долей от истинного кардинального числа.*

Для того чтобы в этом убедиться, мы взяли случай с $m = 256$ корзинами, следовательно, $\sigma = 1.04/\sqrt{256} = 0.065$. Таким образом, 6.5 %, 13 % и 19.5 % – это, соответственно, одна, две и три стандартные ошибки от истины. Оказывается, что в нашем эксперименте соответственно 71 %, 94.8 % и 99.2 % попадают в границы упомянутых ошибок, что примерно указывает на гауссово поведение (даже немного более жесткое). Таким образом, при реализации алгоритма HyperLogLog можно ожидать, что оценки будут вести себя предсказуемым образом и чаще всего будут очень близки к среднему значению (истинному кардинальному числу).

5.5 Пример использования: агрегация с использованием алгоритма HyperLogLog

Давайте вернемся к предыдущему примеру с таблицами ежедневных посещений популярного веб-сайта потребителями. Как мы уже видели, задача вычисления числа несовпадающих значений в столбце (например, определение общего числа пользователей) в большой таблице является довольно сложной, но реальная проблема возникает, когда эти данные нужно обобщить за дни, недели, месяцы и т. д. Поддержание индивидуальных данных в течение длительного периода времени обходится очень дорого, однако для многих предприятий крайне важно иметь возможность возвращаться назад и получать соответствующую статистику за произвольный момент в прошлом. Фотографический веб-сайт Unsplash, на котором размещено большое число изображений и который посещается миллионами пользователей в день, для решения этой проблемы использует алгоритм HyperLogLog (**http:// mng.bz/aDXJ**).

Одна из трудностей, связанных с вычислением числа несовпадающих значений в одном или нескольких столбцах таблицы, заключается в том, что даже если нам волшебным образом будут даны числа несовпадающих значений, это никоим образом не поможет вычислить агрегированное число, как показано на рис. 5.9.

Однако если вместо числа несовпадающих значений мы сможем поддерживать один массив HyperLogLog в расчете на таблицу ежедневных посещений, тогда мы сможем агрегировать результаты за несколько дней, выполняя операцию объединения между двумя (или более) массивами HyperLogLog того же размера и с той же хеш-функцией, как показано на рис. 5.10.

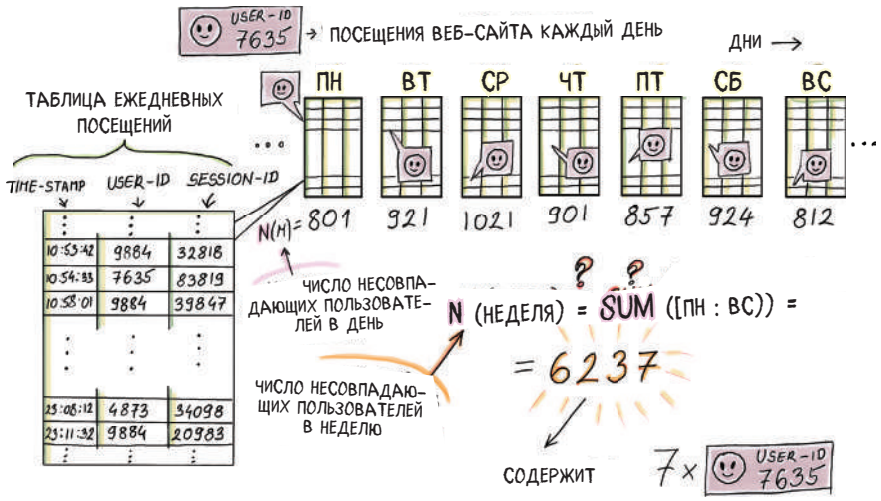


Рисунок 5.9 Каждая строка таблицы ежедневных посещений указывает на одно посещение пользователем, и в каждой таблице поддерживается отдельная переменная числа несовпадающих значений, которая отслеживает число несовпадающих пользователей. Учитывая, что некоторые пользователи возвращаются на веб-сайт повторно, невозможно просто просуммировать индивидуальные количества и получить общее число несовпадающих пользователей за неделю

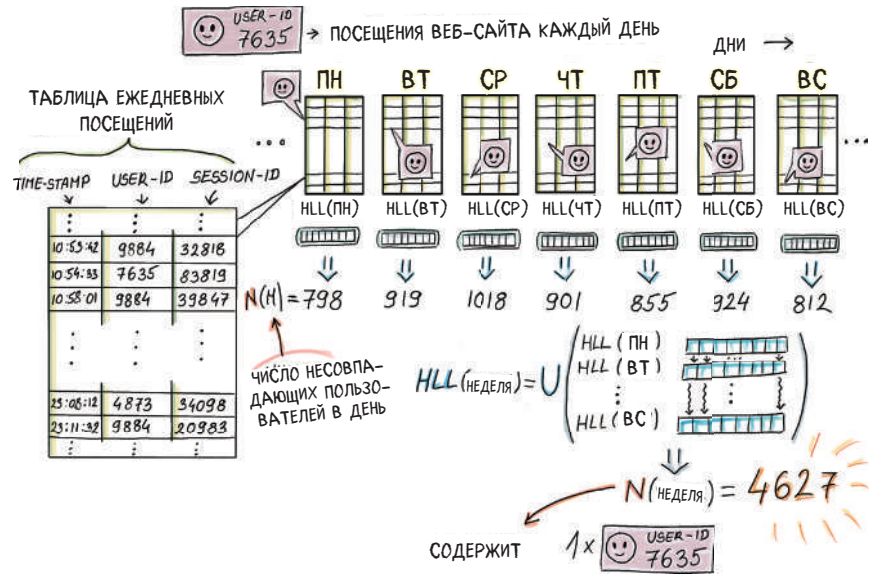


Рисунок 5.10 Поддержание одного массива HyperLogLog на таблицу ежедневных посещений помогает позже агрегировать массивы HyperLogLog за несколько дней, чтобы получать оценку для большего числа таблиц. На самом деле массив HyperLogLog можно легко закодировать, чтобы можно было поддерживать таблицу схем HyperLogLog, дабы ее декодировать впоследствии

Операция объединения двух массивов HyperLogLog `HLL1[1..m]` и `HLL2[1..m]` выполняется за счет создания нового объединенного массива HyperLogLog `HLL_UNION[1..m]` и назначения `max(HLL1[i], HLL2[i])` элементу `HLL_UNION[i]` для каждого i, $1 \leqslant i \leqslant m$. Например, объединение двух массивов HyperLogLog, значения корзин которых равны (1, 4, 2, 5) и (2, 2, 5, 3), приведет к созданию еще одного массива HyperLogLog со значениями корзин (2, 4, 5, 5).

Что происходит с частотой ошибки при объединении большого числа массивов HyperLogLog? Относительная ошибка, завися от числа корзин m, после агрегирования остается неизменной, так как число корзин остается неизменным. Но как бы сильно у нас ни возникало искушение думать, что частота ошибки в алгоритме HyperLogLog не зависит от размера набора данных, как это часто рекламируется, важно учитывать (точно так же, как в случае наброска count-min), что относительная ошибка представляет собой *процент* от истинного кардинального числа, который обычно имеет тенденцию увеличиваться с увеличением размера набора данных. Таким образом, даже несмотря на то, что после объединения частота ошибки остается неизменной, константа, на которую ошибка увеличивается, на самом деле растет пропорционально числу несовпадающих элементов.

HyperLogLog имеет простую кодировку, что делает его благоприятным для хранения в виде записи в таблице HyperLogLog, требования к пространству которой значительно меньше, чем к поддержанию эквивалентных таблиц ежедневных посещений. В силу этого появляется возможность агрегировать HLL за произвольные промежутки времени или в определенные конкретные даты, как показано на рис. 5.11.

Более того, оценки могут выполняться на многих уровнях, на которых можно агрегировать почасовые HLL в ежедневные HLL, затем использовать ежедневные HLL для вычисления еженедельных HLL и т. д. (показано на рис. 5.12). В мире традиционных баз данных выполнение нескольких группировок на разных уровнях обычно означает необходимость однократного сканирования всех данных для каждой группировки, которую мы хотим выполнить. С помощью алгоритма HyperLogLog нужно просканировать все данные только один раз, чтобы создать массивы HyperLogLog, а затем мы их только читаем и комбинируем.

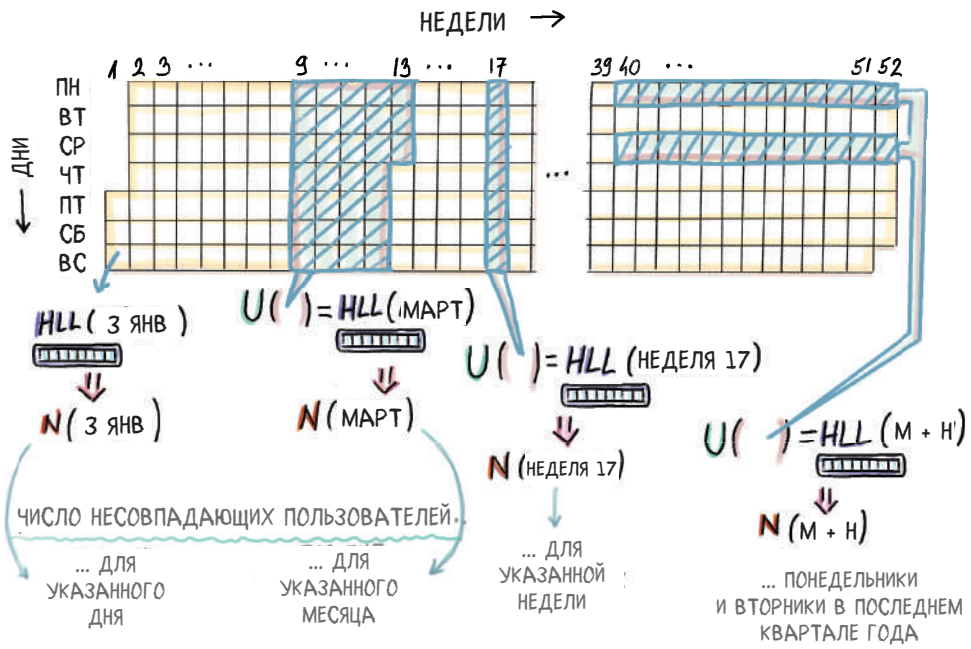


Рисунок 5.11 Храня ежедневные HLL, можно выполнять объединение интересующих данных по произвольному выбору и получать агрегированную оценку кардинального числа за указанный период

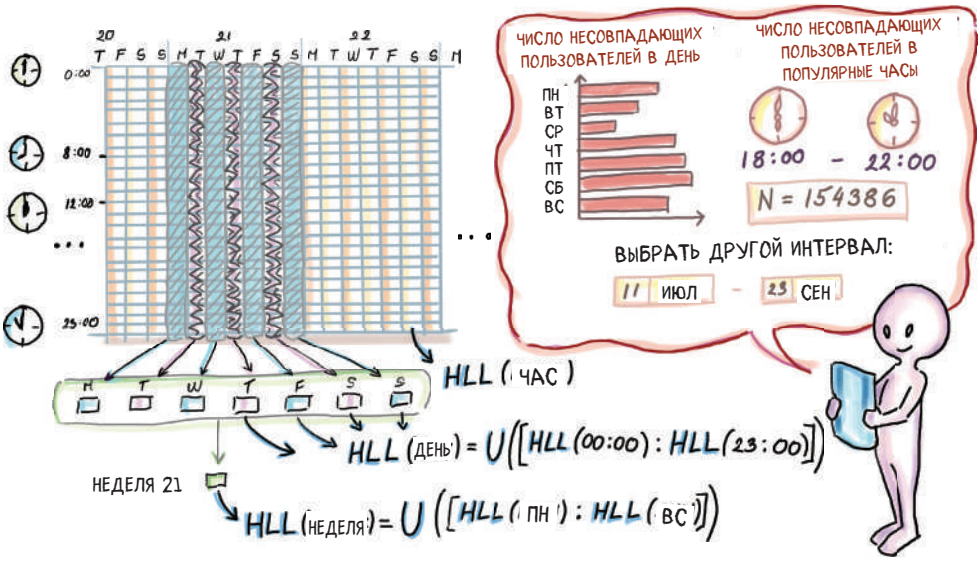


Рисунок 5.12 Агрегирование происходит на нескольких уровнях, в данном случае на часах, днях, неделях и т. д. В базах данных при группировке данных по разным временным интервалам обычно выполняется одно сканирование всех данных по каждому уровню агрегирования

Резюме

- Задача оценивания кардинального числа, или мощности множества, возникает во многих областях разработки программного обеспечения, в первую очередь в базах данных, сетевом трафике и электронной коммерции. Из-за объемов данных классические для баз данных функции точного вычисления кардинального числа заменяются вероятностными методами, которые обеспечивают значительную экономию пространства в обмен на малую ошибку в точности.
- HyperLogLog – это алгоритм/структура данных, в котором используются хеширование и вероятностные свойства случайных битовых строк для опробывания кардинального числа множества. Потребляемое им пространство равно O($m \log_2 \log_2 k$,), а относительная частота ошибки составляет $1.04/\sqrt{m}$.
- Многие компании, управляющие крупными системами, реализовали и адаптировали HyperLogLog под свои нужды, улучшив и модернизировав различные его аспекты (например, компании Google, Redis, Facebook и др.).
- Предоставляемые алгоритмом HyperLogLog оценки имеют приближенно гауссову форму. В ходе симуляций на наборе 2^{16} хешей мы установили, что алгоритм HyperLogLog подчиняется правилам гауссова распределения, позволяя примерно 70 % данных попадать в пределы одной, 95 % – в пределы двух и 99 % – в пределы трех стандартных ошибок.
- Истинная мощь алгоритма HyperLogLog видна при агрегировании большого числа крупных отдельных таблиц, представляющих данные во временной динамике. Вместо того чтобы хранить крупные таблицы, можно хранить таблицу массивов HyperLogLog и далее агрегировать и объединять массивы HyperLogLog за интересующие периоды (например, неделю, месяц, квартал и т. д.).

Часть II

Реально-временная аналитика

До сих пор нас не беспокоило состояние, в котором массивные данные поступают в наше распоряжение. Все алгоритмы, с которыми мы до этого познакомились, могут применяться как к непрерывно прибывающим данным, так и к историческим данным, обитающим в большой системе баз данных. В трех главах части II представлены алгоритмы и структуры данных (наброски), конструктивные соображения и контекст применения которых были обусловлены непрерывным прибытием кортежей данных, именуемых потоками данных. Здесь, из-за мимолетной природы располагаемых данных, алгоритмы должны работать эффективно и присоединять знания о потоке после каждого просмотренного кортежа. Это достигается за счет поддержания набросков потока данных. Некоторые из них, например случайные выборки, являются общими и могут отвечать на многие вопросы о данных. Другие, такие как t-дайджест, более специализированы, и алгоритм/структура данных адаптированы под возврат конкретного признака данных, например разных (хвостовых) процентилей. В общем и целом представление о большом объеме данных, которые прибывают с неравномерной скоростью и после обработки уходят в небытие, является хорошей отправной точкой для дальнейших действий.

Глава 6

Потоковые данные: сведение всего воедино

Эта глава охватывает следующие ниже темы:

- ознакомление с моделью конвейера обработки потоковых данных и ее распределенным фреймворком;
- определение ситуаций, в которых приложения по обработке потоковых данных пересекаются с моделями потоков данных;
- выявление мест, где алгоритмы и структуры данных вписываются в потоки данных;
- формирование базовых вычислительных ограничений и понятий, присущих потокам данных;
- освещение вероятностных предпосылок для следующих двух глав.

В предыдущих главах был представлен ряд алгоритмов/структур данных для формирования набросков (важной характеристики) огромных объемов данных, обитающих в базе данных, или прибывающих и убывающих с молниеносной скоростью, как вы увидели из алгоритма HyperLogLog при его применении для наблюдения за сетевым трафиком. В данной главе указанные алгоритмы будут обобщены.

В первой части этой главы мы не будем подробно рассматривать алгоритмы обработки массивных данных. Вместо этого мы займемся служебной работой и изучим более широкий контекст, в котором рассмотренные к настоящему моменту алгоритмы находят свое применение. На данном этапе необходимо начать работать с потоками данных, и очень кстати одним из естественных сред обитания потоков данных являются приложения по конвейерной обработке потоковых данных и их более широкая системная архитектура. Если это звучит слишком расплывчато, то рекомендуем обратиться к книге «Потоковые данные» Эндрю Г. Псалтиса (Manning, 2017)[40]. Вы не должны думать, что вы купили книгу, которая советует вам

[40] Streaming Data, Andrew G. Psaltis.

купить другую книгу; мы представим здесь достаточно из того, что вам понадобится. Однако, бегло пролистав разделы 1.1, 1.2 и 1.3 книги Псалтиса, вы сможете прояснить все, что пробудит ваш интерес еще больше. Мы будем использовать и показывать модель системы/конвейера обработки потоковых данных Псалтиса, чтобы создать условия и показать, как и где можно использовать фильтры Блума, наброски count-min и массивы HyperLogLog для экономии времени/пространства в данном конкретном архитектурном ландшафте.

В прошлом обработка потоковых данных была исключением, предназначенным только для систем, контролирующих особо важные процессы на ядерных установках или самолетах, где быстрое автоматическое реагирование на что-либо необычное означало спасение человеческих жизней. С появлением интернета мириады запросов, отправляемых пользователями на сервер или в облако по своему выбору, легко концептуально представить как потоковые данные. С появлением интернета вещей любое устройство, достаточно высокотехнологичное, чтобы измерять и затем сообщать о своем текущем состоянии на некоторое расстояние, становится одним из многих производителей, подающим на вход централизованного сервера или облака постоянный поток данных. Это происходит быстрыми темпами, непредсказуемым и изменчивым образом.

Потоки можно визуализировать как нескончаемые последовательности данных и огромные наборы данных, состоящие из многочисленных крошечных частей, но бóльшую часть времени нас не особо интересуют эти крошечные части. Вопрос типа «Какова была точная температура, которую зафиксировал датчик ИД 1092 в 11:34 15 мая 2003 года?» можно услышать только в суде, и как раз для таких целей данные хранятся в архивном хранилище. Но на ежедневной основе нас интересует именно несовершенная общая картина, сведения о которой поступают в реальном времени. Такая обстановка противоречит привычному образу мыслей о традиционных базах данных, разработчики которых очень гордятся тем, что те обеспечивают идеальную точность, но по своим собственным часам. Рисунок 6.1 [1] представляет собой грубое изображение используемого исследователями алгоритма модели обработки потоковых данных.

Мы готовы объяснить принципы, по которым компоненты этого высокоуровневого представления потоковых данных (рис. 6.1) соответствуют конкретным компонентам полностью реализованного и функционального приложения по обработке потоковых данных.

Любой, кто попытается объяснить алгоритм обработки потоковых данных, поместит его на ярус модели конвейера потоковых данных (рис. 6.2), предназначенный для анализа. Именно там, в самом сердце этой системы, ее разработчики представляли себе использование алгоритма. Такое «сужение поля зрения» выглядит полезным для целей его понимания, но оно затуманивает общее видение, если мы хотим развить интуитивное понимание ситуаций, когда в качестве решения следует использовать фильтр Блума.

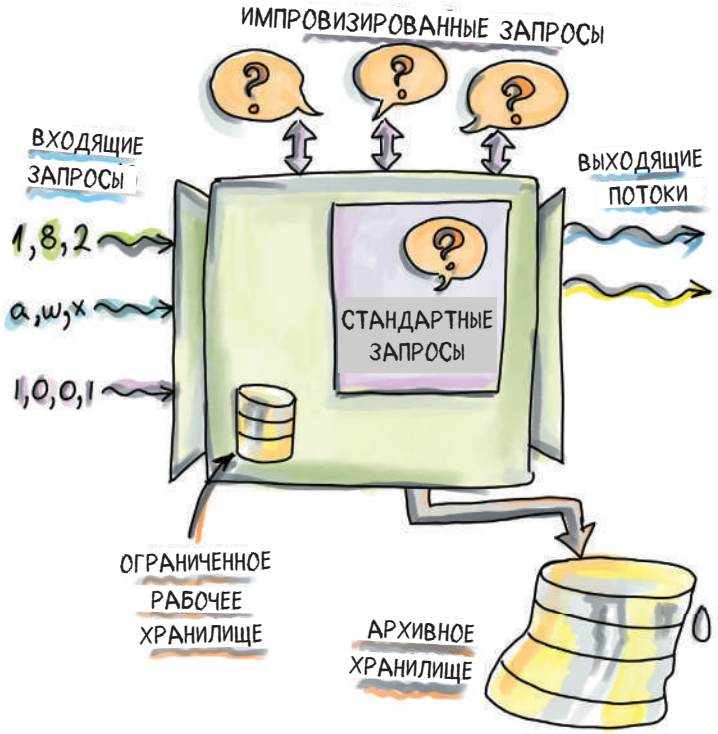


Рисунок 6.1 Модель обработки потоковых данных.
Модель обработки потоковых данных отличается от традиционной системы управления базами данных тем, что данные проходят через процессор и небольшой объем рабочей памяти, и они либо никогда не сохраняются, либо сохраняются в архивном хранилище, которое обычно слишком велико и медленно, чтобы его индексировать и проводить там поиск. Там можно найти нужные элементы, но мы не должны рассчитывать на то, что будем делать это часто и быстро. Весь реально-временной анализ выполняется «на лету». Существуют два типа запросов: стандартные (или неизменные) запросы, которые необходимо вычислять постоянно, и импровизированные[41] запросы, которые время от времени появляются неожиданно, и их содержимое контролируется извне

Допустим, нам нужно отобрать несколько кортежей данных из потока запросов, отправленных в облако, на котором размещена популярная веб-страница или служба (скажем, Google или Amazon). Вероятно, мы бы разработали алгоритм для работы с уникальным ИД каждого запроса. Эти запросы будут проходить ярусы *сбора данных* и *обработки очередей сообщений*, а фактический отбор случайных выборок из потока запросов будет происходить в *ярусе анализа* (рис. 6.2).

[41] Англ. ad hoc; син. экспромтный, разовый. – *Прим. перев.*

Рисунок 6.2 Показана общая модель конвейера обработки потоковых данных. Производители данных инициируют подключения, взаимодействуя с приложением розничного продавца

Иногда пользователи отправляют запрос, но не получают подтверждения о получении. Возможно, они выходят за пределы досягаемости своего сигнала Wi-Fi. Логика внутри устройства может отправлять идентичный запрос повторно. В итоге мы получаем два практически идентичных запроса. И теперь у нас возникает проблема, потому что мы не хотим, чтобы такой посторонний процесс влиял на взятие выборок. Если оставить дубликаты как есть, то наш алгоритм формирования выборки будет отбирать эти дубликаты во много раз чаще из-за наличия идентичных копий по сравнению с запросами, которые были получены только один раз. Помните, мы начинали с яруса анализа? Но, похоже, для использования нашей готовой версии алгоритма формирования выборки необходима какая-то предобработка потоковых данных перед ярусом анализа. В этой ситуации полезно сделать паузу и понаблюдать за нашим планом взятия выборок в системе в целом. Он превратился из простого, готового к подключению и использованию алгоритма, ограниченного ярусом анализа, в композитный алгоритм предобработки плюс формирования выборки, который распространяется на архитектуру потоковых данных еще шире (рис. 6.2). Необходимая дедупликация происходит на ярусе обработки очередей сообщений, следовательно, алгоритм формирования выборки возможен только в том случае, если можно будет выполнять дедупликацию достаточно быстро. Внезапно мы, возможно, придем к мысли о дедупликации, улучшенной за счет фильтра Блума, которую мы опишем позже в этой главе.

Несмотря на то что алгоритмы фильтра Блума, HyperLogLog и формирования выборки из потока предназначены для использования в рамках безопасного, контролируемого яруса анализа, при их применении они органично создают среду взаимозависимых задач, решаемых путем их применения в последовательном порядке либо в ансамбле.

Мы начали с теоретического взгляда, который использовался для построения этих алгоритмов (рис. 6.1), но сами по себе они стали слишком незамысловатыми, чтобы решать задачу взятия выборок из потока. Затем мы расширили поле зрения и увидели, что приложение по обработке потоковых данных является естественной средой обитания таких алгоритмов, с учетом всех его ярусов. Это единственный путь, которым новичок в данной области может распознать общие черты в разных областях применения, причем эти общие черты являются ключом к успешному развитию практических навыков по данной теме. В предыдущих главах мы уже встречали иллюстративные примеры использования каждого алгоритма / структуры данных, но с конвейером обработки потоковых данных (рис. 6.2) у нас есть шанс увидеть их в тесном сопоставлении, используемыми для разных целей, но способствующими решению одной и той же глобальной задачи.

Мы будем использовать рис. 6.2 для описания общей эволюции данных в рамках конвейера обработки потоковых данных. Данные консолидируются в централизованном центре обработки данных, который объединяет данные, возможно, прибывающие из разных географических районов. Здесь может происходить преобразование, обогащение и предобработка данных. Затем данные отправляются на серверы обработки очередей сообщений (собственное или серийное оборудование, собранное вокруг облачной службы), которые проверяют и, по возможности, восстанавливают структурную, временну́ю и, возможно, причинно-следственную согласованность данных. С помощью парадигмы обработки очередей сообщений этот ярус устанавливает и поддерживает баланс между скоростью приема данных от сборщиков и скоростью потребления данных со стороны *яруса анализа*. Такая балансировка может происходить из-за того, что ярус анализа требует больше вычислений по сравнению с тем, что приходится делать производителям при передаче данных на ярус сбора. А разбалансировка же может легко приводить к перенасыщенности данными или *сбрасыванию нагрузки*[42] (вынужденному отсеву необработанных кортежей) и, следовательно, к потере данных. Наконец, данные поступают на ярус анализа, где вычисляются и поддерживаются разные резюме (под)потоков данных, производится выборка из потоков и ответы на непрерывные и импровизированные запросы. Затем поток ответов на запросы с яруса анализа пересылается разным потребителям данных на «периферии» системы обработки потоковых данных, таким как приборные панели данных, приложение по реально-временной продаже рекламы или какое-то автоматизированное приложение по управлению промышленным производством. Уф-ф! Получилось немало, но мы поместили туда все приложение по об-

[42] Англ. load shedding. – *Прим. перев.*

работке потоковых данных. Потерпите – то, что воспоследует, будет менее перегружено.

Раздел 6.1 иллюстрирует органическую среду обитания, которую приложение по обработке потоковых данных создает для рассмотренных ранее алгоритмов. В приложении по обработке потоковых данных они обнаруживаются вполне естественным образом. Приведенные в этом разделе примеры использования должны помочь вам распознавать задачи, которые в контексте массивных данных, таких как конвейер обработки потоковых данных, хорошо подходят для решения, применяя наших предыдущих знакомых (фильтр Блума, набросок count-min) и наших будущих, о которых вы еще узнаете из следующих глав. Будем надеяться, что это поможет вам развить навык концентрации внимания на тех частях системы, которые являются решением узколокализованной задачи, и навык расширения поля зрения, до высоты птичьего полета, на распределенные приложения с интенсивным использованием данных, от их «источника» до «приемника». Для тех из вас, кто все еще неопытен в приложениях по обработке потоковых данных, это будет шанс безопасно увидеть «логово зверя».

В разделе 6.2 мы вводим нативные для потоков данных понятия, которые управляют устройством алгоритмов, и определяем присущие им ограничения, в соответствии с которыми такие алгоритмы конструируются. Образно говоря, мы сужаем поле зрения, сосредоточиваясь на частях рис. 6.1. Мы должны быть в состоянии распознавать эти ограничения как немутируемый признак процедуры генерирования данных, чтобы вырабатывать работающее в их рамках решение. Для достижения этой цели мы рекомендуем прочитать книгу Псалтиса, которая в сочетании с данной книгой создает всесторонний и мощный набор инструментов обработки потоковых данных.

В разделе 6.3 мы возвращаемся к теории вероятности, лежащей в основе взятия и оценивания выборок, поскольку в главе 7 знакомимся с алгоритмами формирования выборок из потоков. Если вы хотите научиться модифицировать изначальные алгоритмы и задавать адаптированные параметры в соответствии с вашей конкретной ситуацией, то, вероятно, вам следует проработать эту часть, поскольку если перед вами окажется крупный объем данных, даже незначительная поправка может привести к значительной экономии пространства/времени. В остальном же вы можете бегло пролистать этот материал, чтобы ваши глаза привыкли к обозначениям, потому что мы будем использовать их в главе 7.

6.1 Система обработки потоковых данных: метапример

На рис. 6.2 показана модель конвейера обработки потоковых данных. Следует учитывать, что на практике изображенные ярусы отличаются друг от друга не так четко. Как мы увидим, указанные ярусы нередко накладыва-

ются: некоторые части системы содержат задания, которые нельзя четко отнести к одному ярусу.

После примера со взятием выборок из запросов это не должно вызывать удивления. Конвейеры обработки потоковых данных содержат огромное число кортежей данных, которые пролетают по всей их длине. И как вы увидите, единственная разница заключается в компонентах кортежей данных, обрабатываемых нашими алгоритмами, то есть отправляемых, эмитируемых, вставляемых в очереди, транспортируемых, принимаемых и анализируемых компонентах кортежей данных.

Мы знаем, что наиболее общая модель пересылки данных по некоторой сети включает в себя по меньшей мере два компонента: метаданные и полезную нагрузку. Мы увидим, что в зависимости от места в конвейере обработки потоковых данных, в котором мы находимся, метаданные и полезная нагрузка могут менять свой подтекст. Это означает, что по ходу конвейера полезная нагрузка (запросы) иногда становится для кортежа данных накладными издержками, тогда как метаданные (уникальные идентификаторы запросов) становятся актуальными для анализа в зависимости от места внутри конвейера, в котором мы находимся в данный момент.

6.1.1 Соединение на основе фильтра Блума

Представьте себе крупного розничного продавца, который продает свои товары онлайн и в магазинах. Под этот профиль подходит Walmart или Whole Foods. У компании может возникнуть потребность в реально-временном (возможно, почти реально-временном) анализе функциональной ассоциации между шаблонами кликов по ее URL-адресам и данными о сделках купли-продажи. Возможно, они захотят оптимизировать свою стратегию ставок в реально-временных рекламных кампаниях (**http://mng.bz/g4pR**). В наши дни нет ничего необычного в том, что подобного рода данные хранятся в двух разных системах баз данных, создавая так называемое гибридное хранилище. Данные о продажах для компании более ценны; следовательно, они нередко находятся в параллельной базе данных на высокопроизводительных серверах или корпоративных складах данных (EDW)[43], тогда как для данных потока кликов может быть достаточно промышленной сети серверов, такой как распределенная файловая система Hadoop[44].

На настоящий момент мы будем исходить из того, что кортежи данных потока кликов прибывают и хранятся в распределенной файловой системе Hadoop (HDFS) и мы хотим соединять данные потока кликов и данные об онлайновых продажах. Для этого мы будем использовать столбец IP-адреса в качестве ключа соединения. Здесь, для краткости, мы абстрагируемся от необходимой временно́й близости, которую соединение должно учитывать. С онлайновой покупкой сопоставляются только те клики, которые

[43] Англ. enterprise data warehouse (EDW). – *Прим. перев.*

[44] Англ. Hadoop Distributed File System (HDFS). – *Прим. перев.*

расположены близко по времени к онлайновой покупке с того же IP-адреса. В данном случае нам нужно соединять покупки, сделанные с IP-адресов, с кликами, сделанными с тех же IP-адресов примерно в то время, когда покупка была зарегистрирована (см. рис. 6.3).

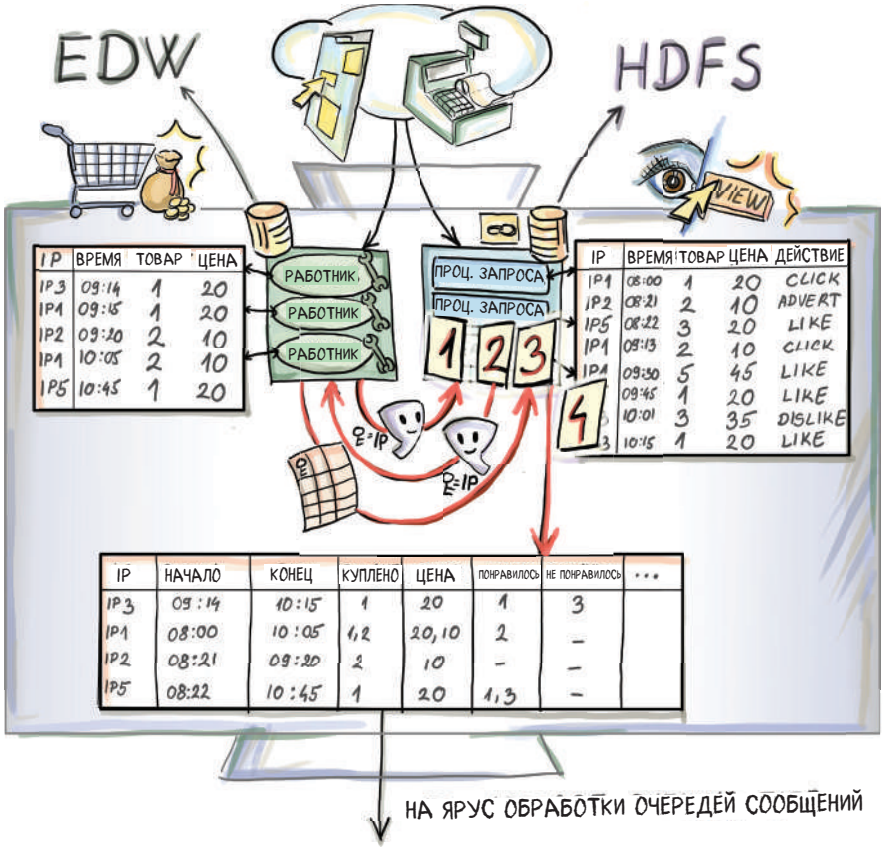


Рисунок 6.3 Обмен между EDW перед соединением, реализованный с помощью быстрой параллельной базы данных (слева) и HDFS (справа). Перед тем как данные о финансовых сделках купли-продажи отправляются со стороны EDW, обе системы хранения обмениваются фильтрами Блума для получения взаимного ключа соединения (IP-адреса). Каждая сторона использует фильтры Блума в качестве критерия, чтобы идентифицировать кортежи, которые будут участвовать в окончательном соединении. Затем HDFS может стасовать только необходимые данные между своими узлами и переместить минимальный объем данных на узел, который выполнит последующее соединение. EDW определит IP-адреса, которые не были указаны в HDFS, и отправит только те, которые будут участвовать в окончательном соединении. Благодаря этому данные о покупке обогащаются компонентом потока кликов с этого IP-адреса и могут в дальнейшем использоваться в качестве источника данных в гибридном конвейере потоковых данных

Можно допустить, что обе базы данных очень велики, но при этом сторона HDFS больше, что является правдоподобным допущением. Мы концентрируемся на минимизации размера таблиц, которые необходимо транслировать между этими двумя системами, чтобы реализовать желаемую операцию соединения. За счет этого будут экономиться пропускная способность и время, в особенности когда применяемые к таблицам локальные предикаты и/или проекции не сильно избирательны (мы приходим к таблицам, которые ненамного меньше всех данных, хранящихся в системах хранения).

Обычная стратегия в таком случае заключается сначала в создании блумовского фильтра (ФБ) ключа соединения каждой стороной. Давайте допустим, что окончательное соединение происходит на стороне HDFS; тогда глобальный фильтр Блума ($ФБ_{EDW}$) на стороне EDW, вычисленный для столбца IP-адреса, отправляется каждому процессору запросов HDFS[45] (самое время взглянуть на рис. 6.3). Здесь он используется как тип предиката (фильтра) для идентификации результирующей таблицы меньшего размера, которая будет участвовать в окончательном соединении. В случае если данные должны тасоваться между процессорами запросов HDFS, то перемещаться должны только данные с ключом соединения в $ФБ_{EDW}$ (с точностью до ложноположительной частоты $ФБ_{EDW}$). Затем сторона HDFS создает свой глобальный $ФБ_{EDW}$ и отправляет его стороне EDW, использующей его для дальнейшего сокращения числа подлежащих отправке строк. После того как все это будет сделано, сторона EDW отправляет результирующую таблицу своей исходной таблице после применения предикатов, проекций и $ФБ_{EDW}$. Благодаря такому двупутному использованию фильтров Блума по сети будут отправляться только те записи, которые участвуют в соединении, и будет исполняться только необходимая растасовка данных между процессорами запросов HDFS на стороне Hadoop.

Упражнение 1

Теперь давайте конкретизируем наше соединение на основе фильтра Блума[46]. Для ясности допустим, что покупка – это кортеж, содержащий время в миллисекундах (занимая 4 байта), IP-адрес (4 байта), список приобретенных товаров (их коды и т. д.) (64 Кб) и итоговую сумму (8 байт). Клики на стороне HDFS таковы: кортежи Spartan хранят время в миллисекундах (4 байта), IP-адрес (4 байта) и URI (64 Кб). Мы можем допустить, что покупки и клики «переплетаются» с постоянной частотой, что позволяет поддерживать соотношение кликов к покупкам постоянным; в нашем случае 45 кликов/покупка. Все люди кликают, но не все что-то покупают.

Допустим, что следующее соединение происходит после того, как будет совершен 1 млн несовпадающих покупок. Каков был бы размер переданных данных, который мы бы сохранили, задействовав здесь фильтр Блума с ложноположительной частотой 0.1 %?

[45] Англ. HDFS query processor (HQP). – *Прим. перев.*

[46] Англ. Bloom-join. – *Прим. перев.*

Давайте посмотрим, где в конвейере обработки потоковых данных все это только что произошло. Такие соединения на основе фильтра Блума могут устанавливаться где-нибудь на ярусе сбора в рамках показанной на рис. 6.2 схемы. Это шаг обогащения/предобработки данных, служащий для генерирования данных, которые подходят для ответа на интересующий вопрос из области деятельности компании. В созданной на стороне HDFS результирующей таблице сохраняются пары темпорально близких кликов и покупок с одного и того же IP-адреса (рис. 6.3) или, точнее, все их поля. Затем этот процесс может служить своего рода непрерывным производителем для фреймворка обработки потоков данных (например, Apache Kafka). Пары (строки в результирующей таблице; рис. 6.3) затем передаются для обработки в очередях, анализа и использования, возможно (почти) в реально-временных индивидуализированных рекламных кампаниях.

6.1.2 Дедупликация

Из-за высокой частоты приема генерируемых производителями данных и, как следствие, большого потока (возможно) предобработанных данных по конвейеру каждый показанный на рис. 6.2 ярус состоит из большого числа соединенных через сеть узлов (машин). Эти вычислительные узлы реализуют задание своего яруса в параллельном режиме настолько быстро, насколько это возможно. Ярус обработки очередей сообщений предназначен для предотвращения сбоев, потери данных (из-за разной скорости генерирования и потребления данных), реализации дедупликации, если это необходимо, и т. д. Указанные механизмы обеспечения безопасности по своей сути влекут за собой некие шаги урегулирования между узлами, отслеживающие данные, которые пропущены на ярус анализа, и все последующее.

Узлы на ярусе обработки очереди сообщений обычно называются *брокерами*, и в дополнение к поддержанию согласованности очередей сообщений они выполняют другие шаги предобработки. Как в примере с запросами, взаимодействуя с веб-сайтом розничного продавца, потребитель может потерять беспроводную связь или войти в лифт и пропустить подтверждение со стороны сервера об отправленном платеже, и тогда мобильное приложение попытается снова отправить тот же платежный запрос. Это привело бы к повторному платежу. Ни потребителям, ни корпоративным системам повторяющиеся платежи подобного рода не нужны. Некоторые системы, в особенности системы электронной коммерции, имеют механизмы дедупликации, позволяющие избегать такого дублирования. Процент дубликатов в реалистичных сценариях не слишком велик (возможно, 1 %), но в системе, которая регистрирует миллиарды событий, они могут приводить к неэффективности, отражающейся в значительной потере прибыли.

В предыдущей главе мы уже решали одну задачу дедупликации. Помните пример с хранением больших файлов и службами резервного копирова-

ния? Эта реальная ситуация, в которой дублируется малая часть сообщений, хорошо подходит для еще одного применения фильтров Блума с отведением узлов «перехвата» в потоковом приложении, возможно, построенном на потоках Apache Kafka. Указанные узлы-работники подключены к высокоскоростным базам данных. Помимо долговременного хранения всех сообщений (или только их части) с целью облегчения возможного отката в случае потери данных, они хранят фильтр Блума для идентификаторов всех сохраняемых сообщений. Каждое прибывающее сообщение проверяется фильтром, и если сообщается о его наличии (с учетом ложноположительной частоты фильтра Блума), узлы-работники его отбрасывают. Дедуплицированные потоки сообщений выстраиваются в очередь, возможно, в выходных топиках Kafka (**https://segment.com/blog/exactly-once-delivery/**), откуда они затем могут перенаправляться узлом балансировки нагрузки нескольким брокерам, ведущим на ярус анализа (рис. 6.4).

Рисунок 6.4 Промежуточные узлы, подключенные к быстрым базам данных, реализуют дедупликацию, чтобы удалять повторяющиеся экземпляры сообщений в конвейере обработки потоковых данных. Каждый узел содержит фильтр Блума для сохраненных им сообщений и по прибытии следующего сообщения проверяет хеш идентификатора сообщения со своим фильтром Блума. Если фильтр Блума сообщает о том, что сообщение уже существует, то данные этого сообщения отбрасываются; в противном случае сообщение сохраняется и передается на ярус обработки очередей сообщений

6.1.3 Балансировка нагрузки и отслеживание сетевого трафика

Как и в любой распределенной вычислительной системе, балансировка нагрузки между брокерами в приложениях по обработке потоковых данных имеет первостепенное значение. Предоставление брокерам несбалансиро-

ванных нагрузок может приводить к тому, что один из них будет получать непропорционально больше подключений, запросов и т. д. Учитывая, что служба работает не быстрее самого медленного брокера, это может приводить к большим сквозным задержкам и прерыванию работы реально-временных приложений. В практическом плане реально-временное обнаружение чрезмерно используемых ресурсов в сети – это классическая задача из области сетевого трафика / обработки распределенных очередей, и она сводится к мгновенному обнаружению выбросов/аномалий. Такие выбросы проявляются в виде шаблонов перегруженных пакетных потоков и типичны для атак типа DoS (отказ в обслуживании) на серверы. Современные стратегии защиты опираются на статистические методы их обнаружения в реальном времени.

Один из таких классов алгоритмических решений указанной проблемы в сетевом трафике основан на мониторинге заголовков пакетов в сети. Для этого алгоритму требуется лишь базовая информация о каждом потоке (FL =[IP-адрес источника, порт источника, IP-адрес местоназначения, порт местоназначения, протокол]). Мониторинг этой формы потока запросов позволяет идентифицировать малое число потоков, которые составляют бóльшую часть сетевого трафика, то есть так называемых *тяжеловесов*. На практике мы хотим обнаруживать некое число из них, скорость которых (число пакетов/запросов (байтов) в единицу времени) превышает некий порог.

В главе 5 мы рассмотрели решение задачи об обнаружении червей в обычной сети, основанное на алгоритме HyperLogLog. Еще одним решением могло бы быть использование наброска count-min, который подсчитывает совокупный размер потока путем прибавления размера каждого пакета, отправленного через этот поток (поток здесь представляет собой пару определителей источник–местоназначение). Ключи, хешированные в набросок count-min, являются заголовками пакетов, а счетчик увеличивается на размер текущего пакета.

В более конкретном случае брокера в рамках приложения по обработке потоковых данных счетчик будет увеличиваться на число запросов, поставленных в очередь одного и того же брокера. Это может происходить из-за кратковременного увеличения числа производителей (в данном контексте это называется *внезапными всплесками*). Приложение по измерению трафика или балансировке нагрузки затем оценивает минимальные числа появлений: «тяжеловесность» каждого потока или «занятость» каждого брокера. Они будут периодически делиться на продолжительность периода измерения. В результате мы получаем скорости потока / постановки в очередь. Сетевой администратор или инженер конвейера данных затем отличает хорошие потоки от плохих, применяя некий порог, основанный на их варианте использования. После выявления виновных они применяют некую стратегию сдерживания.

Благодаря алгоритму наброска count-min потоки ниже порогового значения по определению не являются вредоносными, тогда как из выяв-

ленных некоторые могут быть ложноположительными из-за присущего наброску count-min завышения частоты. Затем можно быстро проверить точный размер/скорость небольшого числа потоков, превышающих пороговое значение, и удалить ложноположительные, оставив в наборе только истинных виновников.

Упражнение 2

Давайте вернемся к варианту использования с дедупликацией для фильтра Блума в системе электронной коммерции. Допустим, ваш вышестоящий руководитель дал вам задание проверить и доработать решение, начатое инженером, который недавно покинул компанию. Вы понимаете логику решения, поскольку оно позволяет экономить время и не взимать с потребителя двойную плату. Тем не менее за это приходится платить. Что происходит, когда фильтр Блума выдает ложноположительный результат по конкретному запросу, содержащему информацию о лайке, отправленном комментарии или кликнутом объявлении? Что делать, если клиентское приложение отправило платеж? Есть ли там что-нибудь, на что следует обратить внимание? Что происходит, когда фильтр Блума возвращает ложноположительный результат для фактически осуществленного платежа?

Обобщенное приложение мониторинга сетевого трафика можно найти в облачной службе, на которой работает ваше приложение по обработке потоковых данных, тогда как вариант использования с балансировкой нагрузки относится к внутренней работе самого потокового приложения (см. рис. 6.5).

Цель этого краткого и поверхностного экскурса в реалистичную архитектуру обработки потоковых данных состояла в том, чтобы дать вам некоторое представление о вездесущности рассмотренных ранее алгоритмов и структур данных в современных приложениях по обработке потоковых данных. Кроме того, надеемся, что помогли вам увидеть все сопутствующие проблемы, которые необходимо решать в такой по своей сути распределенной вычислительной среде. Будем надеяться, что к настоящему времени мы соткали для вас «*ковер, который объединит комнату*».

Вторая цель состояла в том, чтобы соотнести уровень абстракции, необходимый для разработки алгоритмов обработки потоковых данных (рис. 6.1), с уровнем, необходимым для построения реалистичного приложения/конвейера обработки потоковых данных (рис. 6.2). Мы знаем, что первый появляется в нескольких местах второго и что потоковые данные естественным образом склеивают их вместе, и каждый становится видимым на разных «уровнях детализации изображения».

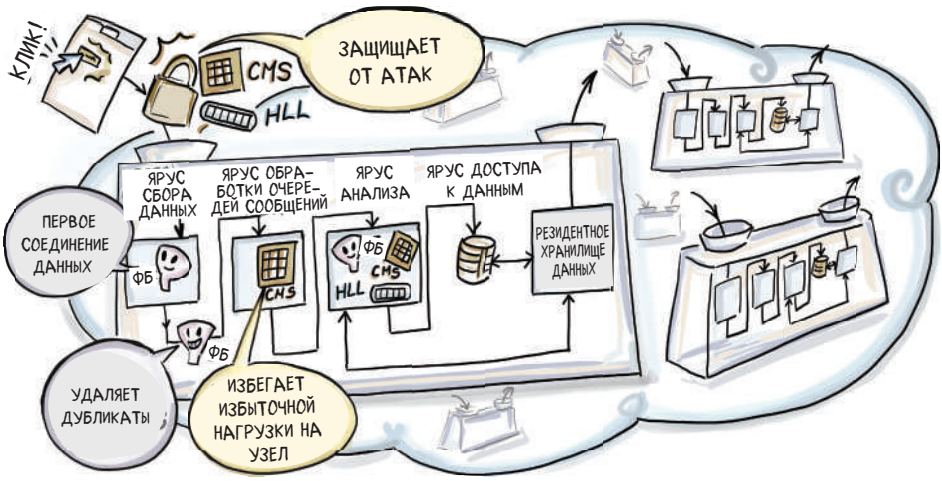


Рисунок 6.5 Архитектура облачных вычислений, обслуживающая разные клиентские конвейеры данных. Приложение мониторинга сетевого трафика, установленное для отслеживания обмена между производителями данных (или любых попыток обмена, исходящих из-за пределов облака), реализовано с помощью наброска count-min. Набросок count-min идентифицирует сетевые потоки, которые проявляют скорости, превышающие определенный заранее установленный порог, и действует соответствующим образом. Схожие задачи балансировки нагрузки между брокерами яруса обработки очередей сообщений решаются аналогичным образом применением еще одного наброска count-min в узле балансировки нагрузки на ярусе обработки очередей сообщений. Обратите внимание, что оба этих наброска count-min оперируют на заголовках пакетов/сообщений и что полезная нагрузка пакетов/сообщений, а именно данные о производимых пользователем кликах, не анализируется до тех пор, пока они не достигнут яруса анализа. Там вычисляются другие фильтры Блума, массивы HyperLogLog, процедуры взятия выборок или иные резюме/сводки

6.2 Практические ограничения и понятия потоков данных

Далее мы представим несколько понятий вычислений и обработки потоковых данных, которые должны соблюдаться разработчиками алгоритмов обработки потоковых данных и на основе которых алгоритмы оцениваются.

6.2.1 В реальном времени

Задание по конструированию приложения по обработке потоковых данных превращает наше представление о времени из философского занятия в очень практичное упражнение по ведению учета времени. Прежде всего приходит в голову и, судя по некоторым постам на форумах по

анализу данных, интересует умы других людей вопрос о том, существует ли реально-временная аналитика вообще. Любая малозаметная ссылка на потоковые данные сообщит, что данные в потоке прибывают непрерывно (возможно, от многочисленных производителей) с такой скоростью, что их сохранение и прохождение более одного раза по каждому кортежу данных невозможно. В некоторых прикладных областях, таких как анализ потоков финансовых данных с целью принятия торговых решений, наличие этой нереалистичной опции хранения и опрашивания всей истории расценивается как бесполезное, поскольку решения будут зависеть от данных только за последнюю неделю, возможно, даже за последнюю минуту. Следовательно, вполне разумно, что типичные требования к алгоритмам обработки потоковых данных заключаются в том, чтобы они работали за *один проход* за *малое время* и в *малом пространстве*.

Давайте вернемся к вопросу о существовании реально-временной аналитики. Даже если наши алгоритмы построены в соответствии с этими требованиями, вычисления требуют времени (не говоря уже о безопасности, коммуникации, планировании и балансировке нагрузки – все это является частью типичного облачного приложения по обработке потоковых данных). Строго говоря, единственные данные, которые действительно являются реально-временными, поступают в виде сенсорной стимуляции от событий, непосредственными свидетелями которых мы являемся. Если это звучит как занудство, то вы, вероятно, правы; было бы трудно привести убедительные аргументы в пользу обратного, но, пожалуйста, потерпите. Есть и хорошие новости: все это разрешимо. Давайте согласимся, что реальное время и понятие (почти) реально-временной аналитики определяются новейшим технологическим решением конкретной бизнес-задачи обработки потоковых данных. Если оно дает результаты и способствует принятию решений, которые поддерживают конкурентоспособность компании, то мы бы не сильно ошиблись, назвав его реально-временным. Другими словами, пользователи/потребители имеют право последнего слова в том, что для них является (почти) реально-временным, даже когда они переоценивают или недооценивают свои потребности.

Если задуматься, то, становясь свидетелями реальных событий в нашей жизни, мы все испытываем одинаковую задержку в части нашей сенсорной и когнитивной оценки происходящих на наших глазах событий. Вот почему мы так легко соглашаемся с понятием реального времени; мы все одинаково «опаздываем». Теперь, когда мы разрешили эту дилемму, возможно, невероятно важную, можно продолжить обсуждение вопроса о том, что подразумевается под малым временем и малым пространством.

6.2.2 Малое время и малое пространство

Исходя из наших соображений, малое пространство будет определяться по отношению к доступной рабочей памяти, изображенной на рис. 6.1, как *ограниченное рабочее хранилище*. В нем должны храниться любые дан-

ные, необходимые механизму обработки потоковых запросов для своевременного ответа на импровизированные или непрерывные запросы. Вот где должно уместиться все разнообразие резюме данных, фильтры Блума, массивы HyperLogLog, результаты алгоритмов формирования выборок (буферизации) на потоках данных, гистограммы потока и т. д.

Малое время относится ко времени обработки алгоритмом каждого нового прибытия, а также ко времени, необходимому для выдачи ответа на конкретный запрос (*время обработки запроса*). Малое время обычно означает сублинейное, как правило, полилогарифмическое по N, где N – это длина подпотока, который можно уместить в ограниченной рабочей памяти.

6.2.3 Сдвиги в концепциях и дрейфы концепций

Поскольку поток данных непрерывен в своей временной составляющей, механизм генерирования данных может демонстрировать и будет демонстрировать прерывистость.

Давайте возьмем приложение по реально-временной обработке потоковых данных в Facebook, задачей которого является предупреждение пользователей о непосредственной локальной угрозе из-за вооруженного конфликта, стихийного бедствия или аналогичной неминуемой опасности, которая затрагивает большой географический район. Далее допустим, что приложение ведет подсчет появлений слов в подпотоках пользовательских объявлений на платформе. Подпотоки могут задаваться с использованием неких географических критериев, которые делают предупреждения о таких событиях актуальными для людей в этом районе. Любое решение нуждалось бы в реализации логики вычисления частоты (число появлений, деленное на продолжительность периода регистрации) появления тех или иных зарезервированных слов. Неминуемая локальная угроза человеческим жизням в этом районе привела бы к внезапному увеличению числа сообщений, связанных с таким бедствием. Потоковые алгоритмы должны уметь обнаруживать такие резкие изменения в потоке данных, именуемые в технической литературе как *сдвиги в концепциях*[47].

Поведение потока данных, сходное со сдвигами в концепциях, но проявляющееся в течение более длительного периода времени и характеризующееся не столько резкими, сколько постепенными изменениями, называется *дрейфом концепций*[48]. Задача его обнаружения менее тривиальна по сравнению со сдвигами в концепциях, и это было давней темой научных изысканий. (Подробный обзор имеющихся методов см. в статье Себастьяо (Sebastiao) и Гаммы (Gamma) [2].)

Оба понятия тесно связаны с понятием оконного потока данных, одного из механизмов учета новизны/недавности потока данных.

---

[47] Англ. concept shift. – *Прим. перев.*

[48] Англ. concept drift. – *Прим. перев.*

6.2.4 Модель скользящего окна

Поток данных теоретически бесконечен. Предполагается, что потоковая обработка начинается в некий четко определенный момент времени t_0 и что в любой момент времени t на запросы отвечают, принимая во внимание все кортежи, наблюдавшиеся между t_0 и t. Эта модель потока данных называется *реперным потоком*[49].

Будем надеяться, к настоящему времени вы уже почувствовали, что это невозможно, поскольку мы знаем, что не можем хранить поток в рабочей памяти и не можем выполнять многократные проходы по его данным – по меньшей мере несвоевременно, чтобы расценивать ответ как актуальный для практических целей. К счастью, словосочетание «принимая во внимание» означает, что резюме, которые мы извлекаем из «галопирующих» данных, являются функцией *всех* кортежей, появившихся на данный момент. Следовательно, даже старые кортежи данных, которые появились давным-давно и из-за ограниченной рабочей памяти были отброшены или помещены в архивное хранилище (рис. 6.1), имеют тот же вес, что и новые.

В некоторых приложениях, таких как потоки финансовых данных, использование старых данных, чтобы управлять текущими ответами на запросы, в лучшем случае не приносит никакой пользы, а в худшем – влечет за собой ответственность. Сталкиваясь со сдвигами в концепциях и дрейфами концепций, запросы к реперным потокам подвержены инерции и слишком медленно «реагируют» на изменения в концепциях. Для этой цели были введены различные механизмы затухания во времени, которые связывают возраст кортежа данных и вес, с которым он влияет на ответы на запросы.

Одной из наиболее заметных является *модель скользящего окна*, которая учитывает только определенное число (окно) самых последних прибывших кортежей данных. Кортежи данных за пределами окна автоматически удаляются из анализа, или им назначается нулевой вес. Следует учитывать, что теоретически они все еще могут находиться в ограниченной рабочей памяти, если скользящее окно конструктивно меньше по величине, чем то, что можно уместить в пространстве, доступном для однопроходных вычислений.

Перемещение скользящего окна может основываться как на времени, так и на числе элементов. В окнах на основе времени отображаются любые кортежи данных, прибывшие за последние W единиц времени, тогда как перемещение в окнах на основе числа элементов регулируется поддержанием постоянного числа W элементов в окне (далее смысл W станет яснее). Обе модели показаны на рис. 6.6 и 6.7 на примере обобщенного потока данных для трех самых последних перемещений скользящего окна.

---

[49] Англ. landmark stream. – *Прим. перев.*

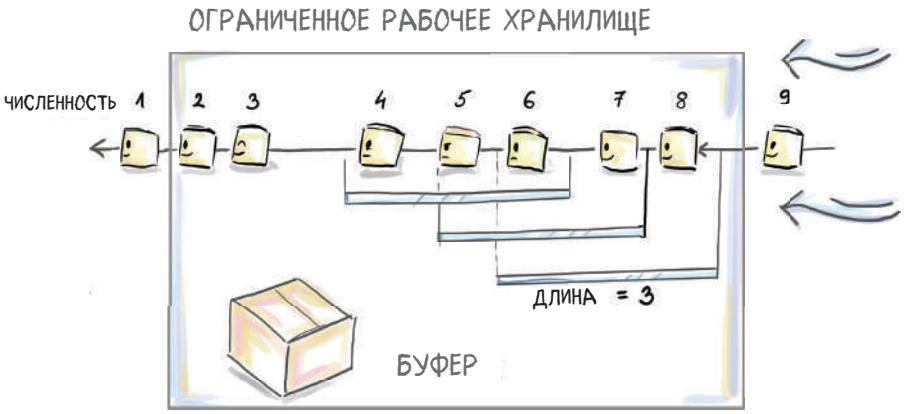


Рисунок 6.6 Последние три перемещения скользящего окна, основанного на числе элементов. Обратите внимание, что длина окна *W* – это не обязательно все, что можно вместить в рабочее хранилище, но это максимум, который можно охватить. Следовательно, в приложениях, где мы хотим, чтобы на анализ потока данных влияло «как можно больше истории», мы увеличиваем длину окна до «всего, что можно вместить». Следует учитывать, что помимо подпотока, который нужен для работы в однопроходном режиме, еще нужно резервировать пространство для резюме данных и вычислений, необходимых для их сборки и обновления. На рисунке на это указывает буферное пространство.

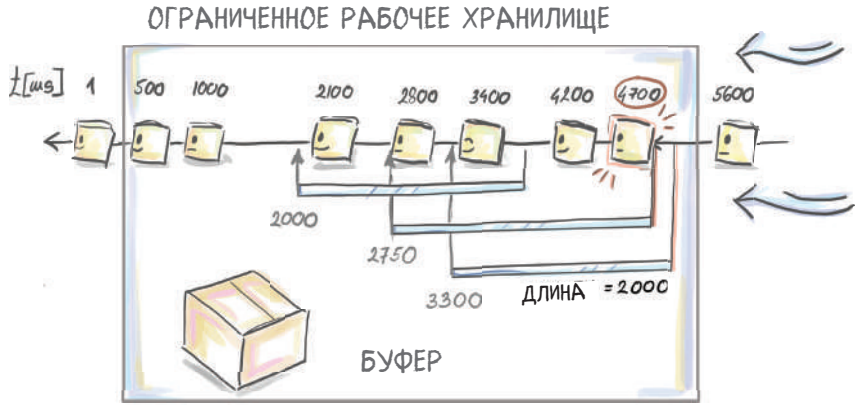


Рисунок 6.7 Здесь мы видим скользящее окно, основанное на времени, длиной W = 2000 мс и его три последних (значимых с точки зрения времени прибытия кортежей данных) перемещения. Мы запустили поток данных в момент времени 0 мс, и в момент времени 1 мс прибыл первый кортеж данных. Затем во временны́х точках 500 мс и 1000 мс появились еще два. Здесь мы наблюдаем поток через 5.500 мс. Указаны три последних перемещения окна, которые изменили его содержимое: с 1.400 до 1.401 мс (кортеж данных прибыл в точке 3.400 мс и вошел в окно), с 2.700 мс до 2.701 мс (обратите внимание, что в результате этого перемещения в окне появилось 4 кортежа данных) и с 2.800 мс до 2.801 мс (кортеж данных в точке 2.800 мс отбрасывается)

Приведенный выше список ограничений, связанных с потоками данных, не является исчерпывающим, но теперь мы можем успешно пройти следующие пару глав, не оставив невыясненными вопросы ни по одному из изучаемых алгоритмов.

Следующий далее раздел служит в качестве обзора теории формирования выборок. Он должен облегчить понимание нюансов главы 7 читателям, более любознательным в техническом плане.

6.3 Немного математики: формирование и оценивание выборок

Идея взятия выборок возникла из-за невозможности отвечать на вопросы о логистически непрослеживаемых крупных наборах данных. Например, если вы и кто-то, с кем у вас общий IP-адрес, вместе пользуетесь интернетом, то ваши запросы принимаются с одного и того же IP-адреса, но при использовании разных браузеров будет оставляться разный HTTP-отпечаток. Нас могло бы заинтересовать среднее число HTTP-отпечатков по всем возможным IP-адресам. Получить правильный ответ нет никаких шансов, но даже приближенная оценка будет больше, чем то, что мы знали, когда эта идея пришла нам в голову. Для получения приближенной оценки мы извлекаем выборки из пространства IP-адресов.

Первое зарегистрированное использование этой идеи произошло в 1786 году в попытке Пьера Симона Лапласа оценить численность населения Франции. Естественно, оценка была неверной, но заслуга Лапласа, сделавшего выборку мощным инструментом, состояла в том, что он обеспечил верхнюю границу вероятности того, что его оценка, основанная на выборке, была слишком далека от правильного ответа. Предоставление оценки не было чем-то новым, но добавление к ней структурированного метода измерения и, возможно, ограничения неопределенности оценки по сравнению с истиной стало новой и, к счастью, получившей широкое применение идеей.

Для того чтобы определить процесс формирования выборки на практике, нужны два компонента: (конечная) *популяция*, которая нас интересует (или какой-то ее аспект), и какой-то способ отбора случайных членов из этой популяции, окончательный, «материализованный» артефакт, именуемый *выборкой*. Вы обнаружите, что слово *выборка* часто используется для обозначения отдельных элементов/наблюдений/образцов, составляющих выборку. Мы находим, что это сбивает с толку. Поэтому мы будем называть отдельные элементы выборки наблюдениями/членами/образцами, тогда как их совокупность будет составлять выборку[50]. В зависимости от того, каким образом мы получаем эти случайные члены, выборка может быть *репрезентативной* (несмещенной) или *смещенной* по отношению к популяции.

[50] Помимо этого, в переводе процесс *sampling* переводится как формирование выборки или взятие выборки взаимозаменяемо, а результат этого процесса *sample* – как выборка. – *Прим. перев.*

Если каждое подмножество из *k* элементов из популяции имеет одинаковые шансы стать окончательной выборкой размера *k*, то процесс взятия выборки является для популяции репрезентативным. В нашем IP-пространстве это означает, что каждое подмножество из *k* IP-адресов имеет равную вероятность быть взятым в нашу выборку. Это также означает, что каждый отдельный член популяции имеет одинаковые шансы быть взятым, независимо от того, что еще известно об этом члене популяции. Если мы можем это гарантировать, то у нас получается *простая случайная выборка*[51].

Давайте подумаем о том, как это соотносится с примером IP-пространства. На самом деле частота запросов через IP-адреса различна. *Обходчики*[52] (скрипты, автоматически посещающие и каталогизирующие веб-сайты) отправляют запросы, возможно, чаще, чем человек, просматривающий веб-сайты в течение одного сеанса. Если бы мы брали IP-адреса по схеме «взять случайный запрос и добавить его IP-адрес в выборку», то у обходчиков было бы больше шансов попасть в нашу выборку. После того как мы решим, что выборка достаточно велика, у нас окажется набор IP-адресов, чрезмерно представленный адресами, относящимися к обходчикам, по сравнению с теми, которые использовались людьми. Это привело бы к неправильному представлению об истинном среднем числе отпечатков в расчете на IP-адрес. Если мы обеспечиваем наличие равных шансов у каждого IP-адреса попасть в выборку, независимо от того, что еще известно об этом IP-адресе, то этого не произойдет. Именно этого мы обычно и добиваемся, поскольку это позволяет нам, подобно Лапласу, полагать, что у нас есть хорошая оценка, и вычислять границы неопределенности данной оценки. Наша изначальная стратегия формирования выборки IP-адресов напрямую из всех полученных запросов – это *стратегия формирования смещенной выборки*[53].

6.3.1 Стратегия формирования смещенной выборки

Мы будем использовать гипотетическую популяцию счетов, зарегистрированных крупным розничным продавцом, чтобы описать причины, которые делают процесс взятия выборки смещенным. На данный момент не имеет никакого значения, каким образом эти данные у нас появились. Они могут находиться в базе данных либо быть получены из потока данных. Давайте обозначим через *N* число индивидуальных покупок. Мы хотим знать долю покупок, совершенных по карте лояльности. Если наш отдел маркетинга планирует и заранее сообщает о том, что эта информация им потребуется, то сохранять и обновлять ее при каждой новой покупке несложно, но если они внезапно захотят ее иметь (импровизированный запрос), то мы сможем получить хорошую оценку за счет поддержания выборки и вернуть выборочную долю покупок по карте лояльности.

---

[51] Англ. simple random sample (SRS). – *Прим. перев.*

[52] Англ. Crawler. – *Прим. перев.*

[53] Англ. biased sampling strategy. – *Прим. перев.*

Покупки подразделены на покупки по карте лояльности и не по карте лояльности. Истинное значение (l) доли покупок по карте лояльности (L) очевидно; это число членов, полученное в результате деления L на N. В нашей выборке размера k это транслируется в наличие i элементов с L и j элементов без L, при этом $i + j = k$.

В таком контексте мы опишем два процесса отбора. Первый – это взятие несмещенной выборки, а второй – взятие смещенной выборки на популяции зарегистрированных покупок. Мы будем использовать пример с $N = 10$, $k = 5$ и $p_L = 4/5$. Пусть популяция представлена множеством

$$\{x_{1L}, x_{2\bar{L}}, x_{3L}, x_{4L}, x_{5\bar{L}}, x_{6L}, x_{7L}, x_{8L}, x_{9L}, x_{10L}, \}.$$

В индексе каждого элемента можно считывать некую форму идентификатора и индикатор наличия/отсутствия характеристики L (в данном случае элементы с идентификаторами 2 и 5 покупаются без предъявления карты лояльности). Для любого 5-элементного подмножества из этих 10 элементов можно привести простой комбинационный аргумент о вероятности выбора этого конкретного подмножества. Вводные лекции по теории вероятностей либо на курсе дискретной математики в первые несколько недель учат, что существует (10 по 5) = 252 разных 5-элементных подмножества, и если предполагается, что каждое из них равновероятно, то вероятность выбора какого-либо конкретного из них составляет 1/252. Теперь представьте, что у нас 252-гранный кубик, показывающий цифры 1, 2, ..., 252, и мы (любым произвольным способом) перечисляем все 252 5-элементных подмножества. Бросание этого кубика и выбор подмножества, указанного на боковой стороне кубика, после его приземления и есть стратегия формирования репрезентативной выборки.

При этом полезно знать вероятность отбора в выборку каждого x_i, $i = 1, 2, ..., 10$. В данном процессе отбора она составляет 0.5 для любого из них, независимо от использования карты лояльности.

Любое пятиэлементное подмножество (наша выборка), которое можно себе представить, принадлежит к одному из трех типов. Выборка имеет тип 5, когда окажется, что все пять покупок были совершены по карте лояльности. Тип 4 состоит из одной покупки, совершенной не по карте лояльности, и четырех покупок, совершенных по данной карте. И последний – тип 3, в котором две покупки совершены не по карте лояльности и три покупки по этой карте. Оценки истинной вероятности $p_L = 4/5$ для этих трех типов выборки равны 1, 4/5 и 3/5 в указанном порядке.

Давайте допустим другую стратегию формирования выборки: сначала мы бросаем трехгранный кубик со смещением (на каждой стороне указано число покупок по карте лояльности; следовательно, 3, 4, 5), чтобы решить, какой тип выборки мы возьмем: 5, 4 либо 3. Затем проникаем в подраздел L и подраздел $\bar{L}$ по отдельности и берем столько покупок, сколько определено первым смещенным кубиком. Например, если разыграно 4, то мы знаем, что нужно взять 1 из подраздела $\bar{L}$ и 4 из подраздела L. Для взятия фактических покупок мы будем использовать описанную выше

стратегию формирования репрезентативной выборки. На этот раз для второй стадии понадобятся три пары кубиков. Нужны именно пары, потому что мы должны репрезентативно отбирать из подраздела $\overline{L}$ и подраздела L. А три пары нужны, потому что в зависимости от типа выборки, который был разыгран с помощью первого смещенного кубика, мы будем брать разное число покупок из подраздела $\overline{L}$ и подраздела L. Это приводит к (8 по 5)-гранному и (2 по 0)-гранному кубикам для выборки типа 5, (8 по 4)-гранному и (2 по 1)-гранному кубикам для выборки типа 4 и (8 по 3)-гранному и (2 по 2)-гранному кубикам для выборки типа 3. Если вы вспомните методы подсчета, то заметите, что первая и третья пары – это два фактически одинаковых кубика; следовательно, нужно всего четыре дополнительных кубика.

Давайте посмотрим на вероятность выбора пятиэлементного подмножества конкретного типа 5, 4 либо 3. Допустим, что грани, соответствующие типам выборок 5, 4 и 3, встречаются с вероятностями соответственно 2/5, 2/5 и 1/5 (здесь вступает в игру смещение первого кубика). Для выборки типа 5 мы имеем

$$\frac{2}{5} \times \frac{1}{\binom{8}{5}} \times \frac{1}{\binom{2}{0}} = \frac{1}{140}.$$

Для выборки типа 4 получаем

$$\frac{2}{5} \times \frac{1}{\binom{8}{4}} \times \frac{1}{\binom{2}{1}} = \frac{1}{350}.$$

Для выборки типа 3 получаем

$$\frac{2}{5} \times \frac{1}{\binom{8}{3}} \times \frac{1}{\binom{2}{2}} = \frac{1}{280}.$$

Следовательно, легко увидеть, что эта стратегия формирования выборки не является репрезентативной, или что она смещена, поскольку не все пятиэлементные подмножества будут «материализованы» в выборке с равной вероятностью. То же самое касается числа выданных запросов, привносящих смещение в нашу стратегию формирования выборки из запросов; смещенный кубик позволял выборкам с бóльшим числом покупок по карте лояльности быть более вероятными, чем с меньшим числом покупок по карте лояльности. Давайте теперь посмотрим, как это влияет на вероятность выбора одного-единственного наблюдения. Вспомните, что в стратегии формирования репрезентативной выборки вероятность попадания в выборку *любого* отдельного x_i составляла 0.5, независимо от признака L. В стратегии формирования смещенной выборки мы не ожидаем, что она будет той же самой, поскольку смещенный кубик «предпочитает» выборки с бóльшим числом покупок по карте лояльности. Это смещение будет «просачиваться» до уровня единичного наблюдения и «склонять» его шансы попасть в окон-

чательную выборку в зависимости от его признака L. В этом случае нужно рассчитать вероятности выбора отдельно для элементов с L и элементов без него. Мы оставляем определение точной вероятности выбора одного x_{iL} и $x_{i\bar{L}}$ в качестве упражнения и здесь просто указываем на то, что для каждого x_{iL} эта вероятность равна 0.525, тогда как для $x_{i\bar{L}}$ она равна 0.4.

Упражнение 3

Вероятности выбора[54] для $x_{iA}, A \in \{L, \bar{L}\}$ в случае стратегии формирования смещенной выборки можно вычислить посредством метода итеративного ожидания. Для каждого $A \in \{L, \bar{L}\}$ отдельно рассчитывается вероятность того, что x_{iA} будет отобран в выборку с учетом показаний смещенного кубика. Выполните все шаги, чтобы получить тот же ответ, что и для $x_{iA}, A \in \{L, \bar{L}\}$.

Упражнение 4

Какие вероятности трех граней смещенного кубика соответствуют стратегии формирования репрезентативной выборки с линией, равной 0.5 на рис. 6.8? Мы могли бы использовать вторую стратегию формирования смещенной выборки со специально подобранными весами вероятностей для каждой грани и эмулировать стратегию формирования репрезентативной выборки.

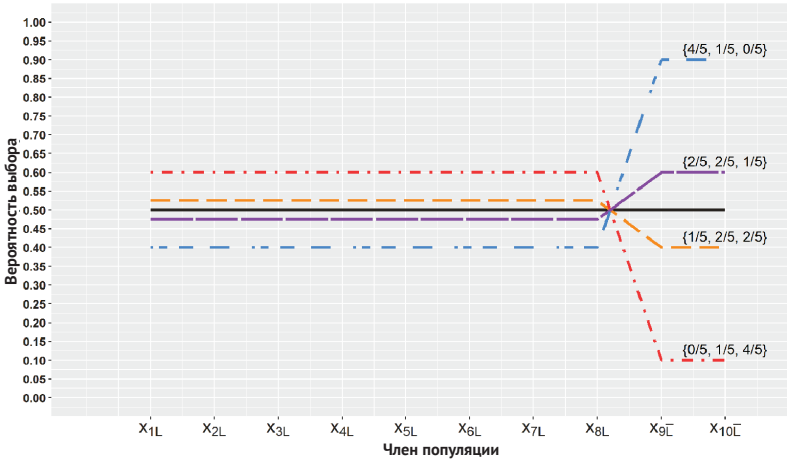


Рисунок 6.8 Вероятности выбора для кубика с другим смещением из нашего примера. Вероятность взятия выборки типа 3, 4 и 5 показана на каждой линии; она показывает вероятность выбора для всех $x_{iA}, i = 1, \ldots, 10$ и $A \in \{L, \bar{L}\}$. Обратите внимание, что для любой стратегии формирования смещенной выборки посредством трехгранного смещенного кубика вероятности выбора движутся в сторону, противоположную от 0.5. Направление движения зависит от существования (A) карты лояльности по отношению к конкретной покупке x_{iA}. Изначальный смещенный кубик соответствует сдвигу, обозначенному соотношениями 1/5, 2/5, 2/5

[54] Англ. selection probability. – *Прим. перев.*

Именно здесь можно визуализировать смещение по-настоящему, потому что в каждом наблюдении популяции мы видим «наклон» в сторону, противоположную от 0.5 в зависимости от признака *L*. На рис. 6.8 показано смещение вероятностей выбора, p_i^{BS}, для кубика с другим смещением по сравнению с вероятностями выбора, $p_i^{SBS} = 0.5$, в рамках стратегии формирования репрезентативной выборки.

6.3.2 Оценивание по репрезентативной выборке

Разумеется, взятие выборок без следующего за этим шага оценивания само по себе не имеет особого смысла, поэтому давайте посмотрим на последующий шаг оценивания репрезентативной выборки. Допустим, мы хотим знать точное число покупок, совершенных при предъявлении карты лояльности; правильное значение этой характеристики для нашей популяции равно $\theta = 8$, и мы получим его, если будем прибавлять 1 для каждого элемента популяции с характеристикой *L* и 0 для всех тех, у кого ее нет:

$$\theta = \sum_{i=1}^{10} c_{iL} = 8\,.$$

Мы оценим эту популяционную сумму, задействовав ее аналог для случайной выборки, «выборочную сумму»:

$$\hat{\theta} = \sum_{j=1}^{5} I_{jL}.$$

Выборка *S* представляет собой случайное подмножество размера 5, поэтому необходимо учитывать, что вместо фиксированных значений c_{iL} теперь мы имеем дело со случайными индикаторными переменными I_{jL} для $j = 1, 2, 3, 4, 5$. О фиксированных значениях можно говорить только после того, как конкретная выборка была материализована. Индикаторная переменная I_{jL} равна 1, если *j*-й элемент выборки имеет характеристику *L*, и 0 в противном случае. Значение этой суммы для любой возможной выборки зависит от типа предыдущей выборки (выборки типа 3, 4 либо 5). Все выборки типа 3, 4 и 5 будут иметь значения соответственно 3, 4 и 5 этой суммы.

Поскольку сумма имеет произвольный характер, давайте проверим ожидание выборочной суммы (ее долгосрочное поведение в эксперименте, в котором мы многократно берем случайную выборку размера 5). Ожидание одной переменной I_{jL} для любого $j = 1, 2, 3, 4, 5$ в рамках стратегии формирования репрезентативной выборки равно 8/10 (мы оставляем это в качестве упражнения), а пять из них составляют четыре. Казалось бы, с четырьмя мы отклоняемся от цели в 2 раза. Но мы видим, что именно на этот коэффициент отклоняется и наша популяция, и размер выборки. Популяция имеет $10 = N$ членов, тогда как выборка *S* имеет $5 = k$. Следовательно, если увеличить масштаб $\hat{\theta}$ на N/k, то мы будем у цели с ожиданием. Окончательная формула оценщика такова:

$$\widehat{\theta_{su}} = \frac{10}{5}\sum_{j=1}^{5} I_{jL},$$

давая несмещенного и согласованного оценщика популяционной суммы θ (по мере увеличения размера выборки $\widehat{\theta_{su}}$ сходится к истинному значению θ = 8). Приведенная в данном уравнении общая форма вертикально масштабирующего оценщика называется *оценщиком Хорвица–Томпсона* суммарного значения признака.

Упражнение 5

Как бы вы рассчитали ожидаемое значение индикаторной переменной I_{jL}? Сможете ли вы продемонстрировать, что оно является тем, что мы утверждаем?

Шаг масштабирования – это общая стратегия, и ее также можно использовать для оценивания других популяционных сумм, таких как общая сумма, потраченная на онлайновые покупки за последний месяц (что здесь представляет собой интересующая нас популяция?). Либо если вы хотите узнать число покупок, совершенных на сумму более 100 долларов за последний месяц, то можно воспользоваться вертикально масштабирующим оценщиком, заменив характеристику L на «покупка на сумму более 100 долларов». Популяционные суммы, подобные θ, выглядят специфично, но по ним можно задавать любое ее линейное преобразование (любое среднее). Обратите внимание, что если разделить θ на 10, то получится $p_L = 4/5$, что тоже является разновидностью среднего, поэтому $\widehat{\theta_{su}}$ также можно использовать для оценивания p_L. Таким образом, стратегия формирования репрезентативной выборки и «вертикально масштабирующий» оценщик являются мощными общими инструментами, служащими для получения хорошей оценки соответствующего популяционного параметра.

Интересующая нас популяция может быть классической, как у Лапласа; это могут быть все кортежи покупок за прошлый месяц, хранящиеся в базе данных, или все покупки, прибывшие за последние 24 часа от производителей в конвейер обработки потоковых данных. Вероятностный аргумент одинаков для всех трех, но техническая реализация процесса отбора в этих трех областях, разумеется, различна.

Резюме

- Конвейер обработки потоковых данных является естественной средой для демонстрации алгоритмов и структур данных.
- Распределенные вычисления и императив реально-временной доставки результатов в приложениях по обработке потоковых данных создают многочисленные возможности для сокращения сквозных задержек за счет разумного использования хеширования, фильтров

Блума, набросков count-min и массивов HyperLogLog. Такие задания, как соединения больших таблиц, хранящихся в разнородных системах хранения, дедупликация в потоке, мониторинг сетевого трафика и балансировка нагрузки, являются реальными примерами подобных возможностей.

- Реально-временная аналитика возможна, если заинтересованные стороны могут договариваться об уровне терпимости к задержкам в таких системах. Механизмы генерирования данных подвержены периодическим или случайным изменениям, и наши алгоритмы обработки потоковых данных должны быть способны своевременно их учитывать. Модели потока данных, такие как скользящие окна на основе числа элементов либо количества времени, позволяют вносить последние коррективы, чтобы обнаруживать такие явления, как сдвиги в концепциях и дрейфы концепций.
- Взятие выборки – это мощный и давно зарекомендовавший себя метод ответа на вопросы о непрослеживаемом множестве путем систематического формирования его подмножества и ответа на тот же самый вопрос на его основе. Давно и хорошо зарекомендовавшая себя теория статистического вывода на основе несмещенной либо смещенной выборки помогает определяться с размером выборки и выбирать оценщика для ответа на вопрос с гарантией точности и прецизионности.

Глава 7

Формирование выборок из потоков данных

Эта глава охватывает следующие ниже темы:

- формирование выборки из бесконечного реперного потока;
- включение новизны с помощью скользящего окна и процедуры взятия выборок из него;
- демонстрация разницы между стратегиями формирования репрезентативной и смещенной выборки на реперном потоке с внезапным сдвигом;
- обследование пакетов и библиотек R и Python, разработанных для написания и исполнения заданий на потоках данных.

Мы готовы в полной мере оценить формирование выборки как отдельное задание, поставленное на ярусе анализа. Хотя мы уже показали, что разбивка архитектуры потоковых данных на ярусы не столь однозначна, мы вообразим, что процессор потоков формирует выборку из входящего потока на этом ярусе. Это поможет внедрить алгоритм формирования выборок без каких-либо дополнительных сложностей, связанных с дедупликацией, слиянием или общей предобработкой данных. В нашем примере с частотой отпечатков IP-адресов входящие запросы сначала проходят дедупликацию IP-адресов, а затем предстают перед процессором потоков, который материализует репрезентативную выборку. Текущее состояние выборки затем используется для приближенного, но быстрого ответа на непрерывный либо импровизированный запрос. Для иллюстрации каждого алгоритма мы будем использовать пример взятия выборки из IP-адресов.

Теория взятия выборок из потоков естественным образом развилась из взятия выборок из баз данных. Формирование (взятие) выборок из базы данных сопровождается продолжительными и обширными научными изысканиями и публикациями, начавшимися еще в 1986 году с работы Олкена (Olken) и Ротена (Roten) [1]. Одно из направлений исследований в области отбора из баз данных, *онлайновое агрегирование*, послужило началь-

ной платформой для главной темы этой главы – *взятие выборок из потоков данных*. Мы представим конкретные алгоритмы, которые работают с различными потоковыми моделями, рассмотренными в разделе 6.2.

7.1 Формирование выборок из реперного потока

Давайте займемся черновой работой и попытаемся «подключиться» к первому потоку данных, взяв выборку из модели реперного потока. Это непрерывный поток данных без окна. Элементы данных прибывают непрерывно, обрабатываются и исчезают навсегда. Пожалуй, *навсегда* – слишком сильно сказано, поскольку обычно поток перемещается в медленное массивное хранилище вторичной памяти. Отбор из такого потока должен каким-то образом обеспечивать, чтобы в каждый момент эволюции потока сохранялась «только» репрезентативная выборка данных, появившихся на настоящий момент. Это более простая задача по сравнению с процедурой взятия выборки из оконного (основанного на последовательности либо временны́х метках) потока данных. Здесь не требуется реализовывать логику обновления выборки после выхода элемента выборки из окна (его устаревания). Это неизбежно требует временны́х затрат и подводит к критерию для оценивания «добротности» алгоритмов формирования выборки, которые мы представим. Алгоритм, хорошо отвечающий на запрос, должен уметь создавать и/или обновлять выборку за один проход по элементам потока. Он также должен давать приближенный ответ на (непрерывный либо импровизированный) запрос, используя выборку полилогарифмического по N размера (N – это число элементов, появившихся на данный момент в реперном потоке). Понятие *приближенности* должно быть конкретизировано, когда разработчики алгоритмов хотят иметь возможность сравнивать свои алгоритмические решения. Приближенный ответ означает, что ответ должен быть в пределах ε абсолютной/относительной ошибки/отклонения от правильного ответа, за исключением некой небольшой вероятности неуспеха, δ, когда это не так. Мы хотим быть точными по ε в $100 \times (1 - \delta)$ процентах случаев. Размер окна ω в оконных потоках играет роль параметра N в части полилогарифмичности размера. Мы исходим из того, что размер ω слишком велик, чтобы вместить в память все кортежи между t и $t - \omega$.

7.1.1 Формирование выборки Бернулли

Взятие выборки Бернулли – это классическая стратегия формирования выборки (Даниэль Бернулли жил в XVIII веке, когда лист бумаги с данными об урожайности сельскохозяйственных культур во временной динамике в разных частях страны был наиболее близким понятием к базе данных). Это также стратегия формирования репрезентативной выборки, которая пережила свою вторую весну с появлением легко и быстро доступных элементов данных (с активным внедрением процедур взятия выборок из баз данных).

Данную стратегию проще всего проиллюстрировать, если представить, что вы играете в видоизмененную игру «любит – не любит», срывая лепестки с цветка по одному за раз (при этом надеясь, что к вам испытывают романтические чувства). После того как вы испортите совершенно прекрасный цветок, вы прекращаете игру. Причем то, что вы сделали, было выборкой лепестков в соответствии с вырожденной версией *формирования выборки Бернулли*: вы выбирали каждый лепесток с вероятностью $p = 1$. Естественно, процесс *формирования выборки Бернулли*, пригодный для практического применения, будет иметь $p \in (0,1)$, где p представляет истинную долю выборки[55] для любого числа элементов, появившихся на данный момент в потоке.

Если мы возьмем поток дедуплицированных IP-адресов, то сможем выбрать каждый $1/p$-й адрес, который пройдет. В этом вся прелесть простоты данного метода: в любой момент можно быть уверенным, что выборка полностью случайна и имеет размер pN, где N – это число элементов, появившихся на данный момент. Затем их можно использовать для вычисления приближенного числа отпечатков в расчете на IP-адрес во всем пространстве IP-адресов.

Для каждого прибывающего IP-адреса мы будем подбрасывать монету, показывающую орла с вероятностью p и решку с вероятностью $1 - p$. Если мы будем вводить появившийся в данный момент IP-адрес при каждом появлении орла, то у каждого IP-адреса будет p шансов попасть в выборку. К примеру, применяя справедливую монету, мы в среднем будем использовать каждый второй появившийся IP-адрес. Обратите внимание на неизбежную особенность: хотя p остается постоянной, размер k выборки является биномиально распределенной случайной величиной, и ее ожидаемый размер Np растет с увеличением N. Теперь можно определить процедуру взятия *выборки Бернулли* более формально: с учетом последовательности элементов $e_1, e_2, e_3, \ldots, e_i, \ldots$ из реперного потока включать каждый элемент e_i потока с вероятностью $p \in (0,1)$ независимо от любого другого элемента, который уже прошел или которому еще предстоит прийти. Эта идея проиллюстрирована на рис. 7.1.

Если вас не интересуют скучные детали реализации, то теперь у вас есть вся необходимая информация о взятии выборки Бернулли, чтобы перейти непосредственно к приведенному ниже псевдокоду наивной реализации. Наивная реализация вызывает алгоритм генератора псевдослучайных чисел для каждого появившегося элемента, выдает равномерно распределенное случайное число от 0 до 1 и включает элемент в выборку, если число меньше p.

Вникание в теорию, которая лежит в основе генераторов псевдослучайных чисел, на данном этапе привело бы к внезапному и времязатратному изменению контекста. К счастью, для того чтобы понять, о чем пойдет речь дальше, нужно знать только то, что генераторы псевдослучайных чисел – это эффективные детерминированные алгоритмы, генерирующие

[55] То есть истинное отношение размера выборки к размеру популяции. – *Прим. перев.*

последовательность псевдослучайных чисел. Если алгоритм придерживается некоторых допущений из теории чисел, то для практических целей указанные числа становятся неотличимыми от последовательности реальных случайных чисел. Также важно отметить, что хотя эти алгоритмы и эффективны, они требуют времени, и поскольку мы работаем в контексте обработки потоковых данных, мы не хотим вызывать их чаще, чем это необходимо.

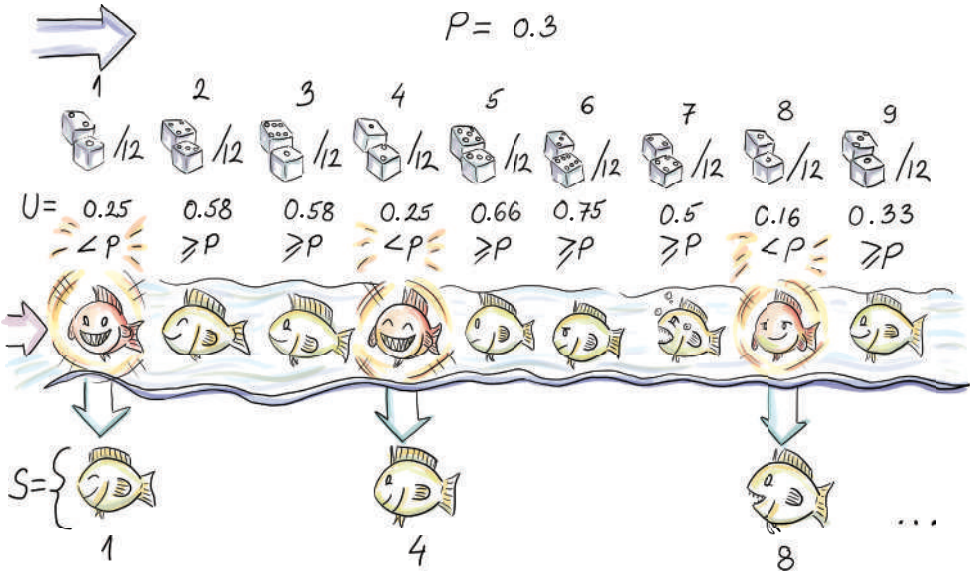


Рисунок 7.1 Вы видите, что в выборку включены первый, четвертый и восьмой элементы потока, поскольку соответствующее псевдослучайное значение, равномерное на интервале [0,1], оказалось меньше $p = 0.3$

Приятно отметить, что генераторы псевдослучайных чисел на самом деле считаются анафемой для случайности (как сказал Дж. фон Нейман: «Любой, кто рассматривает арифметические методы получения случайных цифр, конечно же, находится в состоянии грехопадения»), но теория чисел, лежащая в их основе, не менее увлекательна, так что если вы еще этого не сделали, то почитать о них будет нелишним. Не самое простое, но, пожалуй, самое лучшее место для этого – глава 3 книги Дональда Кнута «Искусство программирования», том 2: Полуцифровые алгоритмы (1998, Addison-Wesley Professional)[56].

Экономя на вызовах генераторов всевдослучайных чисел, наша реализация будет использовать тот факт, что число элементов, через которые нужно проскакивать после последнего включения, является геометрически распределенной случайной величиной. Вместо того чтобы предпринимать какие-то меры при каждом появлении нового элемента в потоке, мы будем это делать только тогда, когда новый элемент будет включаться в выборку.

[56] The Art of Computer Programming, Volume 2: Seminumerical Algorithms, D. Knuth. – *Прим. перев.*

Это объясняется тем, что мы генерируем «пропуск» индексов, мимо которых будем проскакивать всякий раз, чтобы достигать следующего включаемого элемента.

Мы используем очень общую теорему из теории вероятностей, именуемую *обратным интегральным преобразованием вероятности*[57]. (Извините за высокие слова!) В нашем случае она говорит о том, что если U является равномерно распределенной случайной величиной на интервале (0, 1), то $\Delta = \lfloor \log U / \log(1 - p) \rfloor$ дает число элементов, через которые нужно проскакивать перед следующим включаемым элементом ($\lfloor x \rfloor$ обозначает наименьшее целое число, меньшее или равное x), если мы хотим включать каждую p-ю. Уф-ф! Ниже приведен соответствующий псевдокод:

```
S = []                                  ❶
p = 0.01                                ❶
j = 0                                   ❷
i = 1                                   ❷

U = PRNG_unif(0,1)                      ❸

Δ = ⌊log U / log(1 − p)⌋                ❹

j = Δ + 1                               ❺

while (True):
    while (i! = j ):
        i += 1                          ❻

    S.append(e_i)                       ❼

    U = PRNG_unif(0,1)

Δ = ⌊log U / log(1 − p)⌋

    j = j + Δ + 1
```

❶ Инициализировать пустой буфер S для хранения выборки и задать вероятность выбора, p

❷ Задать j, индекс следующего включаемого элемента, равным 0, и индекс i текущего элемента равным 1

❸ Равномерно извлечь первый U

❹ Сделать первый пропуск/проскакивание

❺ Вычислить индекс первого включаемого элемента

❻ Передать все индексы между последним и следующим включениями

❼ Включить текущий элемент в выборку

[57] Англ. inverse probability integral transform. – *Прим. перев.*

В любой момент времени t выборка S является случайной выборкой Бернулли из всех появившихся на данный момент кортежей с вероятностью включения, p. Как видно из псевдокода, мы вызываем генератор псевдослучайных чисел и функцию log только во время включения нового элемента в выборку S.

В обобщенной версии формирования выборки Бернулли для каждого элемента используется отдельная и уникальная вероятность включения, p_i. Эта схема формирования выборки называется *взятием пуассоновской выборки*, и включение элемента X_i называется испытанием Бернулли с вероятностью успеха, p_i. Если у нас есть достаточно хорошее представление о кратностях X_i (то есть некоторые величины X_i в задаче оценивания общей суммы покупок в долларах появляются чаще, чем другие, скажем X_j), то мы можем позволить им отражаться в вероятностях включения. X_i с более высокими кратностями будет иметь более высокую p_i, тогда как те величины, которые появляются лишь изредка, будут иметь соответственно меньшую p_i. Этот тип формирования *смещенной выборки* позволяет прямолинейно строить оценщика Хорвица–Томпсона и сокращает дисперсию оценки. К сожалению, задача генерирования пропусков при формировании пуассоновской выборки не тривиальна.

В распределенной вычислительной среде, в которой работает любое промышленное приложение по обработке потоковых данных, алгоритм формирования выборки должен легко поддаваться параллелизации по нескольким потоковым операторам (формирования выборок). Преимущество стратегии формирования выборки Бернулли должно быть очевидным: взятие выборок из r подпотоков посредством отбора по Бернулли с вероятностью включения p будет приводить к получению репрезентативной выборки из всего потока после сведе́ния r выборок в назначенный ведущий/мастер-узел.

Главным недостатком формирования выборки Бернулли, как и пуассоновской, является случайный размер выборки; теоретически реперный поток будет расти бесконечно. Были предприняты попытки комбинирования отбора по Бернулли и стратегии укорочения размера выборки с целью удаления элементов из выборки Бернулли или использования процесса формирования резервуарной выборки, как только размер выборки превышает установленный порог. Такие стратегии вносят смещение в исходный алгоритм формирования выборки. В случае отбора по Бернулли стратегия формирования выборки становится смещенной, с разным смещением $p_i^{\text{BS}} - p$ для каждого элемента i. Зачастую не существует замкнутой функциональной формы этого смещения, которую можно было бы использовать для восстановления p_i^{BS} путем добавления этого смещения к p, и поэтому становится трудно правильно использовать оценщика наподобие оценщика Хорвица–Томпсона.

7.1.2 Формирование резервуарной выборки

Формирование резервуарной выборки[58] решает проблему переменного размера выборки. Этот алгоритм был популяризирован среди специалистов по информатике в статье Виттера (Vitter) [2], опубликованной в 1985 году. Для любого числа прочитанных из потока элементов выборка, взятая методом формирования резервуарной выборки, будет равномерно распределена между всеми выборками размера k. Доказательство этого утверждения широко доступно, поэтому здесь оно приведено не будет. Следовательно, мы получаем стратегию формирования простой случайной выборки фиксированного размера k из бесконечного реперного потока. Волшебство, да и только!

Алгоритм формирования резервуарной выборки оперирует на потоке данных следующим образом: k элементов потока сначала включаются в резервуар детерминированно (просто добаляя первые k элементов). Вероятность включения каждого дополнительного поступающего элемента с индексом i равна i/k для любого $i > k$. Если мы хотим включить элемент с индексом i, то другой элемент, в настоящее время находящийся в резервуаре, удаляется равномерно случайно, чтобы освободить свое место. Если добавить аналогичное сокращение, которое мы использовали ранее для обхода элементов путем генерации пропусков между теми, которые нужно включить, чтобы не заглядывать в каждый, то будет все необходимое, чтобы проверить ваше понимание алгоритма (см. рис. 7.1). Если вы не собираетесь реализовывать алгоритм, а просто его используете, вы все равно сможете понять рисунок. На нем изображен процесс формирования резервуарной выборки для первых семи элементов потока с использованием резервуара размера $k = 3$.

Прежде чем обсудить время выполнения алгоритма формирования резервуарной выборки, мы подробно опишем одну из возможных эффективных реализаций этой стратегии формирования выборки. Если такой уровень детализации вас не интересует, то можете смело пролистать следующие пару страниц.

Виттер дает эффективную реализацию алгоритма, используя ту же идею генерирования числа Δ_i пропускаемых элементов после включения в выборку элемента с индексом i. Обратите внимание, что здесь число пропускаемых элементов снабжено индексом; следовательно, пропуски имеют разное распределение в зависимости от того, какой объем потока данных был просмотрен до сих пор. Они изменяются по мере эволюции потока. Генерирование таких пропусков отличается большей сложностью, чем в схеме формирования выборки Бернулли, из-за неравной (уменьшающейся) вероятности включения по мере эволюции потока. Лежащая в ее основе теория не слишком сложна, и с ней можно ознакомиться в оригинальной статье либо в разделе 2.3 статьи Хааса (Haas) в рукописи Гарофалакиса

[58] Англ. reservoir sampling; син. отбор образцов в резервуар. – *Прим. перев.*

(Garofalakis), Герке (Gehrke) и Растоги (Rastogi) [3] (с этого места мы будем ссылаться на эту работу как GGR).

В данном методе генерирование пропусков Δ_i для «ранних» i предусматривает задействование упоминавшегося ранее обратного интегрального преобразования вероятности. Для «более поздних» i используется метод *принятия-отказа*[59] [4] в сочетании с аргументом *сжатия*[60].

Во втором из двух упомянутых случаев мы должны знать точную функциональную форму f_{Δ_i}, чего мы почти никогда не знаем. К счастью, Виттер вывел точную форму взятия резервуарной выборки, и эту форму вполне можно использовать. Тем не менее мы все равно должны ее оценивать, чтобы брать выборку с ее помощью, а это требует времени. Поэтому мы не хотим вызывать ее слишком часто. Вместо этого мы берем выборку, используя другую, простую в вычислении функцию, и применяем вероятностный аргумент, чтобы заявлять, что некоторые отобранные элементы происходят из f_{Δ_i} косвенно. Это высокоуровневая идея.

Если же говорить конкретно, то мы находим интегрируемую «треугольную» функцию[61] h_i в диапазоне f_{Δ_i}, являющуюся функцией массы вероятности пропусков Δ_i. Для того чтобы служить треугольной для f_{Δ_i}, мы должны иметь $f_{\Delta_i}(x) \leqslant h_i(x)$, имея в виду, что вероятность того, что Δ_i равна X, всегда будет меньше, чем $h_i(x)$. Затем мы нормализуем h_i с конечным значением α_i, которое является его интегралом над диапазоном f_{Δ_i}, чтобы получить валидную функцию плотности вероятности $g_i(x) = h_i(x)/\alpha_i$. Разумно производить выборку из диапазона f_{Δ_i}, используя $g_i(x)$. Мы берем случайное значение X из $g_i(x)$ и равномерно распределенный U из (0,1).

Если $U \leqslant f_{\Delta_i}(x)/\alpha_i g_i(x)$, то мы берем X, текущую реализацию X, чтобы получить случайное отклонение от f_{Δ_i}; в противном случае мы генерируем следующую пару (X, U) до тех пор, пока условие не будет выполнено. Если мы сильно злоупотребили обозначениями и теорией, то можно сказать, что $g_i(x)$, обусловленная на $U \leqslant f_{\Delta_i}(x)/\alpha_i g_i(x)$, тождественна f_{Δ_i}.

Теперь нужно рассмотреть еще одну деталь. Вспомните, что мы хотим отказаться от оценивания f_{Δ_i} из-за ее стоимости. *Сжатие* вводит функцию «перевернутого» треугольника или, пожалуй, чаши (ванны?), которая «подпирает» f_{Δ_i} снизу.

Таким образом, сжатие – это отыскание функции r_1, которую недорого вычислить, такой что $r_1 \leqslant f_{\Delta_i}$ для всех x в диапазоне. Тогда запрос $U \leqslant r_1(x)/\alpha_i g_i(x)$ может быть подтвержден утвердительным ответом на $U \leqslant r_1(x)$, и только в случае отрицательного ответа должна оцениваться более дорогая $U \leqslant f_{\Delta_i}(x)/\alpha_i g_i(x)$. Мы будем использовать f_{Δ_i}, g_i, α_i, r_1 и функцию кумулятивного распределения G_i в том виде, в каком Виттер вывел их для нас. В частности:

---

[59] Англ. acceptance-rejection method. – *Прим. перев.*

[60] Англ. squeezing argument. – *Прим. перев.*

[61] Англ. hat function; шляпная функция. – *Прим. перев.*

$$f_{\Delta_i}(m) = P(\Delta_i = m) = \frac{k}{i-k} \frac{\prod_{j=0}^{m}(i-k+j)}{\prod_{j=0}^{m}(i+1+j)};$$

$$F_{\Delta_i}(m) = P(\Delta_i \le m) = 1 - \frac{\prod_{j=0}^{m}(i+1-k+j)}{\prod_{j=0}^{m}(i+1+j)};$$

$$a_i = \frac{i+1}{i-k+1};$$

$$g_i(x) = \frac{k}{i+x}\left(\frac{i}{i+x}\right)^k;$$

$$G_i(x) = 1 - \left(\frac{i}{i+x}\right)^k;$$

$$r_i(m) = \frac{k}{i+1}\left(\frac{i-k+1}{i+m-k+1}\right)^{k+1}.$$

```
S = [None] * k                                                    ❶
j = 0
i = 0
Δ = 0                                                             ❶

while (i<k)                                                       ❷
    i+=1                                                          ❷
    S[i]=e_i                                                      ❷

i = i + 1                                                         ❸
U = PRNG_Unif(0,1)                                                ❹

while ⌈1 − ∏_{j=0}^{Δ}(i+1−k+j) / ∏_{j=0}^{Δ}(i+1+j)⌉ > U : Δ = Δ + 1   ❺

j = k + Δ + 1                                                     ❻

while (True):
    if i<=C AND i==j                                              ❼
        U = PRNG_Unif(0,1)
        d = 1 + floor(k*U)                                        ❽
        S[d] = e_i
        U = PRNG_Unif(0,1)

        while ⌈1 − ∏_{j=0}^{Δ}(i+1−k+j) / ∏_{j=0}^{Δ}(i+1+j)⌉ > U
            Δ = Δ + 1
            j = j + Δ + 1                                         ❾
    else if i>C AND i==j                                          ❿
```

```
        U = PRNG_Unif(0,1)
        d = 1 + floor(k*U)
        S[d] = e_i

        U=1

        while U ≥ f_Δi(X) / (a_i g_i(X))                    ⓫

            V = PRNG_Unif(0,1)
            X = I * (V**(-1/k)-1)                           ⓬
            U = PRNG_Unif(0,1)

            if u ≤ r_i(X) / (a_i g_i(X))                    ⓭
                break
                                                            ⓮

          Δ = X                                             ⓯
      j = j + Δ + 1                                         ⓰
    i = i + 1                                               ⓱
```

❶ Инициализировать пустой буфер (резервуар) `S` размера `k`. Задать `j`, индекс следующего элемента, таким образом, чтобы он включал индекс `i` текущего элемента, и первый пропуск `Δ` равным `0`

❷ Включить первые `k` элементов потока; `e_i` – это нагрузка

❸ Переместить индекс за блок повтора

❹ Взять `U` для первого пропуска

❺ Взять первый `Δ` из $F_{\Delta_1}^{-1}(U)$

❻ Вычислить индекс `j` первого включаемого элемента

❼ Ответвление для малых индексов, `i`

❽ Взять `d`, индекс текущего буферного элемента для записи поверх него

❾ Установить индекс для включения нового элемента

❿ Ответвление для больших индексов, `i (i > C)`

⓫ Выполнить принятие-отказ со сжатием

⓬ Взять X как $G_1^{-1}(1-V)$

⓭ Использовать r_1 вместо f_{Δ_i}, чтобы прервать цикл

⓮ В противном случае проверить более дорогую f_{Δ_i}

⓯ Взятый `X` – это новый пропуск `Δ`

⓰ Обновить `j`, чтобы указать на следующий включаемый элемент

⓱ Перейти к следующему элементу в потоке

Приведенный выше псевдокод показывает эффективную реализацию процедуры формирования резервуарной выборки. Размер выборки *k* должен быть задан некоторым значением. Используя пример из рис. 7.2, вы можете увязать этот псевдокод с вашим пониманием алгоритма.

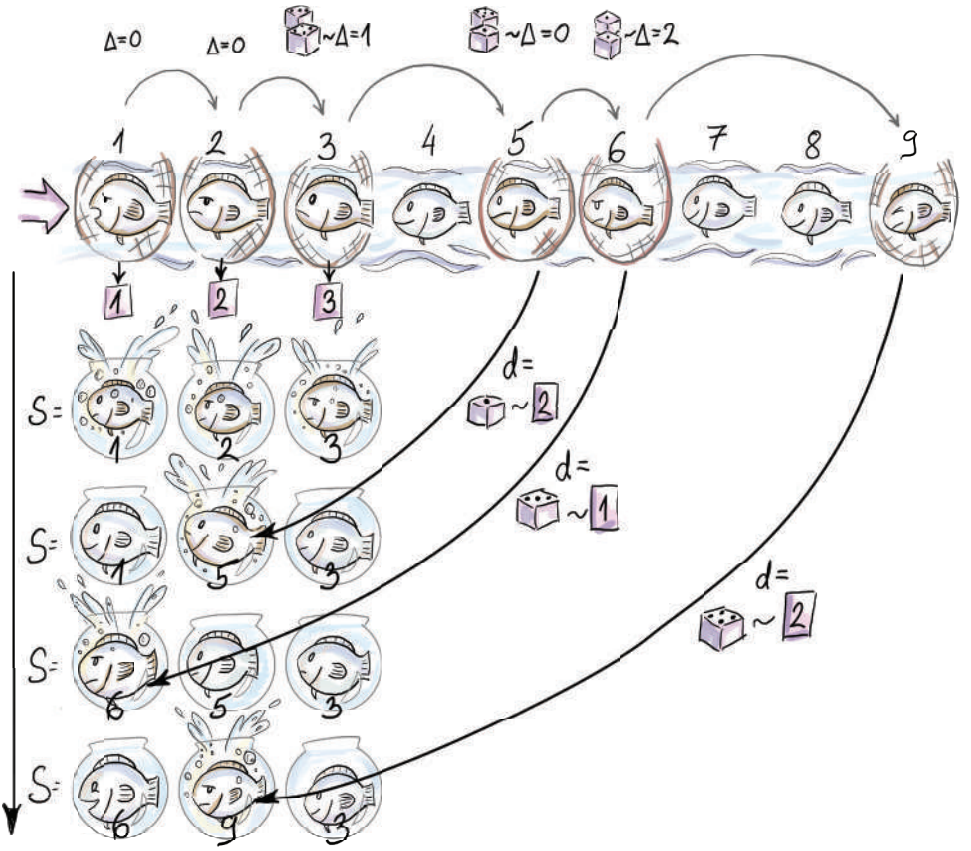


Рисунок 7.2 На рисунке показано содержимое резервуара после первых семи прибытий. Сначала в выборку детерминированно включаются три элемента. Впоследствии e_4 пропускается (это соответствует Δ_3, числу элементов, которые нужно пропустить после включения e_3, равного 1). Затем e_5 случайно заменяет e_2 в резервуаре ($d = 2$). Следующий элемент e_6 включается в произвольную позицию 1 в резервуаре и заменяет e_1 (обратите внимание, что это означает, что Δ_5 был равен 0). Δ_6 больше 0, так как по меньшей мере один элемент, e_7, пропущен

Такие методы, как обратное интегральное преобразование вероятности, метод принятия-отказа и сжатия, являются общими методами эффективного взятия выборок из любого распределения вероятностей, поэтому, хотя вам может потребоваться немного настойчивости, чтобы понять способы их применения, единожды поняв, вы сможете эффективно решать широкий спектр заданий, связанных с формированием выборок. Обратите внимание, что алгоритм работает для любого числа поступающих из потока элементов и может быть остановлен в любое время, приводя к простой случайной выборке размера k из всех элементов, появившихся на данный момент.

Время выполнения алгоритма формирования резервуарной выборки составляет O(k + log n/k), поэтому требования ко времени и пространству соответствуют нашему ограничению «малое пространство, малое время» – по меньшей мере в принципе, поскольку n в реперном потоке бесконечно. Возможно, лучше подумать, который использует выборку, после появления конкретного числа элементов. Анализ различий между временем выполнения наивной реализации и представленной здесь с геометрическими скачками см. в книге «Генерация неоднородных случайных величин» ([глава 12], **http://www.nrbook.com/devroye/**).

Для того чтобы понять, как алгоритм формирования резервуарной выборки обеспечивает простую случайную выборку, мы наглядно продемонстрируем очень тонкий баланс между двумя последовательностями вероятностей. Первая из них $k/i := p_i$ – это вероятность того, что i-й элемент включен (мы назвали ее вероятностью включения). Вторая – это вероятность того, что i-й элемент не будет удален из резервуара, после того как он был включен, если мы увидим $N - i$ элементов после него. Вторая вероятность относится к событию, когда все e_j (j в индексе), которые появляются в «дверях» после e_i (i в индексе), не могут удалить элемент e_i (i в индексе), который находится в резервуаре.

Для одного конкретного e_j вторая вероятность является суммой вероятности того, что e_j вообще не был выбран для включения (в данном случае e_j пропущен), и вероятности того, что если он был выбран для включения, ему не удалось удалить e_i. Эта сумма записывается как

$$q_{ij} = 1 - \frac{k}{j} + \frac{k}{j}\left(\frac{k-1}{k}\right) = 1 - \frac{1}{j}.$$

Следовательно, вероятность того, что e_i является частью резервуара, после того как будет встречено N элементов, равна

$$P_N(e_i \in S) = \min\left(1, \frac{k}{i}\right) \times \prod_{j=\max(i,k)+1}^{N} \frac{j-1}{j}.$$

Мы будем называть ее *вероятностью наличия*[62] в точке N. Первая часть произведения (слева от ×), которую мы обозначим через p_i, является *вероятностью включения*[63] для e_i. Вторая часть (справа от ×) – это кумулятивное произведение (мы будем обозначать его через π_{iN}), которое отражает вероятность того, что ни один из последующих элементов не удалит e_i (предполагая, что мы видим в общей сложности N элементов). Мы использовали min, чтобы обобщить выражение с учетом также первых k элементов (которые включены детерминированно). На рис. 7.3 показаны эти две противоположные силы и результирующая $P_N(e_i \in S)$ для каждого e_i, $N = 100$ и $k = 10$.

[62] Англ. residing probability. – *Прим. перев.*

[63] Англ. inclusion probability. – *Прим. перев.*

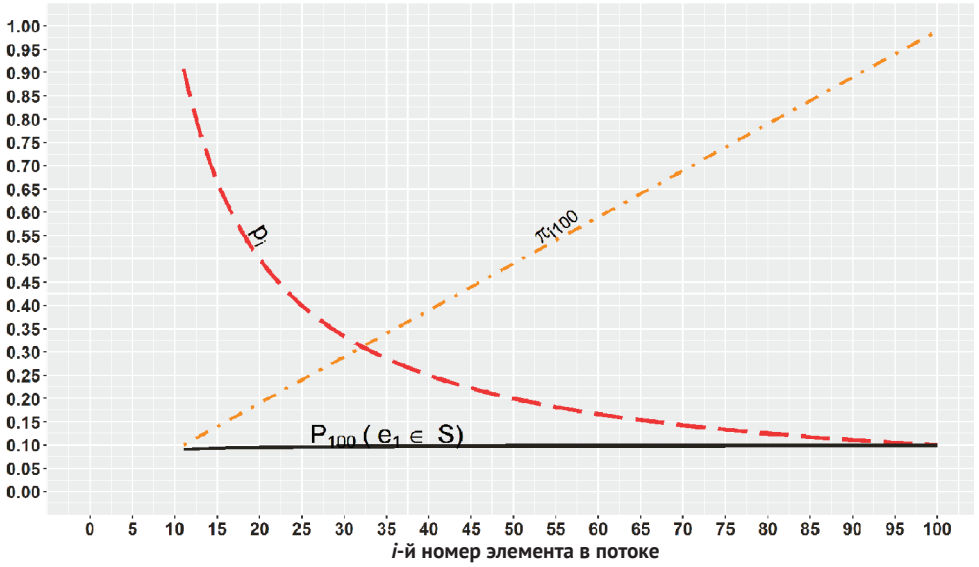


Рисунок 7.3 Мы видим баланс между вероятностями включения (изогнутая, пунктирная линия) и вероятностями удаления вплоть до точки $N = 100$ (прямая, пунктирная линия), отраженный в $P_{100}(e_i \in S)$ (сплошная линия), который приближенно постоянен для всех элементов, появившихся на данный момент (когда резервуар имеет размер 10). Это означает, что каждый элемент, независимо от того, когда он встретился, имеет одинаковые шансы попасть в выборку

Возможно, вы помните, что существует не так много реалистичных потоков данных, в которых мы хотим, чтобы отдаленное прошлое влияло на текущий запрос в той же степени, что и более недавнее прошлое. Такое различающееся взвешивание элементов в зависимости от времени их прибытия невозможно выполнять в рамках формирования резервуарной выборки, поэтому исследователи пытаются «склонять» чашу весов, как показано на рис. 7.3, в ту сторону, где более поздние элементы с большей вероятностью окажутся частью окончательной выборки по сравнению с теми, которые прибыли ранее. Благодаря этому черная линия поднимается вверх по мере приближения к текущему моменту. Это подводит нас к следующему алгоритму формирования выборки – *формированию смещенной резервуарной выборки*.

7.1.3 Формирование смещенной резервуарной выборки

Для того чтобы смещать выборку, мы сосредоточимся на вероятности наличия в точке N, $P_N(e_i \in S)$, которая определяет, не находится ли элемент e_i в момент времени N в резервуаре после того, как появилось $N - i$ элементов после него. Мы хотели бы иметь возможность отклонять $P_N(e_i \in S)$ от

репрезентативного равновесия, при котором все e_i имеют равные $P_N(e_i \in S)$ (см. рис. 7.3). Один из способов состоит в допущении, что $P_N(e_i \in S)$ уменьшается при каждом прибытии нового элемента из потока. Элементы устаревают недетерминированно. Благодаря этому, когда e_i становится все более отдаленным прошлым, а мы предпочли бы не иметь его в нашем текущем резервуаре, его вероятность становится очень малой. В примере с IP-адресами вас, возможно, заинтересует среднее число отпечатков в расчете на IP-адрес только за последний день трафика. В этом случае нужен механизм управления вероятностями наличия, который позволит элементам дневной давности присутствовать в резервуаре, но не более старым.

Для этого мы моделируем вероятности наличия, используя некую не имеющую памяти функцию смещения, $f(i, N)$. Несмотря на то что эта функция имеет два параметра, i (-й элемент из потока) и N (число элементов, появившихся на данный момент), она оценивается одинаково для всех пар (i, N), которые имеют одинаковое расстояние ($N - i$) между ними. Следовательно, $P_N(e_i \in S) = P_{N+k}(e_{i+k} \in S)$, имея в виду, что e_i после того, как мы увидели N элементов, имеет ту же вероятность наличия в резервуаре, что и $e_i + k$ после того, как мы увидели $N + k$ элементов. Мы запрашиваем элементы, которые находятся на одинаковом расстоянии в прошлом от двух соответствующих моментов запроса, N и $N + k$. Тот факт, что за это время мы увидели k новых элементов, оставляет нетронутыми две *вероятности наличия*, $P_N(e_i \in S)$ и $P_{N+k}(e_{i+k} \in S)$. Это то, что имеется в виду, когда мы говорим *не имеет памяти*. Функция не «запоминает» абсолютный момент времени; ей просто нужно знать, о каком далеком прошлом мы спрашиваем.

Вы можете представить себе наклон $P_N(e_i \in S)$ (см. на рис. 7.3 там, где установлено фиксированное значение $N = 100$). Обратите внимание, что вероятность наличия $P_N(e_i \in S)$ остается постоянной при увеличении i. Это ожидаемое поведение классического алгоритма формирования резервуарной выборки. Процесс формирования смещенной резервуарной выборки имел бы $P_N(e_i \in S)$ больше по мере приближения к текущему моменту и меньше к началу потока («началу времени»).

В оригинальной статье о формировании смещенной резервуарной выборки Аггарвала (Aggarwal) [5] используется не имеющая памяти функция экспоненциального смещения $(i, N) = e^{-\lambda(N-i)}$. Заметили ($N - i$) в отрицательной экспоненте? Рассматривая выражение, мы замечаем, что чем шире промежуток, тем больше экспонента (и, следовательно, тем меньше отрицательная экспонента). Это делает все выражение, то есть вероятности наличия, меньше $P_N(e_i \in S)$. В потоках финансовых данных или любом другом потоке, для которого полезно устаревание элементов, это то, чего мы и хотим.

Параметр λ служит коэффициентом *скорости устаревания*. На рис. 7.4 показано значение $P_N(e_i \in S) = f(i, N)$ для нескольких разных значений λ. Мы снабдим вас непосредственным пониманием на интуитивном уровне относительно λ.

В рамках упражнения стоит запомнить, что $P_N(e_i \in S)/P_{N-1}(e_i \in S) = e^{-\lambda}$. Другими словами, после прибытия одного нового элемента вероятность наличия текущего элемента уменьшается в $e^{-\lambda}$ раз. Краевой случай $\lambda = 0$ означает «никогда не забывать», и для наших целей это бесполезное значение λ, но оно помогает наблюдать за происходящим вблизи него. Если двигаться от $\lambda = 0$ вправо малыми, увеличивающимися шагами (то есть $\lambda = 0, 0.001, 0.01, 0.1, \ldots$), то $e^{-\lambda}$ будет принимать значения 1, 0.999, 0.999 и 0.9 в указанном порядке. Здесь мы видим, как λ управляет скоростью устаревания; прибытие одного нового элемента увеличивает вероятность наличия на 99.9 %, 99 % или 90 % от того, что было до прибытия. Из этой управляемой λ прогрессии теперь можно вывести число элементов, которые должны прибыть, чтобы p_i полностью состарилась. Если экстраполировать рассуждения, то мы увидим, что $e^{-\lambda(N-i)}$ является величиной, обратной числу элементов, которые должны прибыть, чтобы уменьшить $P_N(e_i \in S)$ в e^{-1} раз (то есть умножено приблизительно на 0.36; см. рис. 7.4). Уф-ф! Получилось немало!

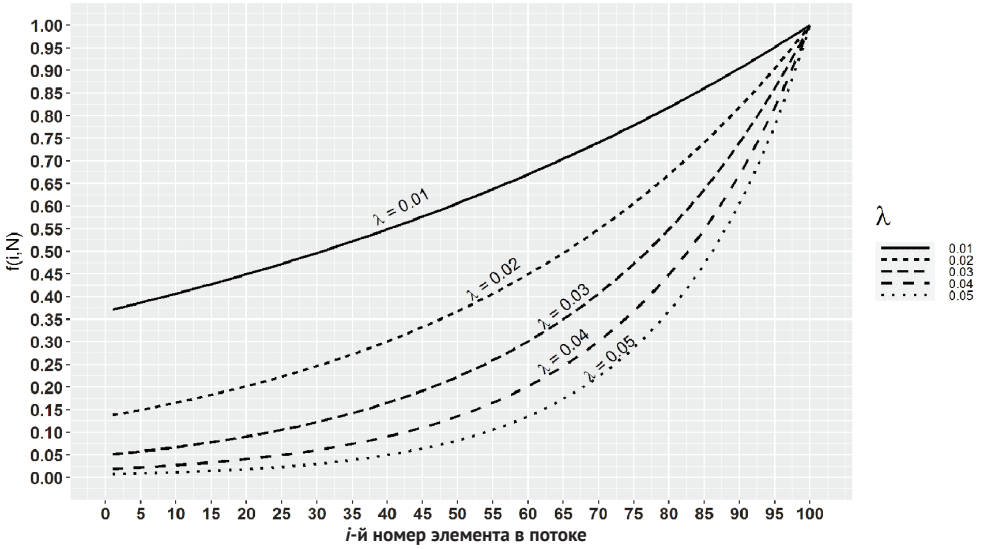


Рисунок 7.4 Обратите внимание, что для $\lambda = 0.01$ нам нужно 100 элементов, чтобы уменьшить $f(i,N)$ в 0.36 раза, тогда как для $\lambda = 0.02$ это число равно 50. Таким образом, чем выше λ, тем легче забывать старые элементы

Сейчас мы посмотрим, как один такой конкретный λ также влияет на размер выборки. Эта схема взятия смещенной выборки не бесплатна, и чтобы поддерживать выборку из реперного потока, мы должны удовлетворять требованиям к минимальному пространству; мы должны знать, как размер выборки увеличивается с учетом появившихся элементов. Авторы оригинальной статьи обозначили выборку через $S(n)$, чтобы указать на ее

зависимость от числа появившихся элементов. Что удобно, они также доказывают, что для больших N (соответствующих реалистичным потокам) размер выборки ограничен сверху границей $1/1 - e^{-\lambda}$. Исходя из этого, мы определили максимальный размер выборки, необходимый для достижения скоростей λ, при которых $P_N(e_i \in S)$ сокращается соответственно медленно (или быстро). Эта первая граница может быть заменена на $1/\lambda$ (используя основную теорему математического анализа), так что нам нужно лишь посодействовать в том, чтобы было достаточно пространства для конкретного, управляемого приложением параметра λ, дабы управлять смещением. Если вычисленный в примере параметр λ соответствует этому ограничению, то мы сможем использовать функцию экспоненциального смещения с этим λ. В нашем случае максимальный размер выборки находится где-то между 10^4 и 10^5. Для случая, когда можно содержать весь максимальный размер выборки в пределах (эффективных) пространственных ограничений потокового приложения, можно использовать следующий простой алгоритм, чтобы поддерживать смещенную выборку над любым числом элементов из потока.

Допустим, что j-й элемент в потоке только что прибыл. Обозначим занятую долю резервуара через $F(j) \in (0,1)$. Новый элемент e_j детерминированно добавляется в резервуар. Это может происходить двумя способами: e_j заменяет случайно выбранный элемент из резервуара с вероятностью $F(j)$ и e_j добавляется в резервуар без каких-либо удалений с комплементарной вероятностью. Псевдокод этой версии взятия смещенной резервуарной выборки показан на следующей ниже странице. На рис. 7.5 представлена стратегия формирования смещенной резервуарной выборки для резервуара размера $k = 3(\lambda = 1/3)$ для первых семи элементов потока.

Пример 1

В нашем случае использования со средним числом отпечатков на каждый IP-адрес мы могли бы иметь постоянную скорость прибытия, равную 12 элементам в секунду. Мы хотели бы знать среднее число отпечатков в расчете на IP-адреса, появившиеся за последние 24 часа. Это 86 400 секунд, за которые мы видим 12 × 86 400 = 1 036 800 IP-адресов с их отпечатками. Параметр λ должен уменьшать $P_1(e_1 \in S)$ начиная с 1 так, чтобы $P_{1036801}(e_1 \in S)$ фактически была равна 0. Это означает, что после того, как мы увидим 1 036 801 элемент, вероятность наличия для первого из них должна практически упасть до 0. Каким было бы значение λ, при котором это произошло бы в нашем приложении? Это эквивалентно вопросу «для какого λ справедливо, что $e^{-\lambda \times 1036800} = 0$?». Используя метод проб и ошибок, вы можете проверить, что для $\lambda = 10^{-6}$ $e^{-\lambda \times 1036800}$ равно 0.35, тогда как для $\lambda = 10^{-5}$ оно становится 3×10^{-5}. Таким образом, λ находится именно между этими двумя значениями, если мы хотим «выхватывать» элементы постепенно в течение дня.

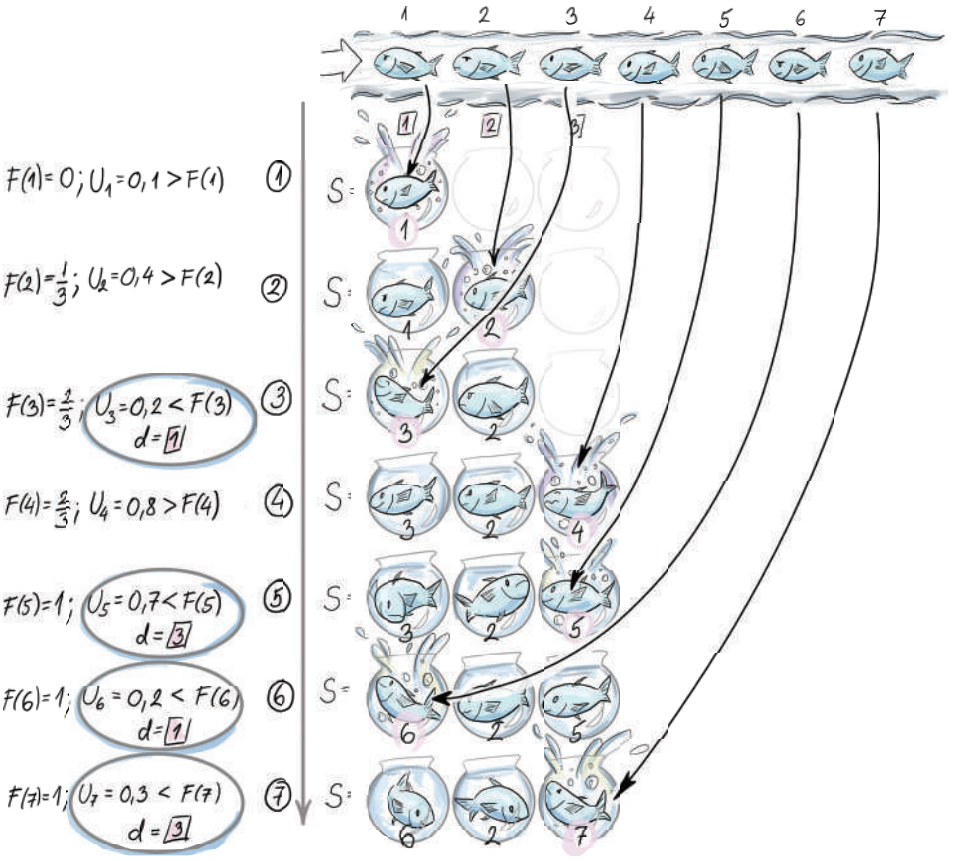


Рисунок 7.5 Содержимое резервуара для первых семи прибытий в рамках стратегии формирования смещенной резервуарной выборки

Элемент e_1 включается детерминированно, а элемент e_2 вставляется без удаления каких-либо существующих элементов из резервуара из-за значений $F(2)$ и U_2. Поскольку занимаемая часть резервуара растет, вероятность включения следующего элемента за счет одного существующего элемента выше ($U_3 < F(3)$); следовательно, e_3 сохраняется в первом месте ($d = 1$). Обратите внимание, что для этого есть две возможные позиции: 1 и 2. e_4 включается в третью позицию без удаления каких-либо элементов ($U_4 < F(3)$). Поскольку на данный момент весь резервуар занят, включаются элементы e_5, e_6 и e_7, и все это за счет элементов, сохраненных в позициях 3, 1 и 3, в указанном порядке:

```
S = [None] * 1/λ                    ❶
COP = 0                             ❶
i = 1                               ❶

while (True)
```

```
U = PRNG_Unif(0,1)
if U < COP
    U = PRNG_Unif(0,1)
    D = 1 + floor(k * COP * U)      ❷
    S[d] = e_i                      ❸
else
    d = 1 + floor(k * COP)
    S[d] = e_i
    COP += 1/k                      ❹
```

❶ Инициализировать пустой буфер (резервуар) `S` размера `k = 1/λ`. Задать `i`, индекс текущего элемента, равным 1, а текущую занятую долю COP[64] резервуара – равной 0
❷ Взять индекс `d` между 1 и максимальным индексом заполнения, `k * COP`, резервуара
❸ Сохранить элемент `e_i` в этом случайном индексе в занятой части резервуара
❹ Добавить текущий элемент в резервуар. В данном случае мы также должны обновить текущую занятую долю

Вы можете пройтись по псевдокоду и использовать пример из рис. 7.5, чтобы проверить ваши рассуждения. Обратите внимание, что после заполнения резервуара значение текущей занятой доли равно 1, и после этого всегда выполняется ветвь `if`.

Упражнение 2

Реализуйте стратегию формирования смещенной резервуарной выборки, используя приведенный выше псевдокод и пакет языка R `stream`, представленный в разделе 7.3, или Python 3.0.

Когда $1/\lambda$ не умещается в доступной нам рабочей буферной памяти, алгоритм модифицируется, «замедляя» вставки за счет введения вероятности вставки $p_{ins} = k\lambda$ вместо $p_{ins} = 1$. Это создает возможность реализации такого же смещения в формировании выборки, но с меньшим размером выборки, p_{ins}/λ.

Такая модификация устраняет проблему слишком медленного первоначального заполнения резервуара. Это может привести к длительному времени ожидания ответа на запросы о размере выборки, гарантирующие приемлемые стандарты точности и прецизионности. Аггарвал дает стратегию решения этой проблемы, поэтому, при необходимости, обратитесь к его статье [5], чтобы ее реализовать.

7.2 Формирование выборок из скользящего окна

Сначала мы обсудим вопрос о том, как брать выборку из окна, основанного на последовательности. Здесь новизна измеряется в порядковом смысле как число прибывших элементов. В потоке IP-адресов IP-адреса могут поступать в разное время, но длина окна будет составлять 1000 адресов,

[64] Англ. currently occupied proportion (COP). – *Прим. перев.*

независимо от того, как это число распределяется по шкале времени. Мы будем иметь дело с последовательностью окон (следовательно, скользящих), W_i, $j \leq 1$, где каждое индексированное окно влечет за собой n элементов, e_j, e_{j+1}, e_{j+2}, ..., e_{j+n-1}. Указанное n не меняется с эволюцией потока, как это происходит с N. Промежутки между двумя прибытиями, как правило, могут отличаться в абсолютных единицах времени, но окна, основанные на последовательности, расценивают этот аспект как несущественный и регистрируют элементы в поочередных целых позициях. Мы будем поддерживать выборку размера k из текущего окна. Обратите внимание, что теперь нам нужно выработать стратегию обновления выборки не только тогда, когда мы решаем вставить новый элемент в выборку, но и всякий раз, когда самый старый текущий элемент выборки выходит из текущего окна («устаревает»). Мы не исходим из допущения, что размер окна n может укладываться в рабочую память; поэтому целесообразно брать выборку из этого окна.

7.2.1 Формирование цепной выборки

Сначала мы объясним, как брать случайную выборку размера 1 из текущего окна и обновлять ее по мере перемещения окна. Алгоритм формирования случайной выборки размера k в таком случае будет просто одновременным (параллельным) выполнением k экземпляров (цепочек) стратегии поддержания лишь одного случайного элемента.

Первоначальная фаза алгоритма, которая длится n дискретных временных шагов (по длине окна), представляет собой формирование обычной несмещенной резервуарной выборки с некоторыми дополнительными операциями. Каждый прибывающий элемент e_i будет выбираться в качестве выборки $S = \{e_i\}$ с вероятностью i/j. Эта часть относится к алгоритму формирования резервуарной выборки. Управляющее скользящим окном дополнение будет выбирать элемент, который будет заменять e_i, когда этот элемент устареет. Мы не делали этого в алгоритме формирования резервуарной выборки. Следовательно, всякий раз, когда мы выбираем случайный будущий индекс, $K \in \{i+1, i+2, i+n)$ и добавляем (K, .) в цепочку, мы знаем, что K-й элемент будет сохранен там, как только он попадет в окно. Он становится вторым элементом цепочки. Первый элемент (i, e_i) хранит текущую выборку размера 1. После просмотра всего окна W_1 (прохода n элементов) в нашем распоряжении получится простая случайная выборка из W_1 размера 1, поскольку стратегия формирования резервуарной выборки это гарантирует. Вдобавок у нас есть последний K, индекс кортежа, который его заменит, как только выборка размера 1 истечет. Теперь, для каждого прибывающего элемента, $i = n + 1, n + 2, \ldots$, у нас есть варианты:

- с вероятностью $1/n$ мы отбрасываем текущую выборку $S = \{e_i\}$ и связанную с ней цепочку, сохраняя индекс K и элемент e_K, который должен был ее унаследовать по истечении e_j;

- мы заменяем ее на $S = \{e_i\}$. Теперь у e_j должен быть преемник, который вступит во владение после истечения e_j, поэтому мы выбираем случайный будущий индекс, $K \in \{i+1, i+2, i+n\}$, и добавляем его в качестве второго элемента во вновь созданную цепочку;
- с вероятностью $(1 - 1/n)$ мы делаем проверку, не является ли i следующим замещающим элементом, который будет сохранен в цепочке ($K = i$?). Если да, то сохраняем i-й кортеж в (последнем) элементе цепочки. Мы выбираем случайный будущий индекс, $K \in \{i+1, i+2, i+n\}$, элемента, который заменит e_j по его истечении, и добавляем его в конец цепочки. Именно так и растет цепочка. В случае $= j + n$, означающем, что e_j покидает окно, второй элемент в цепочке перемещается вверх, а истекшая выборка, $S = \{e_i\}$, удаляется из верхней части цепочки.

Эти варианты предоставляют в каждый отдельный момент i, связанный с обновлением окна, простую случайную выборку размера 1 из окна W_{i-n+1}. На рис. 7.6 показана стратегия формирования цепной выборки для первых семи элементов и размера окна $n = 3$.

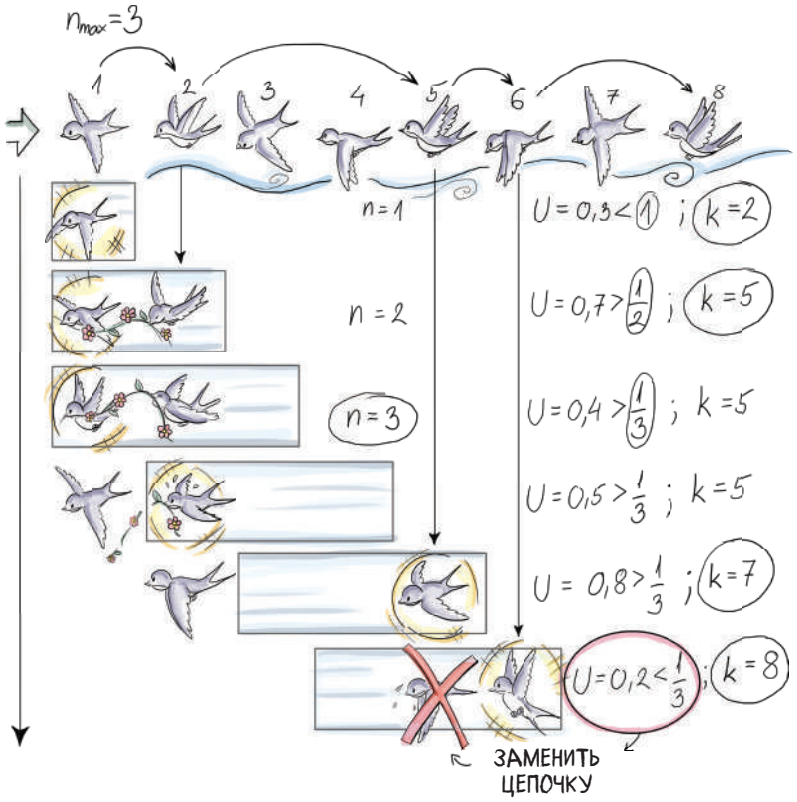


Рисунок 7.6 Содержимое цепочки (список L) для первых семи элементов из оконного потока, основанного на последовательности размера $n = 3$

При чтении постарайтесь поглядывать на рис. 7.6. Сначала детерминированно включается элемент e_1. Затем мы выбираем будущий индекс K, который заменит e_1 по прибытии e_K. K, по-видимому, равен 2. По завершении работы с первым элементом цепочка влечет за собой (1, e_1) и (2, .). В этот момент e_1 является случайной выборкой размера 1. Но, продолжая пример, прибывает элемент 2, и мы замечаем, что он должен быть сохранен как преемник элемента e_1, что немедленно и выполняется, и теперь цепочка хранит (1, e_1) и (2, e_2). Эти связанные с преемником служебные операции выполняются еще до того, как мы решим их выполнить, если выберем e_2 с вероятностью ½ (формирование резервуарной выборки) и полностью отбросим e_1 и его цепочку. Мы бросаем кубик и $U > 1/2$ (в нашем случае $U = 0.7$); следовательно, e_2 не приводит к отбрасыванию существующей цепочки. В завершение работы с e_2 берется его преемник, и оказывается, что $K = 5$. Следовательно, текущая цепочка равна (1, e_1), (2, e_2) и (5, .). Элемент e_3 тоже не приводит к удалению существующей цепочки, так как $U_3 > 1/3$. Поскольку e_3 не является ничьим преемником, он не будет включен в цепочку, и мы двигаемся дальше. Так как при четвертом прибытии длина окна равна $n = 3$, элемент e_1 истекает. Сначала необходимо обновить текущий элемент выборки его преемником в цепочке. Эту роль берет на себя e_2. Элемент e_4 не приводит к отбрасыванию существующей цепочки ($U = 0.4 > 1/3$), и она не была выбрана в качестве чьего-либо преемника, поэтому мы двигаемся дальше. Обратите внимание, что e_4 является первым элементом во второй фазе формирования нерезервуарной выборки. При пятом прибытии происходят две вещи. Во-первых, к назначенной позиции в цепочке добавляется e_5, а также включается индекс его преемника, = 7, таким образом, цепочка равна (2, e_2), (5, e_5) и (7, .), и она находится на пике своей длины. Во-вторых, поскольку e_2 истекает, следующий элемент в цепочке, недавно добавленный e_5, его заменяет. Закончив с e_5, у нас будут (5, e_5) и (7, .) в качестве текущей цепочки и e_5 в качестве текущей выборки размера 1. По прибытии e_6 мы случайно решаем отбросить текущую выборку и начать новую ($U = 0.2 < 1/3$). Текущая цепочка вместе с выборкой отбрасывается, а элемент e_6 добавляется в новую цепочку и становится новой выборкой. Выводится индекс ее преемника K, который в нашем примере равен 8. Элемент e_7 не приводит к отбрасыванию цепочки, и поскольку она не является ничьим преемником (она принадлежала e_5, но эта цепочка была расформирована), мы проходим мимо нее. В каждый момент эволюции потока из рис. 7.6 можно считывать два важных момента: выборку размера 1 (ласкового воробья) и какое скользящее окно она представляет (окно, в котором воробей находится). Выборка всегда является верхним элементом списка.

Далее показан псевдокод с подробными комментариями по взятию выборки размера 1 с использованием алгоритма формирования цепной выборки. Вы можете проследить псевдокод по рис. 7.6 и увидеть места, где цепочка становится длиннее и где ее элементы отбрасываются, чтобы начать новую цепочку:

```
L = []                                  ❶
i = 1                                   ❶
K = 0                                   ❶

while i<=n                              ❷
    U = PRNG_UNIF(0,1)
    if U < 1/i                          ❸
        L.clear()                       ❹
        L.append([e_i, i])              ❺
        U = PRNG_Unif(0,1)
        K = i + floor(n*U) + 1
    else
        if i == K                       ❻
            L.append([e_i, i])
            U = PRNG_Unif(0,1)
            K = i + floor(n*U) + 1
    i+=1

while True                              ❼
    if (i==j+n)
        L = L.pop(1)                    ❽
    U = PRNG_Unif(0,1)
    if U < 1/n
        L.clear()
        L.append
        U = PRNG_Unif(0,1)
        K = i + floor(n*U) + 1
    else if i==K
        L.append([e_i, 1])
        U = PRNG_Unif(0,1)
        K = I + floor(n*U) + 1
    i+=1
```

❶ В начале `L` пуст. Задать индекс `i` текущего элемента равным 1. Задать `K`, индекс будущего замещающего элемента, равным 0
❷ Первая фаза с `n` элементами
❸ Процесс формирования резервуарной выборки решает удерживать `e_i`
❹ Удалить текущую выборку и ее цепочку
❺ Добавить текущий элемент `e_i` в цепочку и определить его преемника `K`
❻ Процесс формирования резервуарной выборки решает пропустить `e_i`, поэтому мы смотрим, не является ли он чьим-либо преемником
❼ Вторая фаза для `I = n + 1, n + 2, ...`
❽ Удалить верхний элемент списка, потому что он выпадает из окна

Упражнение 2

Реализуйте стратегию формирования цепной выборки для окна длиной 100, размером выборки 1 и любого $N > 100$. Анализ пространственной сложности для хранения k независимых цепочек можно найти в ориги-

нальном техническом отчете Бэбкока (Babcock), Датара (Datar) и Мотвани (Motwani)[6] или в статье GGR [7]. Ожидаемое потребление памяти для k цепочек равно $O(k)$, то есть все они имеют не более длины, ограниченной константой. Пространственная сложность алгоритма не превышает $O(k \log n)$ с вероятностью $1 - O(n^{-c})$, следовательно, по нашим критериям она эффективна.

Обратите внимание, что каждая цепочка в алгоритме формирования цепной выборки обеспечивает простую случайную выборку без возврата, в каждый момент времени, размера 1 из текущего окна. Тем не менее при поддержании k параллельных цепочек за раз алгоритм будет выдавать простую случайную выборку длины k с возвратом[65], но это не является ограничивающим фактором.

Далее мы представим аналогичный алгоритм для поддержания выборки размера k из скользящего окна, основанного на временны́х метках.

7.2.2 Формирование приоритетной выборки

При работе с окнами, основанными на временны́х метках, мы не знаем точного числа элементов n в окне, поэтому привязать наш алгоритм к этому параметру невозможно. Для поддержания простой случайной выборки размера 1 в окне, основанном на временны́х метках, мы генерируем приоритет p_t для каждого прибывающего элемента e_t, берущегося равномерно из интервала (0,1). Элемент с наивысшим приоритетом в окне (сейчас – $\omega < t <$ сейчас) представляет собой простую случайную выборку размера 1. Как и с формированием цепной выборки, мы поддерживаем преемников, чтобы наследовать выборку сразу после того, как текущая выборка выйдет из временно́го окна.

Первый элемент e_{t_1} становится выборкой детерминированно, поскольку нет приоритета ($p_0 = 0$), который необходимо преодолеть. По прибытии второго элемента, e_{t_2}, в момент времени t_2 мы проверяем на истинность выражение $p_{t_2} > p_{t_1}$; если оно истинно, то e_{t_2} заменяет e_{t_1} (e_{t_1} удаляется из памяти). В противном случае (e_{t_2}, p_t) сохраняется в связном списке и в качестве первого элемента списка (выборка сохраняется отдельно и не является элементом списка). По прибытии e_{t_3} используется три разных сценария упорядочивания приоритетов в памяти, p_{t_1}, p_{t_2} и p_{t_3}, вновь прибывшего элемента e_{t_3}:

1. $p_{t_1} > p_{t_2} < p_{t_3}$: e_{t_3} добавляется в конец списка, который поддерживает замены на случай, если истечет элемент e_{t_1}, текущие выборки. Список упорядочен по убыванию приоритета и для каждого создания – по возрастанию времени.
2. $p_{t_1} > p_{t_3} < p_{t_2}$: e_{t_3} добавляется за e_{t_1}, а все элементы (в настоящее время это только e_{t_2}) с более низким приоритетом и (неизбежно) более низкой временной меткой удаляются из списка/памяти. Список остается

[65] Англ. with replacement; то есть с возвратом отобранных значений назад в популяцию. – *Прим. перев.*

упорядоченным по убыванию приоритета и для каждого создания – по возрастанию времени.

3. $p_{t_3} > p_{t_1} < p_{t_2}$: e_{t_3} добавляется в начало, а все остальные элементы удаляются из списка/памяти. Список остается упорядоченным по убыванию приоритета и для каждого создания – по возрастанию времени.

В первом случае новый элемент добавляется в конец отсортированного списка (по приоритету), и весь список остается. Во втором случае остается та часть списка, которая имеет более высокий приоритет, чем у нового элемента. В третьем случае предыдущий список отбрасывается, и новый элемент добавляется в верхнюю часть нового списка. Остальные элементы отбрасываются, а новый элемент сохраняется последним.

Алгоритм продолжает обновлять список каждым прибывающим элементом одним из описанных способов. В каждый момент времени l в верхней части списка находится элемент e_t, имеющий второй по старшинству приоритет среди элементов из окна $W_{1-\omega}$, где ω – это продолжительность окна ($l - \omega < t < l$). Текущая выборка – это элемент с наивысшим приоритетом в течение времени t ($l - \omega < t < l$). Элемент из верхней части списка заменяет текущую выборку и становится новой простой случайной выборкой размера 1 сразу после того, как текущая выборка выходит из временно́го окна. На рис. 7.7 показана стратегия формирования приоритетной выборки для шести элементов, прибывающих в указанные моменты времени $t_i s$ для размера окна 600 мс.

Мы постепенно объясним происходящее на рис. 7.7, поэтому рекомендуется держать его перед собой во время чтения. Первый элемент поступает через 100 мс, и p_{100} извлекается равным 0.3. Элемент e_{100} становится текущей выборкой, тогда как список с преемниками остается пустым. В момент 300 мс, когда прибывает элемент e_{300}, его приоритет, p_{300}, устанавливается равным 0.2. Поскольку он меньше приоритета текущей выборки, он добавляется в список в качестве первого элемента со своим приоритетом и временем прибытия. По прибытии элемента e_{550} он разрывает список приоритетов там, где начинаются элементы с более низким приоритетом ($p < p_{550} = 0.25$). Это приводит к удалению элемента e_2 из списка. Новое содержимое списка – это только элемент e_{550} со своим приоритетом и временем прибытия. Такая вероятностная обрезка списка приоритетов не дает ему становиться слишком большим.

Поскольку $p_{550} < p_{100}$, новый элемент не заменяет текущую выборку; он просто становится ее новым и единственным преемником на данный момент. В момент времени 700 мс ни один элемент не прибывает, но поскольку элемент e_{100} истекает, мы должны заменить текущую выборку ее преемником. Элемент e_{550} становится текущей выборкой, а список пуст. Через 1000 мс прибывает новый элемент. Его приоритет равен 0.5, который выше приоритета текущей выборки. Это приводит к удалению элемента e_{550} и его замене на элемент e_{1000} с указанием его приоритета и времени прибытия.

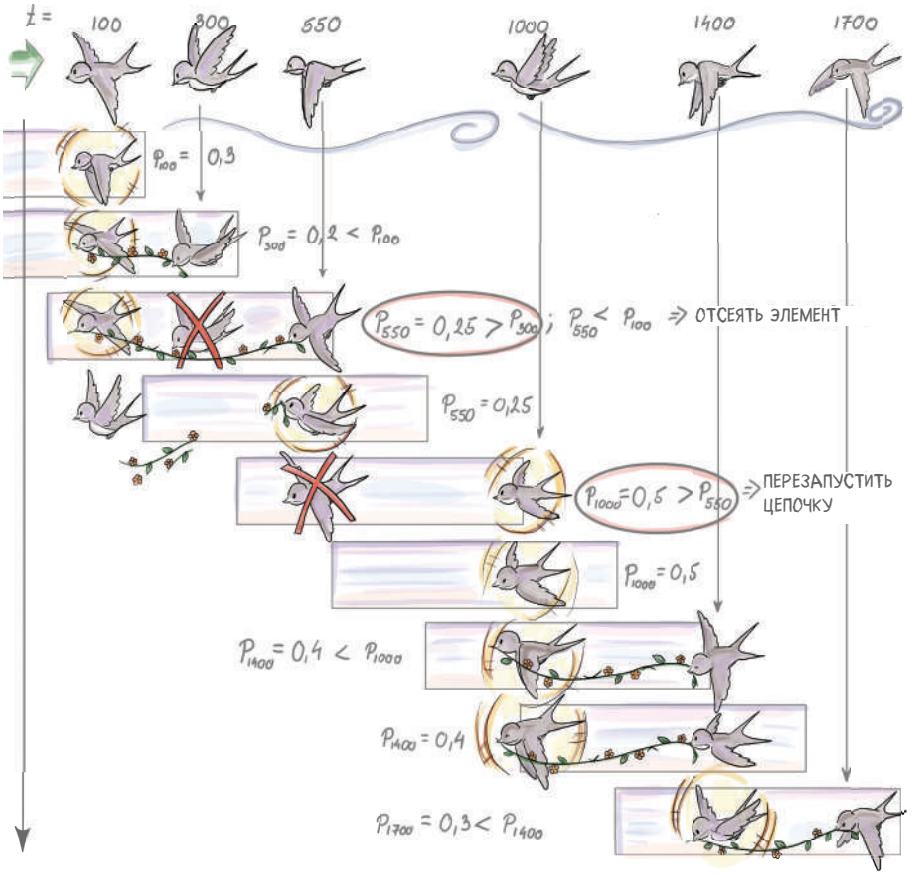


Рисунок 7.7 Содержимое списка преемников и текущая выборка для первых шести времен прибытия (мс) для окна длиной 600 мс, основанного на временны́х метках

Список преемников остается пустым. Элемент e_{1400} не имеет более высокого приоритета, чем у текущей выборки; следовательно, он добавляется в список как пока единственный элемент. В момент 1600 мс текущая выборка e_{1000} истекает и наследуется e_{1400}. Теперь список снова пуст. По прибытии элемента e_{1700} его приоритет устанавливается равным 0.3, поэтому он ниже, чем e_{1400} = 0.4. Таким образом, элемент e_{1700} добавляется в список в качестве первого преемника выборки e_{1400} по ее истечении.

Псевдокод и подробные комментарии к алгоритму формирования приоритетной выборки для выборки размера 1 показаны ниже. Это решение будет работать, если время между прибытиями любых двух элементов потока всегда меньше ω, длины окна:

```
L = []                                    ❶
i = 1                                     ❶
p = 0                                     ❶
```

```
while True:
    if len(L) == 0
        p = PRNG_unif(0,1)                    ❷
        t = t_i
        L.append([e_ti, p, t_i])
    else if (t_i - t ≥  ω )
        L = L.pop(1)                          ❸
        p = PRNG_unif(0,1)
        if p ≥ L[1][2]                        ❹
            L.clear()                         ❺
            t = t_i                           ❺
            L.append([e_ti, p, t_i])          ❺
        else                                  ❻
            j = 0
            while p ≤ L[j][2]
                j+=1                          ❼
                L = L[0:j]                    ❽
                L.append([e_ti, p , t_i])     ❽
    i = i + 1                                 ❾
```

❶ В начале L пуст. Задать индекс i текущего момента времени равным 1. Задать p, приоритет, перед появлением каких-либо элементов равным 0
❷ Обработать первый элемент
❸ Если текущий элемент выборки истек в момент времени t_i, то удалить первый верхний элемент списка
❹ Имеет ли только что прибывший элемент более высокий приоритет, чем первый элемент в списке?
❺ Очистить список и добавить новый элемент в качестве единственного элемента списка. Обновить время t текущей выборки
❻ Мы должны разбить список приоритетов где-то под самым первым элементом
❼ Найти место, где разбить список, и отбросить «хвост»
❽ Отбросить хвост и добавить текущий элемент на его место
❾ Перейти к следующей временной метке (времени прибытия)

Ожидаемое число хранящихся в памяти элементов в этой стратегии в любой момент времени равно O(ln n). Для поддержания выборки размера k можно содержать k списков, назначать k приоритетов, $p_{t_1}, p_{t_2}, p_{t_3}, \ldots, p_{t_{ik}}$ каждому прибывающему элементу e_{t_i} и повторять алгоритм с e_{t_i} столько раз, сколько имеется списков. Ожидаемая стоимость памяти в этом алгоритме равна O(k log n), хотя, с высокой вероятностью, стоимость не превышает O(k log n) (анализ пространственной сложности можно найти в тех же справочных материалах, что и по теме формирования цепной выборки).

Обратите внимание, что алгоритм с k списками, доставляющий выборку размера k, генерирует простую случайную выборку с возвратом из временно́го окна.

Для того чтобы опробовать процедуру формирования выборок из потока на практике, перед тем как перейти к фактической реализации алгоритма формирования выборки, сначала нужен фреймворк для обработки пото-

ков данных как объектов. Конфигурирование такой среды с использованием низкоуровневых функций ОС или даже с применением специальных библиотек R или Python для взаимодействия с потоковым фреймворком, таким как Apache Kafka, может занимать довольно много времени, в особенности если вы просто пытаетесь оперативно проверить надлежащую работоспособность вашего потокового алгоритма.

В следующем далее разделе мы покажем, как использовать эти алгоритмы на языке программирования R в рамках простого фреймворка обработки потоковых данных. Мы избавим себя от черновой работы благодаря пакету R `stream` и при этом еще сошлемся на аналогичный фреймворк на языке Python.

7.3 Сравнение алгоритмов формирования выборок

Теперь, когда мы познакомились с несколькими алгоритмами формирования выборок из потока, мы продемонстрируем способы применения некоторых из них на языке программирования R и, в частности, с помощью пакета R `stream` [8]. Он предоставляет фреймворк для обработки потоков данных с использованием *объектов-данных потоков данных* (DSD-объект)[66]. Они могут быть обертками реального потока данных, резидентных либо дисковых данных, или же генератора, который симулирует поток данных с известными свойствами при проведении контролируемых экспериментов. Определившись с данными, которые мы будем получать из DSD-объекта, мы реализуем задание. В нашем случае оно будет заключаться в поддержании случайной выборки из потока и ее использовании для оценивания среднего значения. Для этого мы будем использовать класс-*задание на обработку потока данных* (DST)[67].

7.3.1 Настройка симуляции: алгоритмы и данные

Мы сравним эффективность адаптации стратегий формирования смещенной и несмещенной выборок к внезапным и постепенным изменениям в потоке данных. Мы сгенерируем поток с внезапным сдвигом в концепции, чтобы проверить надежность алгоритмов формирования выборки по отношению к этой характеристике потока. Указанные два алгоритма оперируют на реперных потоках, поэтому можно говорить о случайной выборке размера k из того, что появилось к настоящему моменту. Стратегия формирования смещенной резервуарной выборки придает больший вес недавно появившимся элементам, а функция смещения и параметр лямбда, в частности, определяют скорость устаревания более старых элементов.

[66] Англ. data stream data (DSD) object. – *Прим. перев.*
[67] Англ. data stream task (DST) class. – *Прим. перев.*

В целях симуляции внезапного изменения в концепции мы создаем поток с помощью функции DSD_Gaussians(). Этот генератор нормально распределенных данных создает 10^6 гауссовых девиатов. Наблюдения из потока данных изменяют свое распределение с $N(1, 1)$ на $N(3, 1)$ за один шаг. Это означает, что источник потока симулирует внезапный сдвиг в одной точке. Для этой цели мы разбиваем поток пополам и получаем 500 К случайных значений из $N(1, 1)$, за которыми следуют 500 К случайных значений из $N(3, 1)$. Мы опробуем два размера выборок (резервуара), $\in \{10^4, 10^5\}$.

Сначала создаем поток, а затем сохраняем его в виде CSV-файла. Это делается для того, чтобы можно было отбирать одни и те же данные с помощью двух алгоритмов:

```
rm(list=ls())                                                          ❶
if (!'stream' %in% installed.packages()) install.packages('stream')    ❶
library(stream)                                                        ❶

setwd(" ")                                                             ❷
set.seed(1000)                                                         ❸
stream_FirstHalf <- DSD_Gaussians(k = 1,                               ❹
                                  d = 1,
                                  mu=1,
                                  sigma=c(1),
                                  space_limit = c(0, 1)
                                  )

write_stream(stream_FirstHalf, "DStream.csv", n = 500000, sep = ",")   ❺

stream_SecondHalf <- DSD_Gaussians(k = 1,                              ❻
                                   d = 1,
                                   mu=3,
                                   sigma=c(1),
                                   space_limit = c(0, 1)
                                   )

write_stream(stream_SecondHalf, "DStream.csv", n = 500000,
             sep = ",", append=TRUE)                                   ❼
```

❶ Удалить из рабочей области возможно оставшиеся объекты. Если пакет stream еще не установлен, то установить его. Затем привязать пакет к рабочей области
❷ Выбрать путь, по которому сохранять файл DStream.csv с данными
❸ Задать начальную позицию генератора псевдослучайных чисел таким образом, чтобы каждый раз создавались одни и те же случайные данные
❹ Создать DSD-объект для первой половины потока
❺ Записать 500 К элементов из DSD-объекта в файл DStream.csv
❻ Создать DSD-объект для второй половины потока
❼ Добавить 500 К элементов из DSD-объекта в файл DStream.csv

Реализации представленных в этой главе алгоритмов формирования смещенной и несмещенной резервуарных выборок доступны в пакете

stream в виде функции DSC_Sample(). В симуляциях используется два разных размера выборок: 10 K и 100 K элементов. Мы загружаем поток из файла, используя класс DSD_ReadCSV. Поток обрабатывается пакетами по 100 K элементов; следовательно, весь поток обрабатывается за 10 шагов. На каждом шаге вызывается функция update(CurrentSample, stream_file, n=100000). Она ожидает DST-объект, DSD-объект и число новых читаемых из потока элементов. В основе функции update() лежит парадигма задания, которое мы исполняем на потоке. В нашем случае это формирование выборки из потока. Прочитав 100 K новых элементов из потока, необходимо соответствующим образом скорректировать текущую выборку. Следовательно, вызов update() обеспечивает интеграцию новых 100 K элементов объектом-выборкой в свое текущее состояние. Поскольку мы вызываем update() 10 раз, у нас есть 10 снимков выборки, каждый после появления дополнительных 100 K элементов. На этих 10 остановках вычисляется и сохраняется среднее значение текущей выборки. В дальнейшем мы будем их использовать для оценивания того, насколько хорошо процедуры формирования смещенной и несмещенной резервуарных выборок приспосабливают свои средние значения к внезапному сдвигу в среднем значении потоковых данных. Этот сценарий повторяется для стратегий формирования смещенной и несмещенной резервуарных выборок с размерами выборок 10 K и 100 K. Вспомните, что коэффициент скорости старения, λ, в рамках стратегии формирования смещенной резервуарной выборки обратно пропорционален размеру выборки:

```
rm(list=ls())
if (!'stream' %in% installed.packages()) install.packages('stream')

stream_file <- DSD_ReadCSV("DStream.csv")                              ❶
CurrentSample <- DSC_Sample(k=10000, biased=FALSE)                     ❷

MeanResults_Size10K <- NULL                                            ❸

for(i in seq(1,10)){
        update(CurrentSample, stream_file, 100000)                     ❹

        names(CurrentSample$RObj$data) <- "sample_so_far"              ❺

        current_sample_avg <-
          mean(as.numeric(CurrentSample$RObj$data$sample_so_far))      ❻

        MeanResults_Size10K <- c(MeanResults_Size10K, current_sample_avg) ❼
}

reset_stream(stream_file, pos=1)                                       ❽

CurrentSample<-DSC_Sample(k=100000, biased=FALSE)
```

```
MeanResults_Size100K<-NULL
for(i in seq(1,10)){
  update(CurrentSample, stream_file, 100000)
  names(CurrentSample$RObj$data) <- "sample_so_far"
  current_sample_avg <-
      mean(as.numeric(CurrentSample$RObj$data$sample_so_far))
  MeanResults_Size100K<-c(MeanResults_Size100K, current_sample_avg)
}
close_stream(stream_file)                                          ❾
```

❶ Создать DSD-объект stream_file из файла DStream.csv
❷ Создать DST-объект, который реализует стратегию формирования резервуарной выборки с опцией biased, установленной равной FALSE, и размером выборки, равным 10 K
❸ Пустой вектор для хранения 10 средних значений из 10 поочередных снимков выборки
❹ Обновить выборку, добавив 100 K новых элементов
❺ Переименовать переменную, в которую объект-выборка сохраняет выборку, во что-нибудь более информативное
❻ Вычислить среднее значение выборки
❼ Сохранить среднее значение выборки в векторе
❽ Сбросить поток для стратегии формирования несмещенной резервуарной выборки с размером выборки 100 K
❾ Закрыть DSD-объект

`CurrentSample` – это объект класса `DSC_Sample`, который является подклассом класса-*задания на обработку потока данных* (DST). Таким образом, класс `DSC_Sample` можно использовать для реализации любой стратегии формирования выборки в качестве задания на потоке данных. Исходный код смещенной версии формирования резервуарной выборки идентичен, при этом параметр `biased` получает значение `TRUE`. Параметр λ, определяющий смещение в сторону вновь прибывших, в этом случае равен $1/k$.

На рис. 7.8 показаны средние значения выборки в ходе эволюции потока для этих двух стратегий формирования выборок. Мы видим динамику изменения среднего значения выборки для стратегий формирования резервуарной выборки и смещенной резервуарной выборки, а также размеры резервуара $k = 10^4, 10^5$ на реперном потоке с внезапным сдвигом в точке $i = 500$ K.

Мы видим, как стратегия формирования смещенной резервуарной выборки из-за неравного взвешивания недавнего и отдаленного прошлого адаптируется к внезапному сдвигу и оценивает среднее значение очень быстро и несмещенно после сдвига, тогда как стратегии формирования несмещенной резервуарной выборки это сделать не удается. Быстрота, с которой стратегия формирования смещенной выборки может обнаруживать сдвиг, зависит от параметра λ. При $\lambda = 10^{-4}$ вероятность остаться в выборке уменьшается на e^{-1} каждые 10 000 элементов, поэтому параметр $\lambda = 10^{-4}$ забывает быстрее или имеет меньший охват прошлым. Следовательно, смещенная выборка с $\lambda = 10^{-5}$ медленнее приближается к текущему

истинному среднему значению. Будем надеяться, что эта симуляция дала вам некоторое представление о реалистичных условиях, в которых вы будете брать выборки из потоков, и о решениях, которые придется принимать, учитывая имеющиеся данные.

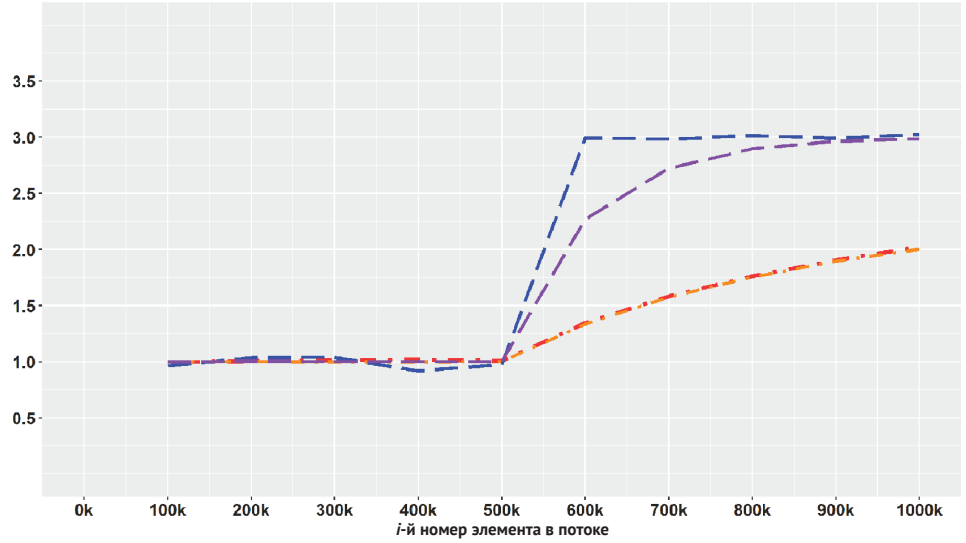


Рисунок 7.8 Пунктирными линиями соединены средние значения выборок, взятых с использованием алгоритма формирования смещенной резервуарной выборки. Верхняя, с $\lambda = 10^{-4}$, и нижняя, с $\lambda = 10^{-5}$, пунктирные линии отслеживают средние значения выборки на каждых 100 K новых элементах для двух размеров несмещенных резервуарных выборок

Для тех из вас, кто хотел бы попробовать формировать выборки из потоков с помощью Python, есть два варианта. Первый – более облегченный и позволяет развертывать простого производителя Kafka на базе Python, который читает данные с временными метками из CSV-файла. Соответствующий репозиторий находится на GitHub в рамках лицензии MIT (**github.com/mtpatter/time-series-kafka-demo**). Второй – библиотека Faust для сборки потоковых приложений на Python (**faust.readthedocs.io/en/latest**). Это очень хорошо задокументированная и богатая библиотека, лучше подходящая для обработки потоков данных производственного уровня. Если вы просто хотите конвертировать CSV-файл данных с временными метками в реально-временной поток, пригодный для тестирования вашего алгоритма формирования выборки, то этот вариант будет излишним.

Резюме

- Мы познакомились с пятью алгоритмами формирования выборки из потока данных (три для реперных потоков и два для оконных потоков). Формирование выборки Бернулли – это очень простой алгоритм

формирования репрезентативной выборки, но если вы хотите его использовать, то вам придется подумать о какой-то стратегии ограничения размера выборки, не привнося большого смещения.

- Стратегия формирования резервуарной выборки решила нашу проблему с переменным размером выборки и обеспечила простую случайную выборку из элементов, появляющихся в любой момент. Если мы хотим подчеркнуть более свежие элементы в выборке, то одним из способов является взятие смещенной резервуарной выборки, но нам нужно подумать о желаемой скорости устаревания и о том, как это соотносится с имеющимся пространством. Все зависит от реалистичных параметров, с которыми вы сталкиваетесь в своем собственном приложении.
- Другой способ подчеркивать недавно прибывающие в поток элементы – использовать скользящее окно. Мы научились реализовывать стратегии формирования цепной выборки для окон, основанных на последовательности, и формирования приоритетной выборки для окон, основанных на времени, если требуется брать выборки из этого окна по его размеру.
- Мы увидели, как в рамках симуляции стратегия формирования смещенной резервуарной выборки сумела подстроиться к внезапному сдвигу в концепции, тогда как обычная стратегия формирования резервуарной выборки отреагировать на это изменение не смогла и привела к смещенному ответу на запрос о недавнем истинном среднем значении потоковых данных. Напомним, что вам придется корректировать параметр скорости устаревания таким образом, чтобы он соответствовал понятию достаточной свежести в вашем конкретном случае использования.

Глава 8

Приближенные квантили на потоках данных

Эта глава охватывает следующие ниже темы:

- рассмотрение понятия точных квантилей и понимание ограничений, налагаемых контекстом обработки потоковых данных;
- понимание различных типов ошибок для приближенных квантилей;
- применение алгоритмов t-дайджеста и q-дайджеста к потоку данных;
- сравнение t-дайджеста и q-дайджеста на реалистичных данных о продолжительности посещений веб-сайта.

Представленные в предыдущей главе разные алгоритмы позволяют формировать (не)смещенную выборку из всех кортежей данных, прибывших на текущий момент. В некотором смысле выборка – это очень гибкий набросок данных: она формируется один раз, а затем с ее помощью можно заявлять, что ее среднее значение или любая другая характеристика является хорошей оценкой этой же характеристики у всех данных, прибывших из потока на текущий момент. Вспомните причину, по которой мы перешли от формирования выборки Бернулли к процедурам формирования выборок, поддерживающих выборку фиксированного размера: выборка Бернулли растет вместе с увеличением числа прибывающих элементов, и поэтому в сочетании с потоковыми данными она просто непрактична.

Вместе с тем, беря выборку Бернулли, получаешь то, чего не получаешь другими методами, – постоянную пропорцию отбора[68] при любом значении N. Ситуация, при которой берется в среднем каждый $1/p$-й элемент, не изменится независимо от величины N. За это приятное свойство приходится «платить» наличием среднего размера выборки pN. Почему это

[68] Англ. sampling rate; в статистике это пропорция популяции, попавшая в выборку (не путать с частотой отбора/дискретизации в обработке цифровых сигналов). – *Прим. перев.*

хорошо в условиях, когда размер выборки увеличивается? Центральная предельная теорема гласит, что любой оценщик на основе нашей выборки имеет стандартную ошибку (прецизионность), которая уменьшается вместе с квадратным корнем из размера выборки. Таким образом, статистически неплохо иметь более высокий размер выборки. Проблема в том, что по мере того, как мы видим все больше и больше элементов из потока, плотность выборки при отборе методом Бернулли не меняется, но меняется при других. В алгоритмах, которые поддерживают фиксированный размер выборки, равный k, плотность выборки неизбежно уменьшается, поскольку мы всегда имеем плотность k/N по мере роста N.

Эта плотность в широком смысле служит мерой распределенности взятых в выборку точек по всем точкам в потоке. (Если вас интересует более формальная трактовка данного понятия «плотности» выборки, прочтите введение к статье по ссылке: **https://arxiv.org/pdf/2004.01668v1.pdf**.)

Описанные в этой главе алгоритмы призваны «увязать» ограничения на конечный размер с конкретным понятием *постоянной плотности* (или прецизионности) по мере роста N, чтобы отвечать на запросы о *приближенных квантилях*. Хотя, возможно, это покажется волшебством, но это вполне достижимо в условиях допущений, которые не влияют на практическую важность алгоритмов.

8.1 Точные квантили

Бесперебойное присутствие в интернете, постоянный контакт и обслуживание потребителей сегодня имеют для компаний первостепенное значение. Именно поэтому компании и организации вкладывают значительные средства в обеспечение постоянного наличия контента и доступа к своим веб-сайтам. Одной из характеристик, представляющих интерес для действующего веб-сайта, является время, которое пользователь проводит на нем. Исходя из этих данных, можно определять среднее время, которое пользователь проводит на веб-сайте. А затем по некоторому стабильному усредненному профилю можно определять патологические примеры в данных о проведенном на веб-сайте времени.

Мы просимулировали немного данных, которые отслеживают распределение реальных данных, описанных и показанных на сайте Apache DataSketches (**http://mng.bz/gwoG**). Приведенные там данные показывают реальные данные, извлеченные с одного из их внутренних серверов. Эти данные представляют один час проведенного на веб-сайте времени в миллисекундах. Данные в изначальной шкале имеют очень длинный правый хвост; следовательно, отображать их на одной миллисекундной шкале в виде обычной гистограммы не посоветовал бы ни один офтальмолог. Такие данные лучше отображать в виде гистограммы, как показано на рис. 8.1. Фактическая ширина корзин увеличивается по мере продвижения слева направо. Тем не менее мы рисуем их в виде полос одинаковой ширины. Благодаря такому подходу мы имеем возможность наслаждаться визуаль-

ным представлением. Обратите внимание, что технически это столбчатая диаграмма, а не гистограмма.

При каждом посещении веб-сайта пользователем внутренние серверы, на которых он размещен, регистрируют начало и конец посещения. Данные показывают продолжительность пребывания в миллисекундах примерно для 26 млн посещений. У большой доли, около 14 %, проведенное время зарегистрировано как 0.

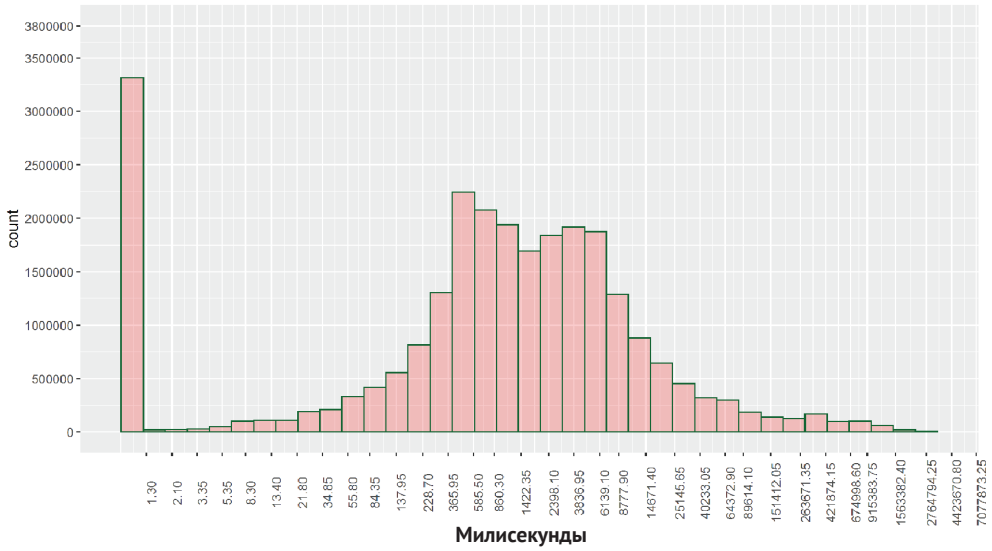


Рисунок 8.1 Столбчатая диаграмма (подтасованная гистограмма), показывающая продолжительность пребывания в миллисекундах примерно для 26 млн посещений

На данный момент нас не волнует, откуда эти данные поступили: из потока либо из базы данных, но нас могут заинтересовать ответы на следующие ниже вопросы:

- Какова медианная продолжительность пребывания на веб-сайте, размещенном на одном внутреннем сервере?
- Каков 95-й процентиль продолжительности пребывания на веб-сайте, размещенном на одном внутреннем сервере?
- Каков 95-й процентиль продолжительности пребывания на веб-сайте, размещенном на всех внутренних серверах?

Предназначение таких запросов совершенно очевидно: знать ситуации, когда *хватка* веб-сайта начинает ослабевать или когда она становится помехой в виде длительных задержек сервера (которые, конечно же, можно запрашивать аналогичным образом напрямую). Это может быть выявлено по движению медианы во временной динамике (или, если уж на то пошло, по любому другому квантилю, отслеживать который, по вашему мнению, имеет особое значение, чтобы оптимизировать какой-либо бизнес-процесс).

Что это за штука такая, *квантиль*, которая может провести наш корабль принятия решений через бурлящее море данных? Неудивительно, что понятие квантиля прижилось в человеческом коллективном опыте задолго до появления любых больших данных. Следовательно, на нем остался след запачканных мелом нарукавников. Другими словами, вам придется смириться с греческими буквами и теорией, чтобы разобраться в понятии. Теоретический квантиль φ (фи) некоторого (непрерывного) распределения вероятностей (плотности $f(x)$) является обратной функцией кумулятивного распределения $F(x)$:

$$\varphi_x = F(x) = P(X < x) \Leftrightarrow F^{-1}(\varphi_x) = x.$$

Теперь проиллюстрируем это на примере наших данных о затраченном времени. Будет легче, если вставить обсуждаемые нами значения в выражение. Если принять $\varphi_x = 0.5$, то тогда мы захотим знать, ниже какого значения затраченное время x составляет 0.5 (половину) от всех зарегистрированных длительностей посещений. Иными словами, половина всех посещений будет короче указанной длины x. Это и есть медиана данных. Любой φ-квантиль данных о продолжительности пребывания на нашем веб-сайте вычисляется путем их сортировки, а затем выбора φN-го элемента в сортированной последовательности. Например, при 25 961 440 посещениях медианная продолжительность пребывания равна 12 980 721-й записи в сортированной последовательности. Эта запись равна 1150.592 миллисекунды (R(1150.592) = 12 980 721). Последнее выражение в круглых скобках читается как ранг 1150.592 миллисекунды, равный 12 980 721-му.

Мы бы поступили аналогичным образом с вычислением любого другого точного квантиля (например, 95-й процентиль имеет 95 % данных «ниже» себя). Задача вычисления квантилей в информатике известна как задача о *сортировке и отборе*; по самому названию можно предположить, что она имеет долгую историю интеллектуальных усилий и научных исследований.

Отыскание минимума или максимума (соответственно рангов 1 и N) нуждается только в линейном времени и постоянном дополнительном пространстве. То же самое касается рангов, которые находятся вблизи границ данных (то есть отыскать любой ранг, который находится на постоянном удалении от минимума или максимума, немногим сложнее; например, ранг c или $N - c$ для некой константы c). Для отыскания других, менее тривиальных рангов, можно легко использовать сортировку, где после сортировки массива $A[0, \ldots N - 1]$ за $O(N \log N)$ мы находим ранг r, обращаясь к элементу $A[r - 1]$ массива (то есть за постоянное время). Высокая цена сортировки окупится, если данные в значительной степени статичны и мы ожидаем выполнения большого числа ранговых запросов. Однако если требуется найти только несколько тех или иных рангов и/или данные часто изменяются, то сортировка становится дорогостоящим хобби.

И действительно, для того чтобы найти любой априорно фиксированный ранг в несортированном наборе данных, достаточно потратить $O(N)$. Детерминированный *алгоритм медианы медиан* с временной сложностью

наихудшего случая O(N), разработанный Блюмом, Флойдом, Праттом, Ривестом и Тарьяном (Blum, Floyd, Pratt, Rivest, Tarjan, аббр. BFPRT) [3], работает рекурсивно, разбивая данные на группы размером 5, беря медианы каждой из $\lfloor n/5 \rfloor$ групп (линейно-временная операция!) и рекурсивно повторяя на медианах до тех пор, пока не останется один элемент. Затем этот элемент используется в качестве высококачественной опорной точки и вводится в алгоритм быстрого отбора[69] (очень напоминающий быструю сортировку), который перестраивает данные вокруг опорной точки и выполняет рекурсию на стороне ранга r. С учетом того, что нам предоставлены высококачественные опорные точки из рекурсивной схемы BFPRT (которая удобно разбивает данные на две равные доли), алгоритм быстрого отбора выполняется за O(N).

В этот момент вы, возможно, скажете: «Так вот же наш алгоритм; почему бы просто не использовать его?» Предостережение от такой идеи – полученный Манро (Munro) и Патерсоном (Paterson) [2] результат столь же классический, как и сам 1980 год. Они показали, что любой алгоритм, который точно вычисляет медиану, используя по меньшей мере p прохождений по данным, требует как минимум $O(N^{1/p})$ рабочей памяти. Определить p, с которым придется работать в условиях обработки потоковых данных, очень легко. А именно мы получаем всего один проход по данным. Это означает, что нам требуется память, размер которой линейно зависит от размера входных данных. Этот отрезвляющий результат должен заставить нас смириться с некоторой ошибкой ε при оценивании φ_x на потоковых данных.

8.2 Приближенные квантили

Теперь, когда мы знаем, что получить точные квантили в условиях ограничений на потоковые данные невозможно, можно, в стиле всего наполовину полного стакана[70], поговорить об ошибке. Алгоритмы, разработанные для таких условий, всегда должны давать какие-то гарантированные границы ошибки. Если уж на то пошло, все алгоритмы, вычисляющие приближенные ответы, должны иметь такие границы. Существует три типа ошибок, с которыми можно столкнуться, если пролистать (не)опубликованные исследования в этой области:

- аддитивная ошибка аппроксимации ранга;
- относительная (мультипликативная) ошибка аппроксимации ранга;
- относительная ошибка в фактической области значений данных.

8.2.1 Аддитивная ошибка

Большинство алгоритмов, разработанных для этой задачи, работают так, чтобы гарантировать фиксированную аддитивную ошибку εN в аппроксимации ранга *для любого* $\varphi \in [0,1]$. Здесь N – это число элементов,

[69] Англ. quick-select algorithm. – *Прим. перев.*

[70] То есть оптимистично. – *Прим. перев.*

появившихся на данный момент. Это подводит нас к ε-*приближенному* φ-квантилю. Под ним подразумевается, что если запросить квантиль $\varphi_x \in [0,1]$, то мы всегда будем получать элемент z с рангом $R(z) \in [\varphi N - \varepsilon N, \varphi N + \varepsilon N]$. Обозначение φ_x подразумевает, что ϕ квантиль данных на самом деле равен x, а не z, отсюда и граница ошибки вокруг φN. Более тщательный анализ границы ошибки показывает, что z «обещает» от имени своего ранга $R(z)$ быть не дальше, чем εN, от истинного ранга φN элемента x, который нас действительно интересует, но который мы не смогли увидеть. С этой гарантией на $R(z)$ для любого возвращенного z можно по меньшей мере положиться на

$$|\varphi N - R(z)| \leqslant \varepsilon N.$$

Если допустить в алгоритме некую случайность, то эта граница ошибки все равно должна соблюдаться, за исключением малой вероятности неуспеха δ (дельты). Разработчики недетерминированных алгоритмов предоставляют вероятностное доказательство того, что при некоторых ослабленных допущениях алгоритм не будет обманывать вас слишком часто. Вот откуда взялась δ.

Обратите внимание на два важных следствия такого определения ошибки аппроксимации:

- ошибка измеряется в единицах ранга, а не в единицах базовой области значений данных;
- ошибка постоянна для фиксированного N, но поскольку для потоковых данных N будет увеличиваться с каждым новым прибытием, допустимая фактическая ошибка для ε-приближенного φ квантиля также будет увеличиваться в абсолютном смысле.

Это то, о чем следует помнить. В качестве иллюстрации понятия аддитивной ошибки мы будем использовать набор длительностей в миллисекундах для 10 посещений веб-сайта:

55.3, 43.1, 70.4, 64.6, 52.3, 72.4, 89.2, 82.6, 67.7, 95.6.

Сначала этот набор сортируется:

43.1, 52.3, 55.3, 64.6, 67.7, 70.4, 72.4, 82.6, 89.2, 95.6

для $\varepsilon = 0.1$, $x = 50$ и $R(x) = 2$. Тогда все легальные ранги находятся в интервале $[R(x) - 0.1 \times 10, R(x) + 0.1 \times 10] = [1,3]$. При возврате 1, 2 или 3 в качестве ранга в 50 миллисекунд соблюдается граница аддитивной ошибки. Тогда ε-приближенный квантиль 0.1 может составлять 43.1 или 52.3. Обратите внимание, что абсолютные ошибки по шкале наших фактических данных, измеряемых в миллисекундах, равны $|43.1 - 50| = 6.9$ и $|52.3 - 50| = 2.3$ миллисекунды.

Упражнение 1

Продолжим пример с аддитивной ошибкой. Если мы получим 90 новых элементов (в дополнение к 10, которые мы использовали для иллюстрации понятия) через поток данных, то это увеличит размер набора до 100 (для простоты и воспроизводимости предположим, что все они больше 95.6). Какими теперь будут приближенные ранги и ε-приближенные квантили 0.1, которые соблюдают границу аддитивной ошибки?

8.2.2 Относительная ошибка

Под названием *относительная ошибка* вы найдете следующее ниже определение ошибки, которую несет с собой z:

$$|R(x) - R(z)| \leqslant \varepsilon R(x).$$

Слово *относительный* здесь связано с тем фактом, что ошибка пропорциональна фактическому рангу, который вы хотите оценить, а не числу элементов, которые вы увидели.

Единственным рангом, прецизионность которого не изменяется между относительной и аддитивной ошибками, очевидно, является максимальный: $x_{\max}(R(x_{\max}) = N)$. ε-приближенный квантиль для минимума допустим только на расстоянии ε от истинного ранга 1. Давайте еще раз посмотрим на сортированный пример данных:

43.1, 52.3, 55.3, 64.6, 67.7, 70.4, 72.4, 82.6, 89.2, 95.6.

Если мы захотим запросить 0.1-приближенный минимум этих данных, ($R(x) = 1$ и $x = 43.1$), то наилучшим приближением будет 52.3 с его рангом 2. Это не соблюдает определение 0.1-приближенного квантиля в смысле относительной ошибки. Следовательно, 2 – это лучшее, что можно сделать, и недостаточно хороший результат, если мы хотим соблюсти границу относительной ошибки. При $\varepsilon = 0.1$ единственной порядковой статистикой, для которой мы могли бы установить границу относительной ошибки, является максимум. Вы можете это проверить.

Следовательно, гарантировать относительную ошибку в общем случае труднее, чем соблюдать аддитивную: тот же алгоритм ε, который соблюдает границу относительной (мультипликативной) ошибки, тривиально соблюдает границу аддитивной ошибки, но не наоборот.

Относительная ошибка важна для точного оценивания квантилей в хвостах распределения. Большинство данных, получаемых в онлайновом режиме, являются длиннохвостыми (данные о посещениях веб-сайта тоже). При $R(x) \ll N$ или $N - R(x) \ll N$ (соответствует левому и правому хвостовым квантилям) мы хотим, чтобы точность оценок была высокой, поскольку такие процентили, как 99-й, 99.5-й или 99.975-й, могут демонстрировать большие абсолютные разницы. Вспомните, как нам пришлось увеличивать ширину столбцов, когда мы углублялись в правый конец дан-

ных веб-сайта? И все из-за этой увеличивающейся разницы. В еще одном варианте использования, мониторинге сетевых задержек, несколько очень плохих периодов отклика могут вызывать массу проблем у части пользователей, которые могут выражать свое разочарование на странице своей любимой социальной сети. Даже если задержки, испытываемые большинством пользователей, невелики, это большинство молчит. Длиннохвостые данные могут иметь большую разницу между 99.5-м и 99.975-м процентилями в абсолютном выражении, скажем более 30 секунд. Поэтому хотелось бы иметь возможность более высокой достоверности оценки квантилей в хвостах распределения. И границы относительной мультипликативной ошибки справляются с замером этого необычного поведения в хвостах лучше, чем аддитивные представления об ошибке.

8.2.3 Относительная ошибка в области значений данных

Третий тип ошибок, с которыми вы можете столкнуться, – это относительная ошибка относительно значений ваших фактических элементов данных. Для квантиля φ с $R(x) = \varphi N$ мы хотим видеть элемент z такой, что соблюдается

$$|x - z| \leqslant \varepsilon x.$$

Понятие ошибки такого рода привязано к фактической шкале ваших данных и применимо только к числовым данным. Поскольку оба предыдущих определения ошибки определены относительно рангов, они могут использоваться для привязки ошибок к любым данным, которые могут быть упорядочены. Из-за этого недостатка общности не многие исследователи решаются разрабатывать алгоритмы, качество которых оценивается по этому типу ошибок.

Теперь, когда мы определились не только с тем, что всегда ошибаемся, но и с тем, насколько ошибаемся, можно задаться вопросом о том, как реализовывать наброски или дайджесты, которые будут помогать в этом деле механически.

8.3 T-дайджест: принцип его работы

Все алгоритмы вычисления приближенных квантилей, с которыми вы столкнетесь, представляют собой форму самоорганизующихся структур данных, реагирующих на распределение данных. Они будут называться *сводками*, *дайджестами*, *скетчами* или *набросками*. На очень высоком уровне они хранят некую малую порцию наблюдаемых данных с метаданными по каждому сохраненному элементу данных. Затем они используют эти данные для ответа на запрос о приближенном ранге элемента или, наоборот, для возврата (приближенного) элемента данных в ответ на тот или иной квантильный запрос.

Часто бывает так, что эффективный метод предлагается до того, как будет предоставлен своего рода дисклеймер в форме гарантий ошибки. Например, алгоритм случайного леса стал популярным и широко использовался (около 13 лет), прежде чем было доказано асимптотическое поведение (согласованность и стандартная ошибка) этого непараметрического оценщика. Первый алгоритм, который мы представим, происходит из такой эвристической части теоретической информатики. Как и любая другая хорошо зарекомендовавшая себя и эффективная эвристика, она была широко принята сообществом с момента ее презентации в 2013 году. Согласно официальной документации, t-дайджест используется во многих известных базах данных и фреймворках обработки потоковых данных / распределенных вычислений и библиотеках, таких как Apache Kylin, Apache Druid, Apache DataSketches, PostgreSQL и Elastic Search. По своей конструкции T-digest предлагает *эмпирические* относительные ошибки в квантильном пространстве, которые представляются более чем приемлемыми для широкого круга приложений. Однако с формальным доказательством границы ошибки пока не все ясно – коллегия присяжных заседателей все еще находится в совещательной комнате. Сначала дадим определение дайджеста.

8.3.1 Дайджест

Если вы знаете, с чего начинался знаменитый журнал «*Ридерз Дайджест*», то сможете здесь провести более возвышенную аналогию по сравнению с простой аналогией, связанной с обыкновенным перевариванием пищи. Эта штука должна потреблять некие данные, а затем решать, что сохранять, а что интегрировать посредством метаданных, перед тем как отбрасывать. Указанные метаданные и есть сама структура данных. Ее можно представить как серию равноудаленных больших сгустков вдоль оси миллисекунд, подобных кластерам, которые охватывают диапазон данных. В случае данных веб-сайта запрос о медиане касается данных, представленных кластерами ниже середины распределения. Число этих кластеров обычно задается до прибытия данных и впоследствии учитывается в пространственных требованиях алгоритма.

На рис. 8.2 показан дайджест с пятью кластерами для $N = 10$ элементов, прибывших в указанном порядке. У нас нет оснований полагать, что они будут поступать в каком-либо определенном порядке. Сначала мы разбиваем элементы данных на наборы π_i в соответствии с их поочередными, ненакладывающимися интервалами *индексов прибытия*. Это соответствует муравьиному уровню на рис. 8.2. π_1 равен {52.3, 72.4, 83.2} и охватывает индексы прибытия от 1 до 3. Указанные разделы называются кластерами, и для каждого мы вычисляем среднюю продолжительность посещения, именуемую *центроидом*, и число точек данных, влияющих на это среднее значение, именуемое *весом*. На рис. 8.2 это показано на первом листовом уровне. Затем мы сортируем кластеры в соответствии с их средним значением. Обратите внимание, что второй листовой уровень отсортирован по

размеру. В дополнение к этому для каждого кластера мы отмечаем сумму весов слева и справа от него. Теперь у нас есть дайджест, содержащий меньше информации по сравнению с изначально полученными данными.

Результирующую структуру можно использовать для ответа на запрос о $R(72.4)$. Мы нашли бы первое среднее значение равным или большим 72.4. Если бы мы добавили все веса слева от этого кластера, то получили бы некую оценку приближенного ранга. На рис. 8.2 мы вернули бы значение 8 и отклонились бы от истинного ранга 7 на 1.

Дайджест называется строго упорядоченным, если

$$i < j \implies x \leqslant y \text{ для } x \in \pi_i \quad y \in \pi_j.$$

Дайджест слабо упорядочен, если

$$i + \Delta < j \implies x \leqslant y \text{ для } x \in \pi_i \quad y \in \pi_j$$

для некоторого целого положительного числа $\Delta \geqslant 1$. На рис. 8.2 показан результирующий слабо упорядоченный дайджест. Обратите внимание, что 74.2 больше, чем 64.6, хотя центр тяжести кластера 64.6 больше, чем центр тяжести кластера, частью которого является 74.2. Следовательно, по определению, он не является строго упорядоченным. Таким образом, интуиция подсказывает, что кластеры, вероятно, стянуты не очень плотно вокруг своего центра тяжести и некоторые неоднозначные точки данных «плавают» между ними. Параметр Δ косвенно является мерой этой стянутости кластеров. Он определяет число кластеров, на которые i и j должны быть дистанцированы друг от друга, чтобы все элементы, представленные i-м кластером, были меньше, чем все элементы, представленные j-м.

Упражнение 2

Каков наименьший параметр Δ на рис. 8.2, который делает дайджест слабо упорядоченным (для этого конкретного Δ)? Есть ли такой? Другими словами, существует ли число кластеров, которые, если перепрыгнуть через все элементы из i-го кластера, будут меньше, чем все элементы в j-м?

В тривиальном плане ограничение размера кластера до 1 (выбирая одиночек[71] в качестве раздела) сделает результирующий дайджест строго упорядоченным. В основе алгоритма t-дайджеста лежит умный динамический выбор размеров кластеров. Далее мы увидим, как размеры кластеров управляются косвенно, путем их продуманного соотнесения с ширинами интервалов, подразделяющих квантильный диапазон [0,1].

8.3.2 Масштабные функции

Гениальная часть t-дайджеста заключается в динамическом обновлении размеров (связанных с весами, но не являющихся самими весами) кластеров по мере поступления новых данных. В целях пояснения мы *подадим* данные в t-дайджест в отсортированном возрастающем порядке. Это мож-

[71] Англ. singleton; син. одноэлементное множество, синглтон. – *Прим. перев.*

но сделать для малого, конечного (по отношению к доступной рабочей памяти) числа наблюдений, имея в распоряжении буфер рабочей памяти и сортируя наблюдения перед их вставкой в t-дайджест. Позже станет ясно, что это вовсе не ограничивающее допущение.

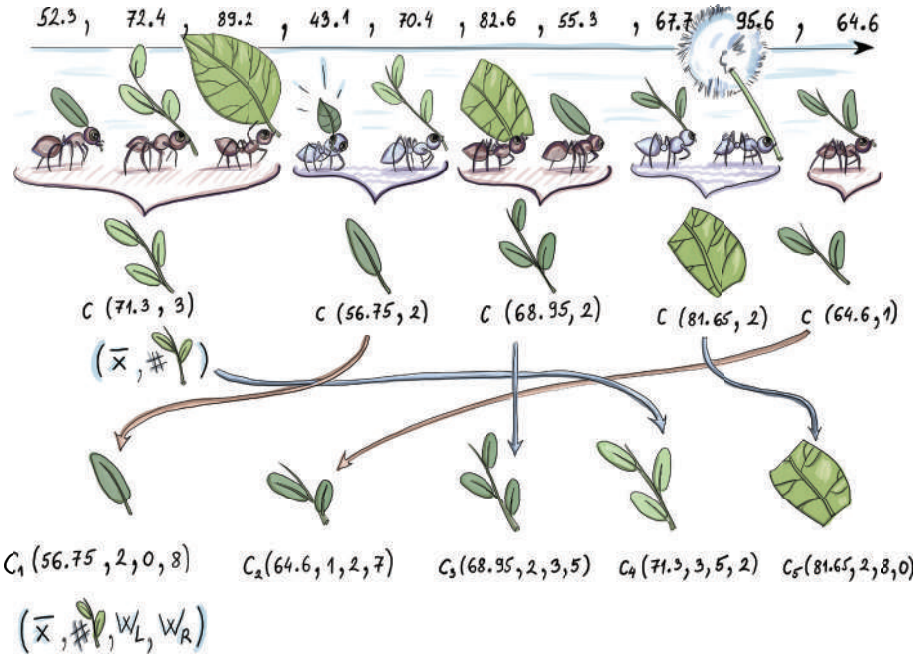


Рисунок 8.2 Дайджест для 10 элементов. Мы упорядочиваем кластеры в соответствии с их средним значением. Кластеры 1, 2, 3, 4 и 5 состоят соответственно из 3, 2, 2, 2 и 1 точек данных. Каждый из них, помимо числа элементов, также хранит среднее значение. W_{left} и W_{right} хранят число элементов данных слева и справа от каждого кластера. Обратите внимание, что этот дайджест не имеет строгого порядка

Ключевым рычагом здесь являются функции, которые определяют, какой кластер должен слиться со своим соседом и когда. В каждый момент времени распределение кластеров вдоль оси определяется масштабными функциями[72]. На рис. 8.3 показано, как кластеры подразделяют квантильный диапазон [0,1] для двух разных масштабных функций. В обоих случаях квантильный диапазон [0,1] покрывается, но ширины интервалов, относящихся к одним и тем же кластерам, между ними разные. Мы увидим причину, по которой один из них лучше другого для наших целей оценивания приближенного квантиля.

На рис. 8.3 кластеры росли в ширину многократной интеграции с соседними кластерами до тех пор, пока новый кластер, полученный в результате слияния, не достиг максимальной ширины. Этот процесс роста оста-

[72] Англ. scale function. – *Прим. перев.*

навливается, когда результирующий кластер выходит за границу размера, определяемого масштабной функцией. Хорошие масштабные функции не удерживают границу размера одинаковой для каждого кластера. Они ставят ее в зависимость от их позиции в подынтервале квантильного диапазона [0,1]. Другими словами, граница размера, выше которой кластеры больше не могут объединяться, зависит от того, где в квантильном диапазоне [0,1] предпринимается попытка слияния двух кластеров. Другое дело, если два сливающихся кластера находятся вблизи середины диапазона, а не на краях интервала. Следовательно, точный подынтервал квантильного диапазона [0,1], о котором кластер в каждый момент «сообщает», определяется масштабной функцией.

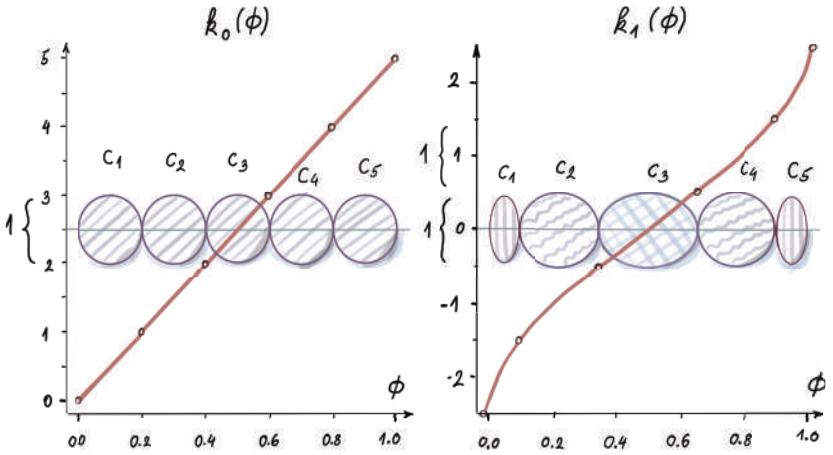


Рисунок 8.3 $\mathcal{K}^{k_1}$ и $\mathcal{K}^{k_0}$ различаются между k_1 и k_0. *k*-размеры в обеих функциях отводят пять кластеров (оба t-дайджеста показаны в виде упорядоченного набора полностью слившихся кластеров; никакие два поочередных кластера нельзя объединить без нарушения весовой границы). Кластеры *k*-размера в обоих случаях равны 1; тем не менее сообщаемые кластерами подынтервалы различны, и для k_1 они переменны, при этом меньшие кластеры расположены по краям, а большие – в середине. Для k_1 они сообщают о подынтервалах одинаковой ширины, равной [0,1]

Для этих целей предлагается несколько функций. В оригинальной статье о t-дайджесте (**https://arxiv.org/abs/1902.04023**) приведены следующие:

$$k_0(\varphi) = \frac{\delta}{2}\varphi;$$

$$k_1(\varphi) = \frac{\delta}{2\pi}\sin^{-1}(2\varphi - 1);$$

$$k_2(\varphi) = \frac{\delta}{4\frac{\log n}{\delta} + 24}\log\frac{\varphi}{1-\varphi};$$

$$k_3(\phi) = \begin{cases} \frac{\delta}{4\frac{\log n}{\delta}+21} \log 2\phi, & \text{если } \varphi \leq \frac{1}{2} \\ \frac{\delta}{4\frac{\log n}{\delta}+21}(-\log 2(1-\phi)) & \text{в противном случае.} \end{cases}$$

Здесь n – это текущее число полученных элементов данных. Они действительно выглядят загадочно, но от вас требуется понять лишь k_0 для фиксированного δ (это простая линия с наклоном δ/2). Вы можете просмотреть остальные, если вам интересно. Все они имеют одинаковую форму кривой, показанную на рис. 8.3. Для наших целей думайте только о форме.

Помните, мы говорили о размерах? Теперь мы объясним их зависимость от масштабной функции k и будем называть размер кластера C_i его k-размером $\mathcal{K}_i$. Как вы видите, все эти масштабные функции монотонно возрастают, и они определены для любого квантиля $\varphi \in [0,1]$. Мы будем использовать их для вычисления разностей $k(\varphi_{\text{right}}) - k(\varphi_{\text{left}})$ для (под)диапазонов $[\varphi_{\text{left}}, \varphi_{\text{right}}]$ в интервале [0, 1]. Например, первый кластер для масштабной функции, k_0, на рис. 8.3 покрывает [0, 0.2]. Его k-размер равен $k(0.2) - k(0) = 1 - 0 = 1$. Как вы видите, две разные функции k не имеют общей области значений.

k-размер $\mathcal{K}_i$ кластера C_i связан с шириной $\varphi^i_{\text{right}} - \varphi^i_{\text{left}}$ подынтервала, о котором C_i сообщает в данный момент. Границами подынтервала φ^i_{left} и φ^i_{right} являются

$$\varphi^i_{\text{left}} \frac{\mathcal{W}_{\text{left}}(C_i)}{n} f,$$

$$\varphi^i_{\text{right}} = \varphi^i_{\text{left}} + \frac{|C_i|}{n},$$

где $W_{\text{left}}(C_i) = \sum_{j<i} |C_i|$ – это сумма весов всех кластеров слева от i-го из упорядоченного набора. Тогда граница k-размера каждого кластера C_i равна

$$\mathcal{K}_i = k(\varphi^i_{\text{right}}) - k(\varphi^i_{\text{left}}) \leqslant 1.$$

Хорошо видно, что k-размер – это разница между k-значениями границ. Сразу после того, как только одиночный кластер достигает k-размера 1, он больше не может принимать ни соседних одиночек, ни соседних кластеров. Именно так масштабные функции управляют размерами кластеров. Будьте внимательны: на оси y рис 8.3 показаны k-значения, а не k-размеры. k-размеры вычисляются как разности между k-значениями (единицы скрываются за левыми фигурными скобками).

Асимметричная концепция, при которой кластеры вбирают в себя кластеры, реализована в t-дайджесте путем слияния кластеров (интеграции соседних кластеров в один). Именно так они увеличиваются в абсолютном

весе ($|C_j|$) и k-размере. В полностью слившемся t-дайджесте никакие два (соседних) кластера не могут сливаться, так как при этом нарушается их граница k-размера, равная 1:

$$\mathcal{K}_{i,i+1} = \mathcal{K}_i + \mathcal{K}_{i+1} > 1.$$

Допустим, что для масштабной функции k_1 вы выбираете параметр $\delta = 10$, и $k_1(0) = -10/4$ и $k_1(1) = 10/4$, как показано на рис. 8.3. Это означает, что k_1 простирается на 5 единиц по шкале k-размера, а его аргумент перемещается на один шаг, от 0 до 1.

Возможно, вы уже догадались, что это история о производной от k_1. На периферии интервала [0,1] k_1 изменяется быстрее, а затем где-то примерно в середине скорость изменения снижается до минимума. Там она становится линейной с постоянным наклоном (обратите внимание, что k_0 имеет такое постоянное наклонное поведение на всем интервале [0,1]). После этого она снова начинает набирать обороты и заканчивается с такой же большой скоростью изменения, как и в начале. Таким образом, ширина сообщаемого кластером подынтервала [0,1] обратно пропорциональна скорости изменения масштабной функции на этом подынтервале. Чем круче функция для конкретного подынтервала, тем меньше фактический размер кластеров. На «крутых» участках быстро изменяющаяся функция быстрее достигает максимального k-размера, равного 1, а кластеров становится много, но они малы.

Следовательно, как видно по рис. 8.3, на краях интервала размеры кластеров меньше, а в середине – больше. Также обратите внимание, что при k_0 все кластеры имеют одинаковый размер, независимо от подынтервала [0,1], в котором вы находитесь. Это обнаруживает границы минимального числа кластеров, поддерживаемых в любой точке. На рис. 8.3 показано минимальное число кластеров, которое можно получить при $\delta = 10$. Оно равно пяти; следовательно, минимальным является диапазон, покрываемый масштабной функцией, так как в пределах каждой единицы допускается не более одного кластера из-за границы k-размера, равного 1. Эти пять кластеров должны иметь максимальный размер.

Следовательно, k_1 предоставляет возможность «сужать поле зрения» до тех хвостов, которые близки к минимуму и максимуму данных, где происходит что-то необычное (например, в приложениях по обнаружению аномалий). Для квантилей φ, близких к 0 и 1, $k_1(\varphi_{\text{right}}) - k_1(\varphi_{\text{left}})$ округляется до 1 всякий раз, когда $k_1(\varphi_{\text{right}}) - k_1(\varphi_{\text{left}})$ оказывается меньше. Может случиться так, что k-размер достигнет 1, при этом в кластер не будет допущен ни один целый элемент данных, а половину элемента данных в кластер поместить нельзя. Это численный артефакт масштабной функции и числа элементов данных, которые появились к настоящему моменту, поэтому мы не хотим иметь (и физически не можем иметь) менее 1 элемента данных на кластер.

Упражнение 3

Какое максимальное число кластеров можно получить при $\Delta = 10$? Как выглядят k-размеры при наличии максимального числа кластеров (зная, что два соседних кластера сливаются всякий раз, когда они выясняют, что их слияние останется в пределах k-размера)?

8.3.3 Слияние t-дайджестов

Теперь, когда мы разобрались в работе масштабной функции, алгоритм становится невероятно простым. Здесь мы описываем версию алгоритма слияния. Алгоритм одинаков как для обновления одного t-дайджеста вновь прибывшим набором элементов данных, так и для слияния двух t-дайджестов. При этом он состоит из двух поочередно выполняемых фаз: сортировки и слияния.

Мы исходим из того, что, помимо пространства, отведенного для t-дайджеста S_n, у нас есть дополнительный буфер для приема конечного числа l поступающих элементов данных $X_L = [x_1, x_2, x_3, \ldots, x_l]$. По мере их прибытия мы конкатенируем их с S_n (для обозначения числа элементов пока что мы используем незаглавное n). Мы укладываем их рядом. Представьте это как неупорядоченное объединение t-дайджеста и X_L, который представляет l одиночек со средними значениями, идентичными данным x_i с весом 1.

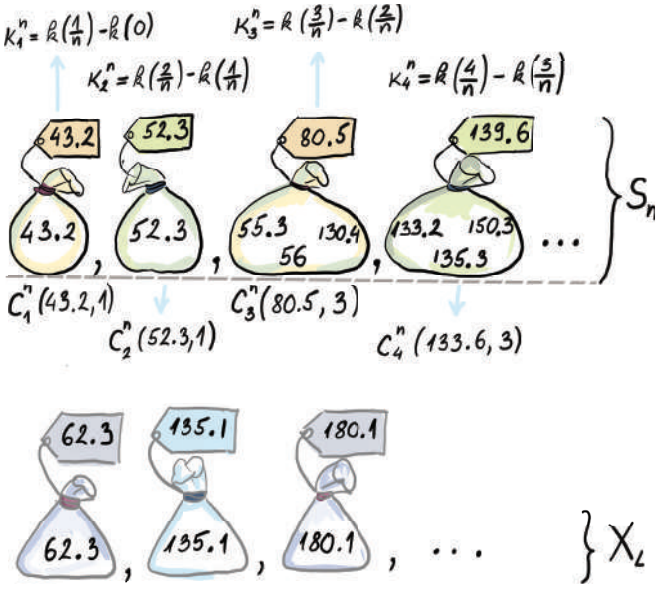


Рисунок 8.4 Строго упорядоченный t-дайджест S_n (верхние индексы у K и C обозначают число элементов из потока, появившихся на данный момент). Обратите внимание, что аргумент k-функции появляется с шагом $1/n$. X_L представляет l новых элементов из потока, которые мы добавляем в t-дайджест (мы показываем только левый хвост набора X_n и S_n)

Затем мы сортируем все $|S_n| + |X_L|$ кластеров по их среднему значению и выполняем следующую проверку слева направо. Есть ли сосед справа от кластера, с которым он мог бы слиться, оставаясь в пределах своего k-размера, равного 1? Если два соседних кластера могут слиться подобным образом, то они это делают слева направо. Если кластер может слиться со своим соседом, то новый результирующий кластер будет иметь среднее значение, равное средневзвешенному значению центроидов двух слившихся кластеров. Вес результирующего кластера будет равен сумме весов отдельных кластеров.

После сортировки в соответствии со значением центроида мы перемещаемся слева направо и проверяем, не приведет ли добавление кластера справа к превышению k-размера границы, равной 1. На этот раз аргумент k-функции увеличивается на $1/(n + l)$. Значения $\mathcal{K}_i^n$ на рис. 8.4 были рассчитаны с шагом на $1/n$ для каждого представленного кластером элемента. Это объясняется тем, что теперь нужно учитывать l новых элементов.

На рис. 8.5 показана начальная точка после фазы сортировки. Она становится входными данными, с которыми начинает работать фаза слияния.

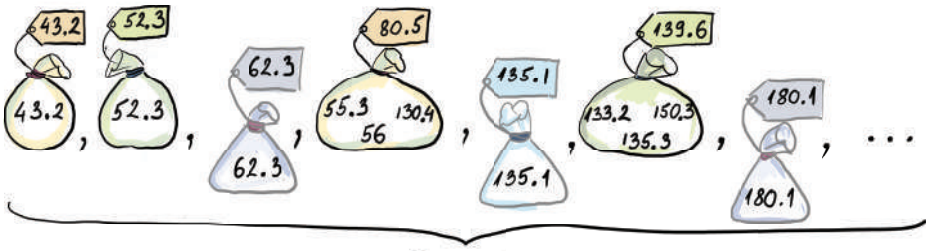


Рисунок 8.5 Отсортированный нижний конец $[S_n, X_n]$ перед началом фазы слияния

На рис. 8.6 показан процесс слияния с пошаговым приращением для левого хвоста S_n и X_L. Другими словами, мы показываем только нижние концы целых S_n и X_L, но поскольку фаза слияния начинается слева и перемещается вправо, этого должно быть достаточно, чтобы понять, как оно происходит. Приведенное выше описание и псевдокод из оригинальной статьи должны дать вам хорошую основу, если вы когда-нибудь захотите реализовать эту концепцию самостоятельно.

На рис. 8.6 показано, что попытка первого кластера ассимилировать соседнего одиночку оказывается безуспешной, поскольку результирующий k-размер слишком велик. Это означает, что мы останавливаемся и формируем первый кластер слева как одиночку (с тем же значением и весом, что и до слияния). Следующая попытка слияния выполняется вторым одиночкой по отношению к своему правому соседу (см. рис. 8.6, временная точка 3). Она проходит хорошо, если судить по k-размеру потенциально нового будуще-

го кластера. Тот же процесс продолжается, и теперь тенденция к слиянию расширяется в сторону следующего кластера справа. Следующий кластер с центром тяжести 80.5 имеет вес 3. При добавлении к весам одиночек 2 и 3 это приведет к тому, что k-размер будет больше 1. Следовательно, мы должны остановиться и объединить двух одиночек, которые остаются внутри легальной границы k-размера (рис. 8.6, момент времени 5). Одновременно с этим кластер с центроидом 80.5 начинает смотреть вправо и выясняет, не приводит ли сумма его веса и веса одиночки справа от него к устойчивому увеличению k-размера. Похоже, что нет, и в момент времени 6 (см. рис. 8.6) мы создаем новый *старый* кластер с центром тяжести 80.5 и весом 3. Затем процесс продолжается аналогичным образом. Если при каждом создании нового кластера он является результатом слияния одного или нескольких кластеров, то его центр тяжести вычисляется заново, и его вес обновляется.

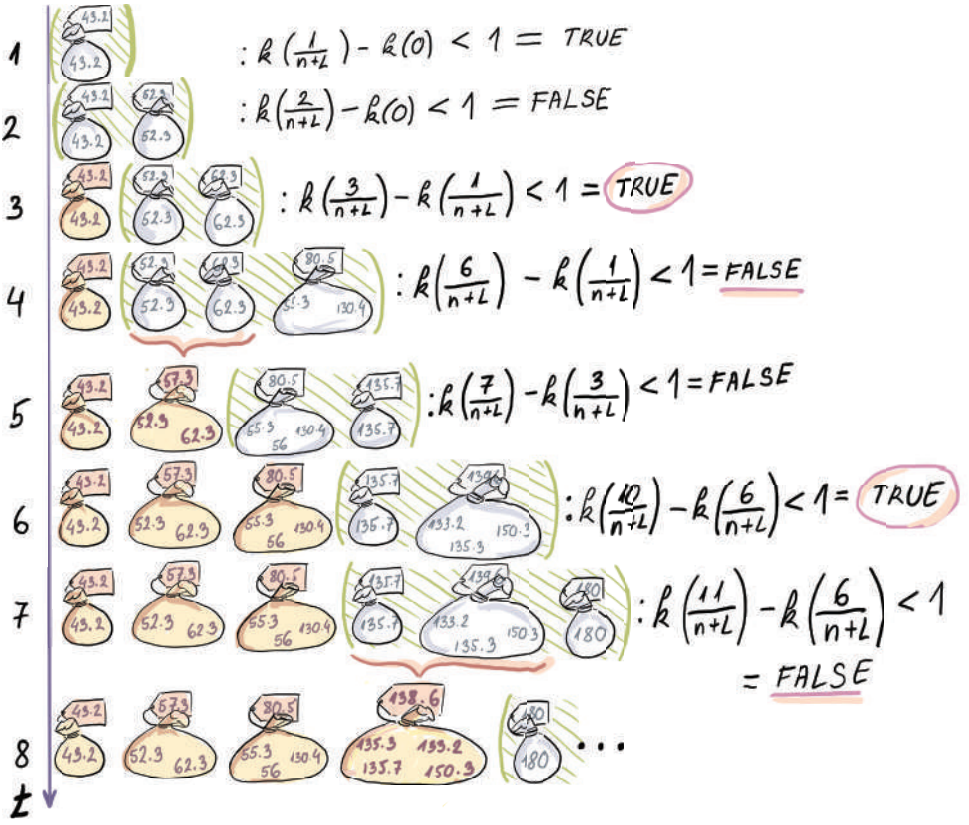


Рисунок 8.6 Фаза слияния в рамках алгоритма t-дайджеста. Всякий раз, когда ассимиляция кластера справа недопустима (обозначается отрицательным ответом на запрос k-размера), мы формируем новый кластер (обозначается закрытым [более темным] мешком). В каждый момент времени текущие попытки слияния с соседом(ями) справа обозначаются зелеными скобками. Кластеры в момент времени 8 являются первыми четырьмя кластерами нового S_{n+1}

Числовые значения на этикетках мешков на рис. 8.6 – это то, что мы фактически сохраняем, тогда как наблюдения из потока (показанные на мешке) отбрасываются. Сырые значения в мешке алгоритмом не сохраняются, но мы показываем их здесь для наглядности. В дополнение к этому мы сохраняем вес каждого нового кластера.

Обратите внимание, что при слиянии строго упорядоченный t-дайджест может стать слабо упорядоченным: 62.3 миллисекунды больше, чем 55.3 миллисекунды; тем не менее 62.3 миллисекунды являются частью C_2 (в S_{n+1}), тогда как 55.3 представлены кластером C_3 (в S_{n+1}) после окончания фазы слияния. Ситуация может стать еще более радикальной, если использовать этот алгоритм для слияния двух t-дайджестов с заменой одиночек на реальные кластеры второго t-дайджеста. Большие значения Δ (вспомните, что Δ определяет вид упорядоченности дайджеста) в этом случае могут увеличивать ошибку, но, похоже, на практике это случается нечасто.

Случай слияния двух t-дайджестов протекает точно так же, как это было показано на рис. 8.6, но теперь мы имеем дело не с l одиночками из потока, а с неким m_2 числом кластеров второго t-дайджеста.

Возможность получать t-дайджест для «двух миров», $D_1 \cup D_2$, путем слияния t-дайджестов, построенных на D_1 и D_2 по отдельности, очень впечатляет, так как позволяет использовать вычислительную архитектуру MapReduce. Допустим, наши данные о продолжительности посещения веб-сайта разделены на подпотоки в зависимости от географического местоположения, откуда происходит посещение. Если объем данных огромен, а это бывает связано с популярными веб-сайтами, такими как социальные сети или поисковые системы, то задание на аппроксимацию хвостовых квантилей, вероятно, придется параллелизовывать. Подпотоки можно отправлять в соответствии с географическим подразделом на разные узлы в потоковом приложении. Они создают свои t-дайджесты, построенные на непересекающихся наборах данных, и отправляют их на свой ведущий узел. Там t-дайджесты будут агрегированы путем их слияния, чтобы аппроксимировать квантили всех данных. Такая характеристика наброска, или резюме, желательна, и хорошие алгоритмы содержат эту функциональность, именуемую *слияемостью*[73]: способностью резюме сливаться с другими резюме, при этом предотвращая рост ошибки результирующего резюме, как формально определено в статье Аггарвала и соавт. (**http://mng.bz/p2NG**). Эта статья довольно техническая, но если вы сможете проследить изложенные в ней идеи, то она станет очень увлекательным чтением и поэтому настоятельно рекомендуется.

8.3.4 Пространственные границы t-дайджеста

Если мы посмотрим на пространственные границы этого алгоритма, то для одного t-дайджеста нам нужно поддерживать только m кластеров, и каждый кластер хранит постоянный объем информации, среднее значе-

[73] Англ. Mergeability. – *Прим. перев.*

ние и вес и, возможно, некие служебные метаданные, если это необходимо. Таким образом, необходимое для t-дайджеста пространство ограничено максимальным числом кластеров, которые мы поддерживаем в любой момент времени. Максимальное число кластеров получается для функции k_1 следующим образом: поскольку всем кластерам разрешается сливаться, они сливаются, при этом максимальное число кластеров возникает, когда кластерам *почти* разрешается слиться, но не совсем.

Это означает, что при слиянии соседних кластеров *k*-размер будет чуть больше 1 (например, 1.01). Если это так, то средний *k*-размер кластера не меньше 0.5; иначе хотя бы одна пара может слиться. Если $n > \delta$, то максимальное число центроидов равно δ, параметру масштабной функции (вспомните, что минимум был $\delta/2$). Более подробный анализ числа кластеров и максимального веса каждого кластера можно найти в короткой статье одного из создателей t-дайджеста, Теда Даннинга (Ted Dunning) (**https://arxiv.org/abs/1903.09921**).

По-видимому, мы имеем постоянную пространственную границу $O(\delta)$ для любого числа *n* прибывших элементов данных. Параметр δ t-дайджеста, по понятным причинам, называется *параметром сжатия*. С нашей точки зрения, для большинства приложений, в которых используются t-дайджесты, это довольно близко к волшебству (оставляя за скобками тот факт, что ошибка должна устанавливаться эмпирически заново для каждого приложения, а универсальной гарантии не существует).

8.4 q-дайджест

В этом разделе мы представим другой квантильный дайджест (или q-дайджест), введенный Шриваставой (Shrivastava) и соавт. [3], являющийся предшественником эвристики t-дайджеста, которая предлагает гарантии сложности наихудшего случая по ошибке и пространству. Помимо успокоения наших алгоритмических душ, q-дайджест служит хорошим примером структуры данных, которая наиболее точно отвечает на запросы об элементах с наибольшей частотой. Наличие этой функциональности желательно при работе с частотными данными, однако она не используется многими другими структурами данных. Вспомните главу 4, в которой говорилось, что набросок count-min давал одинаковый диапазон завышенных оценок частот как бестселлеров, так и книг, которые так и не были проданы.

q-дайджест можно использовать в ситуациях, когда элементы имеют заранее заданный диапазон легальных значений, $U = [1, \sigma]$. По существу, для того чтобы его использовать, нужно знать возможный максимум данных. Это реалистичное допущение для любых данных, генерируемых каким-либо журналированием или умным устройством. Цель q-дайджеста – резюмировать набор данных *S*, который присутствует в форме пар ключ-значение, $S = \{a_1{:}c_1, a_2{:}c_2, \ldots, a\sigma{:}c\sigma\}$, где a_i – это элемент из *U*, а c_i – вес/частота элемента a_i. Кроме того, $\sum_{i=0}^{\sigma} c_i = n$ (общая сумма наблюдений).

Вот примерное представление о том, как q-дайджест размывает точность, чтобы экономить пространство: если число появлений элемента расценивается как слишком малое, чтобы отдельно хранить в виде пары ключ-значение, то информация о его числе появлений сливается с информацией о соседнем элементе с аналогичным низким числом появлений, чтобы сберечь информацию о числе элементов в соответствующем диапазоне, но информация о числе конкретных элементов будет утеряна.

Например, в зависимости от заданных значений параметров q-дайджеста две пары ключ-значение, {3:1, 4:1}, могут быть слиты в одну пару: {[3,4]:2}. Другими словами, мы переходим от знания о существовании одной копии 3 и одной копии 4 к знанию о наличии в интервале [3,4] двух копий. Эта идея далее применяется к интервалам с малым числом появлений. Сами интервалы затем могут быть слиты ради экономии места, если элементы в этих интервалах не имеют достаточно высоких частот. Решение о том, каким является высокое или низкое число появлений, и последующий шаг «размытия» определяются параметром сжатия *k*. Вспомните, что в t-дайджесте были масштабные функции; так вот, это аналогичное понятие, призванное управлять числом интервалов. Далее мы покажем, как конструировать и хранить q-дайджест.

8.4.1 Конструирование q-дайджеста с нуля

В целях понимания принципа работы q-дайджеста мы вообразим неявное дерево *T* (дерево не сохраняется). Дерево представляет собой полное двоичное дерево с σ листьями (по размеру равное универсальному множеству *U*), где *i*-й лист слева обозначает *i*-й элемент универсального множества *U*. В простом примере, показанном на рис. 8.7, универсальное множество равно $U = [1,8]$, а набор данных равен $S = \{1{:}1, 3{:}5, 4{:}9, 5{:}2, 7{:}2, 8{:}1\}$. Эта информация о частоте каждого элемента будет храниться в каждом соответствующем листе *T*. Каждый узел *v* из *T* имеет соответствующий счетчик числа появлений $count(v)$, и вначале $count(v)$ заполнен только у листьев.

При создании q-дайджеста из этого неявного дерева мы следуем двум правилам:

1) для каждого внутреннего узла *v* в *T*, который не является корневым, $count(v) \leqslant n/k$ (это правило разрешено нарушать листовым и корневым узлам);
2) для каждого узла *v* в *T*, который не является корневым, $count(v) + count(v_S) + count(v_P) > n/k$, где v_S обозначает сестринский узел узла *v*, а v_P обозначает родителя узла *v* (это правило разрешено нарушать корневому узлу).

Здесь мы показываем процесс преобразования неявного дерева из рис. 8.7 в q-дайджест в соответствии с правилами 1 и 2 за пару шагов. В этом примере мы имеем $n = 20$ и $k = 5$, поэтому максимальное допустимое значение в узле равно 4, и каждая «треугольная сумма» из правила 2 должна

быть не менее 5. Обратите внимание, что в начале процесса правило 1 не нарушается, так как оно не относится к листовым узлам, а они единственные содержат значения в самом начале.

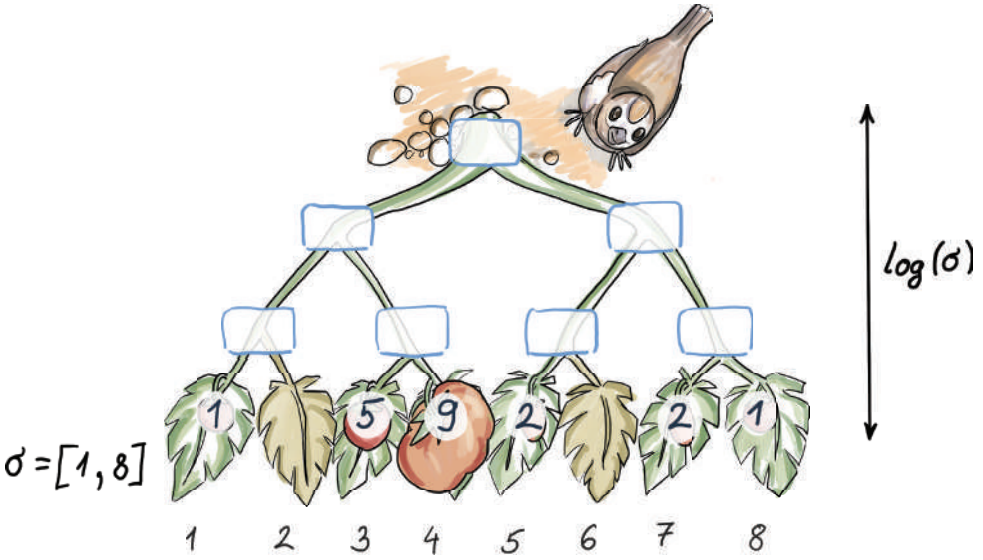


Рисунок 8.7 Изначальные данные и их частоты, представленные в листьях неявного дерева

Процесс начинается на листовом уровне, где, двигаясь слева направо (как в t-дайджесте), мы идентифицируем все «треугольники» на нижнем уровне, которые нарушают правило 2. Всякий раз, когда это происходит, значения узлов v и v_S суммируются, добавляются к значению в v_P и затем удаляются из v и v_S. Например, в самом правом треугольнике внизу первого дерева на рис. 8.8 мы суммируем 2 и 1, помещаем 3 в их родителя, а затем удаляем 2 и 1. Мы продолжаем в таком же ключе на следующем уровне, снова двигаясь слева направо и находя проблемные треугольники. Процесс заканчивается в корне, и корню разрешается иметь бесконечно низкие или высокие значения.

Используя этот процесс, мы приходим к окончательному q-дайджесту, представленному последним неявным деревом на рис. 8.8. Дерево никогда не сохраняется в явной форме; сохраняются только те узлы, в которых есть значения. Если задать узлам дерева поэтапное перечисление слева направо, то результирующий q-дайджест на рис. 8.8 сохранит следующую информацию: Q = {[1,8]:1, [5,6]:2, [7,8]:3, [3]:5, [4]:9}. Если перечислить каждый узел и соответствующий интервал по уровням слева направо, то будет получено следующее описание: Q = {1:1, 6:2, 7:3, 10:5, 11:9}. Мы перешли от хранения шести значений ключей, когда у нас были изначальные данные, к хранению пяти из них – не так уж и много. Но при большем объеме уни-

версального множества и большом числе низких значений, начинающихся с листовых узлов (типичное распределение Ципфа, демонстрируемое данными о проведенном на веб-сайте времени), экономия пространства становится весьма существенной.

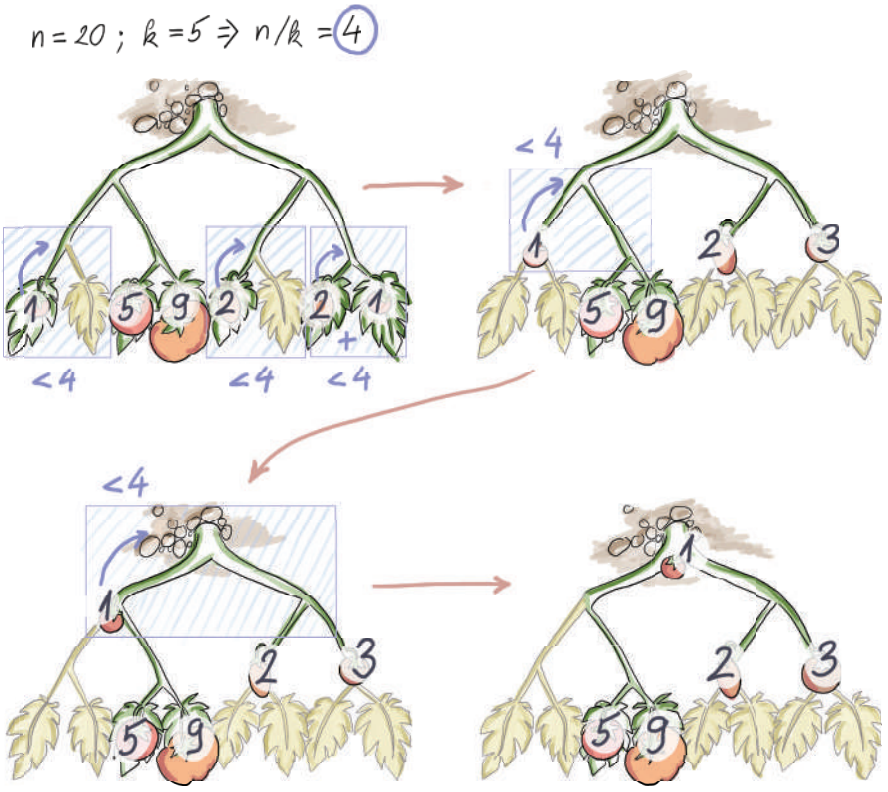


Рисунок 8.8 Строительство q-дайджеста с нуля

8.4.2 Слияние q-дайджестов

q-дайджесты изначально были разработаны для сенсорной сети и условий работы в распределенной среде, где q-дайджесты могут вычисляться локально, а затем сливаться с q-дайджестами в других узлах. Если оба q-дайджеста относятся к одному и тому же универсальному множеству, то процесс слияния q-дайджестов довольно прост. Имея два дерева T_1 и T_2, q-дайджесты могут сливаться путем создания дерева T над идентичным универсальным множеством и суммирования соответствующих узлов из T_1 и T_2 в T. Взгляните на процесс, изображенный на рис. 8.9, где $n_1 + n_2 + 30$, $k_1 = k_2 = 6$. Максимальное значение в каждом узле изначальных T_1 и T_2 составляло 5.

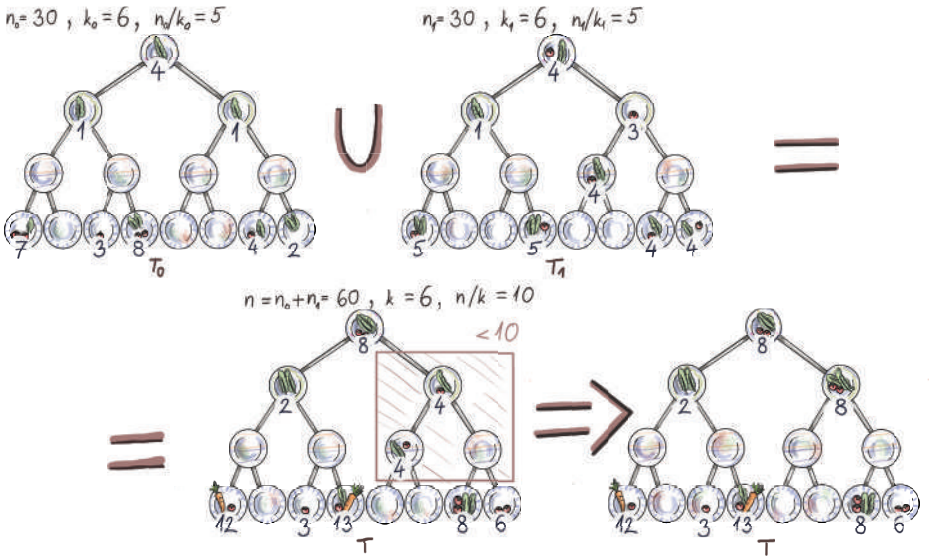


Рисунок 8.9 Слияние двух q-дайджестов, где $n_1 = n_2 = 30$, $k_1 + k_2 = 6$. Поскольку дайджесты имеют одинаковые размеры, результирующий q-дайджест имеет одинаковый размер, где значения в узлах являются суммами в соответствующих узлах. Например, результирующий q-дайджест имеет значение 8 в корне, поскольку участвующие в операции слияния оба q-дайджеста имеют значение 4. Однако это может привести к тому, что q-дайджест будет слит не полностью. В данном примере результирующий q-дайджест имеет $n = n_1 + n_2 = 60$ и $k = k_1 + k_2 = 6$, поэтому мы ищем треугольники, сумма которых равна 10 или меньше, и мы распространяем эти значения вверх вплоть до родителя, как и раньше

Сразу после создания дерева T из T_1 и T_2 оно также должно пройти процесс конструирования легального q-дайджеста, соблюдающего оба ранее упомянутых правила, используя следующие параметры: $n = n_1 + n_2, k = k_1 = k_2$. Если два сливаемых q-дайджеста содержат примерно сопоставимые или равные суммы наблюдений, то порог результирующего q-дайджеста (n/k) будет в два раза больше порога предыдущих q-дайджестов. В нашем примере максимальное значение узла в результирующем q-дайджесте равно 10. Два изначальных q-дайджеста, подлежащих слиянию, хранили в общей сложности 16 пар ключ-значение, но результирующий q-дайджест использует только 8 пар ключ-значение, вдвое меньше.

8.4.3 Соображения по поводу ошибки и пространства в q-дайджестах

Максимальное используемое q-дайджестом пространство зависит от параметра сжатия k и равно $3k$. Обозначим размер Q через $|Q|$ (измеряемый в числе пар ключ-значение). Тогда, исходя из правила 2, мы имеем:

$$\sum_{v \in T} count(v) + count(v_s) + count(v_p) > |Q| \times \frac{n}{k}.$$

Кроме того, справедливо, что

$$3 \times \sum_{v \in T} count(v) \geq \sum_{v \in T} count(v) + count(v_s) + count(v_p),$$

так как в правом выражении число появлений каждого узла подсчитывается не более чем три раза (каждый узел появляется один раз как родительский, один раз как сестринский и один раз как сам по себе; подумайте о том, что здесь подсчитывается для листьев и корня). Левое выражение равно $3n$, следовательно, $|Q| \times n/k$, в результате давая $|Q| < 3k$.

Что касается частоты ошибки, то прежде чем вычислять ошибку при отправке квантильных запросов, нужно понаблюдать за дальностью отклонения значения в пределах одного узла неявного дерева *T*. Все узлы на пути между корнем и узлом *v* могут содержать значения, потенциально относящиеся к интервалу, указанному *v*. Учитывая этот факт и глубину дерева, равную log σ, общая ошибка в пределах одного узла равна $\log \sigma \times n/k$. Если мы наблюдаем эту ошибку в относительном смысле (в процентах от *n*), то получаем, что ошибка в пределах одного узла составляет не более $\log \sigma/n$.

8.4.4 Квантильные запросы с использованием q-дайджестов

Для того чтобы делать квантильные запросы с помощью q-дайджестов, полезно сортировать узлы неявного дерева в обратном порядке его обхода[74]. Другими словами, мы помещаем интервал $i = [x, y]$ в отсортированной последовательности только после того, как все его потомственные подынтервалы уже были размещены. Сразу после того, как это будет сделано и будет предоставлен квантильный запрос *x*, мы прокручиваем массив пар ключ-значение, накапливая значения до тех пор, пока не будет достигнуто или превышено *x*. Сообщенный квантиль представляет собой правый конец интервала, в котором *x* был превышен. На рис. 8.9 приведен пример квантильного запроса к результирующему q-дайджесту.

Последовательность, полученная в результате обратного порядка обхода узлов этого дерева, узлы которого перечисляются по уровням, выглядит следующим образом: {8:12, 10:3, 11:13, 2:2, 14:8, 15:6, 3:8, 1:8}. Это соответствует правосортированным диапазонам {[1]:12, [3]:3, [4]:13, [1,4]:2, [7]:8, [8]:6, [5,8]:8, [1,8]:8}. Допустим, мы получили запрос, $\varphi = 0.5$; следовательно, $x = n/2 = 30$ (то есть мы ищем медиану). Прокручивая слева направо и накапливая значения в списке, мы получим сумму ровно 30, то есть 12 + 3 + 13 + 2 = 30, и сообщим о правом конце последнего узла как о

[74] Англ. post-order traversal. – *Прим. перев.*

медиане. Для узла, содержащего 2, правый конец его диапазона равен 4. Сообщенная медиана равна 4. По аналогии предположим, что мы ищем $3n / 4 = 45$. Мы снова начинаем слева, накапливая сумму 12 + 3 + 13 + 2 + 8 + 6 + 8 = 52. С последним значением, 8, мы проскакиваем мимо искомого значения, но наша ошибка будет соизмерима с максимальными значениями узлов, поэтому мы сообщаем о правом конце интервала, соответствующего паре ключ-значение [5,8]:8, то есть 8. Возвращенный результат равен 8.

q-дайджест может также использоваться для ответов на многие другие типы запросов, в первую очередь на диапазонные запросы, обратные квантили и консенсусные запросы. При наличии m ячеек памяти для построения q-дайджеста ошибка в квантильном запросе составляет не более $\varepsilon \leq (3 \log \sigma)/m$. Мы получаем это, задавая коэффициент сжатия k равным $m/3$.

8.5 Исходный код симуляции и ее результаты

Для того чтобы увидеть работу t-дайджестов и q-дайджестов в действии, мы разработали симуляционный сценарий, в котором продемонстрировали эмпирическое поведение их ошибок и сравнили их оценки перцентилей, находящихся далеко в правом хвосте.

Мы взяли 10 выборок без возврата, каждая из которых содержала 10^5 элементов из 2 Гб данных веб-сайта, показанных на рис. 8.1. Поскольку q-дайджест работает только с целыми числами, мы округлили данные веб-сайта до ближайшей миллисекунды. Благодаря этому мы можем использовать оба алгоритма на одних и тех же выборках. Используемые в исходном коде 10 выборок по 100 К наблюдений в каждой, а также общие данные веб-сайта доступны в репозитории исходного кода книги.

Результаты вычисляются с помощью библиотеки Python `tdigest` и версии q-дайджеста, реализованной на Python, из блог-поста **http://mng.bz/pOVw**, после валидации исходного кода на нескольких малых потоковых примерах. Ниже показан исходный код вычисления и сохранения результатов соответственно из t-дайджеста и q-дайджеста. Он читает 10 выборок, и после того, как все 10 выборок были потреблены их собственным объектом t-дайджеста или q-дайджеста, исполняется запрос на получение 95-го и 99-го процентилей данных. В итоге мы получаем 10 оценок 95-го и 99-го процентилей:

```
import pandas
from pandas import DataFrame
from tdigest import TDigest
import numpy as np
import os
```

```
df = pandas.read_csv('./test.csv')                                    ❶

resNinetyFive = np.array([])                                          ❷
resNinetyNine = np.array([])                                          ❷

columns = list(df)
for j in columns:
    tDigest = Tdigest(delta=1 / 200)                                  ❸

    tDigest.batch_update(df[j], w=1)                                  ❹

    resNinetyFive = np.append(resNinetyFive, tDigest.percentile(95))  ❺
    resNinetyNine = np.append(resNinetyNine, tDigest.percentile(99))  ❺

res = DataFrame({'NinetyFive': resNinetyFive,
                 'NinetyNine': resNinetyNine})                        ❻

os.chdir("./")                                                        ❼
res.to_csv("Results_TD_WebsiteSample.csv", index=False)               ❼
```

❶ Прочитать 10 выборок длиной 100 K, каждый в виде кадра данных. Файл .csv можно найти в хранилище исходного кода книги
❷ Создать пустые массивы, в которых будут храниться результаты 95-го и 99-го процентилей
❸ Для каждой выборки составить t-дайджест с величиной, обратной $\delta = 200$ (в реализации и статье дельта параметризуется по-разному)
❹ Позволить дайджесту потребить j-ю выборку. w означает, что мы добавляем одиночки с весом 1. Для двух разных дайджестов это были бы фактические веса кластеров
❺ Добавить оценку 95-го и 99-го процентилей в массив результатов
❻ После того как все 10 выборок потреблены, создать кадр данных для результатов
❼ Указать каталог и сохранить результаты

Эффективность реализации оказывается намного выше для решения с t-дайджестом, настолько, что делает сравнение эффективности тривиальным:

```
import numpy as np

df = pandas.read_csv('/path/to/test/data')                            ❶

resNinetyFive = np.array([])                                          ❷
resNinetyNine = np.array([])                                          ❷

columns = list(df)

for j in columns:
    universeSize = max(df[j])+1                                       ❸
    qDigest = QDigest(universeSize, 20)                               ❸
```

```
    length = len(df['sample1'])                                    ❹
    for i in range(length):
        qDigest.insert                                             ❹

    qDigest.compress()                                             ❺

    resNinetyFive = np.append(resNinetyFive,
                              qDigest.quantile_query(([0.95])))    ❻
    resNinetyNine = np.append(resNinetyNine,
                              qDigest.quantile_query(([0.99])))    ❻

res = DataFrame({'NinetyFive': resNinetyFive,
                 'NinetyNine': resNinetyNine})                     ❼
res.to_csv("/path/to/test/output", index=False)                    ❽
```

❶ Прочитать 10 выборок длиной 100 K, каждая в виде кадра данных. Файл .csv можно найти в хранилище исходного кода книги
❷ Создать пустые массивы, в которых будут храниться результаты 95-го и 99-го процентилей
❸ Для каждой выборки составить q-дайджест с размером универсального множества в качестве параметра. +1 – это для нулей
❹ Класс q-дайджест принимает по одному элементу за раз и потребляет j-ю выборку поочередно
❺ q-дайджест реорганизовывается после того, как будет потреблена вся выборка, в соответствии с правилами двух треугольников
❻ Добавить оценки 95-го и 99-го процентилей в массив результатов
❼ После того как потреблены все 10 выборок, создать кадр данных для результатов
❽ Указать каталог и сохранить результаты

Что касается параметров, выбранных для создания двух дайджестов, то мы решили сделать их примерно одинаковыми по размеру. Для q-дайджеста мы выбираем параметр сжатия = 20, а для t-дайджеста – $\delta = 200$, что позволяет получить около 1 Кб каждого дайджеста. Для каждой выборки мы нашли максимальную продолжительность посещения, так как она необходима для создания q-дайджеста. Для всех 10 выборок максимумы оказались в пределах от 2 742 437 до 2 763 605 миллисекунд; следовательно, размеры универсальных множеств не настолько варьируются, чтобы помешать значимому перекрестному оцениванию результатов.

Ошибка, которую мы показываем, вычисляется следующим образом: для каждой из 10 выборок можно получить точные 95-й и 99-й процентили, поскольку можно отсортировать данные и найти фактические значения. Это те значения x, которые мы надеемся получить. Из дайджестов мы получаем значения z и можем проверить разницу между $R(x)$ и $R(z)$. В случае 95-го процентиля $R(x)$ должно составлять 95 000. Если дайджест вернул 94 990-й элемент, то показываемая абсолютная ошибка равна $|0.94990–0.95000| = 0.00010$. В этом можно убедиться на правом графике; именно настолько

t-дайджест приближается к истине. На рис. 8.10 показаны результирующие абсолютные эмпирические ошибки для оценок 95-го процентиля по каждой из 10 выборок. Первым делом стоит обратить внимание на то, что мы не показываем абсолютные эмпирические ошибки на той же оси *y*. Средние абсолютные эмпирические ошибки q-дайджеста примерно в 5000 раз превышают средние абсолютные эмпирические ошибки t-дайджеста. Если бы мы показали их обе на одной оси, то не смогли бы визуально оценить их внутригрупповую изменчивость. Средняя абсолютная ошибка t-дайджеста (треугольников) составляет 1.2×10^{-5}, а для q-дайджеста она в среднем 4965.4×10^{-5}. Похоже, что t-дайджест превосходит q-дайджест, если судить по абсолютной эмпирической ошибке на этих данных, на порядок величины 10^3.

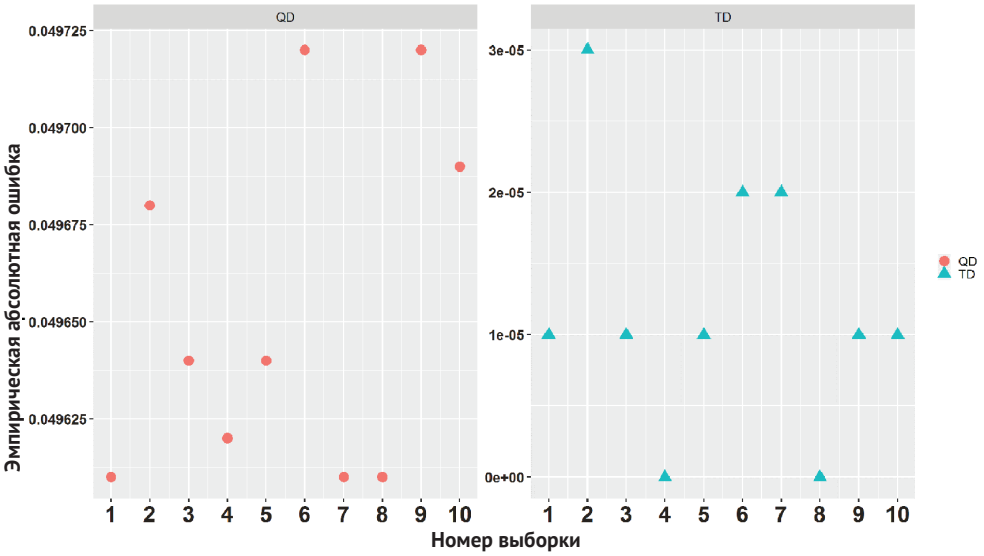


Рисунок 8.10 На графиках показана эмпирическая абсолютная ошибка, которую демонстрируют q-дайджест (слева) и t-дайджест (справа) при оценивании 95-го процентиля. Ошибка при оценивании $q \in [0,1]$ вычисляется следующим образом: если предоставляемый дайджестом истинный ранг ответа (значения) равен r, то абсолютная ошибка равна $|r/n - q|$, где n – это число элементов, обработанных на данный момент

Для того чтобы в полной мере оценить эту разницу, сначала необходимо понять, что значит отклониться на 1.2×10^{-5} при оценивании 95-го процентиля. Поскольку данные были созданы, мы знаем точный квантиль любого $q \in [0,1]$ (при условии что выборка обеспечивает такую достоверность; то есть трудно получить точный 99-й процентиль выборки из 10 элементов). Следовательно, если получаемый из дайджеста истинный ранг z при запросе 95-го процентиля равен r, тогда абсолютная ошибка становится $|r/n - 0.95|$, где n – это число элементов в выборке (или появившихся в потоке на данный момент). Таким образом, если значение

отклоняется на 1.2×10^{-5} от 0.95, то это означает, что t-дайджест показывает 95 001-е или 95 002-е проведенное на веб-сайте время вместо 95 000-го. Согласно тем же соображениям, q-дайджест возвращает 90 035-е или 99 966-е проведенное на веб-сайте время в их упорядоченной последовательности вместо 95 000-го. Кроме того, еще следует обратить внимание вот на какую вещь: здесь мы говорим не о миллисекундах, а о рангах, занимаемых длительностями посещений.

Ту же картину можно увидеть на рис. 8.11, где мы показываем аналогичные результаты для 99-го процентиля. Хотя разница в средней абсолютной эмпирической ошибке в семь раз меньше, чем для 95-го процентиля, t-дайджест по-прежнему ошибается на один элемент, тогда как q-дайджест ошибается почти на тысячу. Тем не менее заявленная в оригинальной статье граница ошибки q-дайджеста остается в силе. Учитывая наш максимальный размер универсального множества 2 763 605 и $k = 20$, можно вычислить, что ошибка должна быть ниже 0.74. Это верно и для нашего случая. Граница аддитивной ошибки для оценивания 95-го процентиля в размере 0.74 означает, что даже при указании 22-го процентиля в качестве ответа удастся сохранить теоретическую верхнюю границу ошибки.

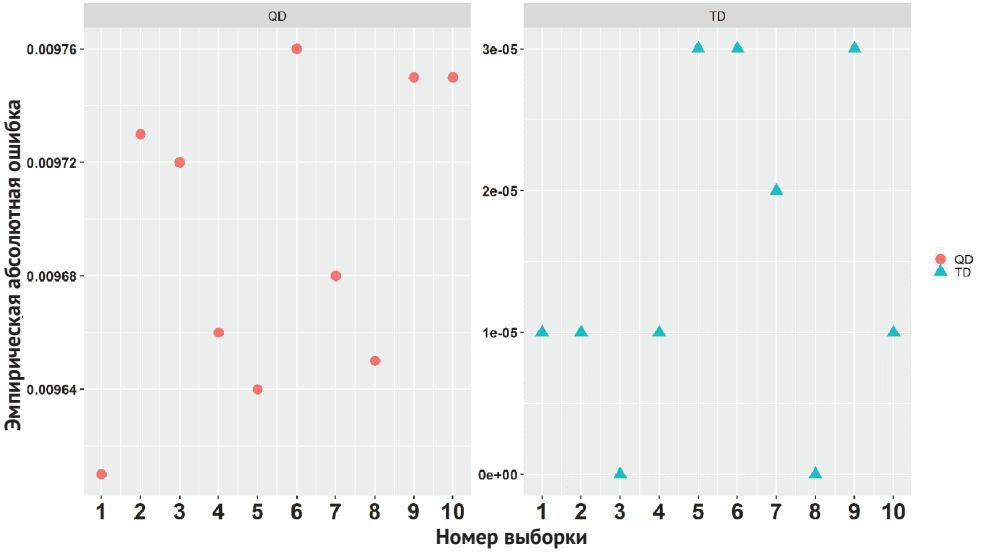


Рисунок 8.11 Эмпирическая абсолютная ошибка, которую демонстрируют q-дайджест (слева) и t-дайджест (справа) при оценивании 99-го процентиля. Ошибка при оценивании $q \in [0,1]$ вычисляется следующим образом: если истинный ранг предоставляемого дайджестом ответа (значения) равен r, то абсолютная ошибка равна $|r/n - q|$, где n – это число элементов, принятых на данный момент

Согласно нашим результатам, t-дайджест явно превзошел q-дайджест при оценивании верхних квантилей, таких как 95-й и 99-й процентили на этих данных. Поскольку t-дайджест не содержит какой-либо ошибки верх-

ней границы, нельзя утверждать, что она остается ниже нее, но, судя по эмпирическим результатам на этих данных, она определенно остается ниже q-дайджеста.

Резюме

- Наличие функциональности получения импровизированных приближенных квантилей в любое время в приложении по обработке потоковых данных, в особенности в том, которое используется для обнаружения аномалий, имеет очень важное значение. Эффективные онлайновые алгоритмы, поддерживающие небольшие резюме о всех данных и выдающие ответы о квантилях с некой гарантией ошибки, предоставляют возможность создавать небольшой набросок распределения данных в форме гистограммы. Благодаря этому можно наблюдать за несколькими квантилями одновременно и устанавливать чувствительные пороговые значения.
- Ошибка, с которой работают алгоритмы аппроксимации квантилей, является либо аддитивной, либо относительной (мультипликативной). Аддитивная ошибка, εn, одинакова независимо от того, какой квантиль оценивается. Относительная ошибка, $\varepsilon R(x)$, является относительной к конкретному интересующему квантилю, поэтому она меньше для малых квантилей и наибольшая, εn, для максимального значения. Ошибка в пространстве данных, но не в пространстве рангов, иногда также называется относительной ошибкой, но она не имеет ничего общего с рассмотренной нами относительной (мультипликативной) ошибкой.
- t-дайджест – это очень популярный эвристический алгоритм вычисления приближенных квантилей. Мы увидели механизм поддержания малых выборок на краях квантильных интервалов и больших выборок в середине, а также то, как это обеспечивается за счет регулирования k-размера посредством масштабных функций. Это приводит к более высокой точности при оценивании хвостовых квантилей, таких как 95-й или 99-й процентиль, или даже более точных значений, близких к 1. Мы увидели, как t-дайджест дает очень точные оценки в этих диапазонах на 10 выборках из реалистичных данных о проведенном на веб-сайте времени.
- Мы узнали, что такое алгоритм q-дайджеста и как создавать его с нуля, а также как объединять два или более алгоритмов. q-дайджест работает только для целых чисел, и для того чтобы его использовать, нужно быть знакомым лишь с универсальным множеством, из которого данные поступают. Для потоковых данных это может быть не так. Если мы не знаем, какой диапазон данных нам предстоит увидеть, то q-дайджест имеет ограниченное применение. Это ограниче-

ние не распространяется на t-дайджест, и, по-видимому, он работает лучше, чем q-дайджест, показывая абсолютные эмпирические ошибки меньше на два-три порядка при оценивании одних и тех же верхних квантилей.

Часть III

Структуры данных для баз данных и алгоритмы внешней памяти

Если в первой и второй частях мы занимались сжатием и отбором данных, чтобы они умещались в оперативной памяти, то теперь мы можем вздохнуть с облегчением – наши данные, все до единого элемента, удобно расположились на диске. В трех главах третьей части мы научимся эффективно конструировать алгоритмы и структуры данных для крупных наборов данных, размещенных на диске. В частности, мы узнаем, как работают поиск, вставка и удаление данных в разных видах баз данных, как эффективно сортировать большие файлы на диске. Мы также рассмотрим различия в структуре индексов в базах данных, оптимизированных под чтение и под запись. Первым шагом на этом пути будет понимание того, как стоимость операций ввода-вывода (то есть стоимость передачи одного блока данных с диска в основную память) доминирует над стоимостью работы центрального процессора в три или более раз. Таким образом, объектив, через который мы будем наблюдать за эффективностью алгоритма, будет размывать все, что происходит в оперативной памяти, и сужать поле зрения до передачи данных между диском и оперативной памятью. Освоение способов проведения анализа «О» большое с точки зрения переноса данных с диска будет одним из главнейших выводов части III.

Глава 9

Введение в модель внешней памяти

Эта глава охватывает следующие ниже темы:

- введение компьютерных ограничений, влияющих на разработку приложений с интенсивным использованием данных;
- введение и описание модели внешней памяти;
- разработка простых алгоритмов сканирования, поиска и слияния данных во внешней памяти;
- обзор вариантов использования, в которых исследователи данных и программисты работают с огромными файлами;
- применение обозначения «О» большое для измерения эффективности алгоритмов с точки зрения операций ввода-вывода.

В этой главе представлены основополагающие идеи, которые составляют третью часть книги. Вначале мы познакомимся с алгоритмами работы с внешней памятью и моделью внешней памяти [1]. Эта модель научит рассматривать эффективность алгоритмов и структур данных в контексте работы с крупными наборами данных, хранящимися на диске.

Большинство приложений содержат данные в том или ином типе локального или дистанционного хранилища, яркими примерами которого являются файлы и базы данных. Хранилища обеспечивают гибкость, позволяющую долговременно и очень дешево фиксировать крупные объемы данных. Даже когда система извлекает выгоду из сводок данных, которые быстро удовлетворяют запросы из оперативной памяти, мы все равно хотим хранить изначальные данные в каком-нибудь более медленном и вместительном хранилище. Как мы видели на примере фильтров Блума и системы Google WebTable, когда запрос возвращает `Присутствует`, мы обращаемся к диску, чтобы получить пару (ключ, значение) и метаданные либо выяснить, что мы имеем ложноположительный результат.

Структуры данных в основе реляционных (и других типов) баз данных учитывают устройство хранилищ и памяти, чтобы предлагать оптималь-

ную производительность операций с диском, и они отличаются от структур данных, которые оптимально выполняют те же задания в оперативной памяти. Как мы увидим, миры структур данных в оперативной памяти и на диске существенно различаются. Проводя аналогии между двумя типами структур данных и алгоритмическими конструкциями, мы стремимся продемонстрировать общие темы и алгоритмические инструменты, которые могут оказаться полезными при решении любых задач на диске.

Прежде чем двигаться дальше, давайте уточним, что мы имеем в виду, когда говорим «диск». В зависимости от контекста данные могут храниться в разных типах локальных или дистанционных хранилищ, таких как локальный твердотельный накопитель (SSD), локальный магнитный диск, облако или некоторая комбинация этих трех. В этой главе мы будем называть все эти устройства хранения просто «дисками», поскольку центральные проблемы, которые мы планируем решать, являются общими для всех типов хранилищ, главной из которых является низкая скорость доступа. Между разными технологиями хранения данных есть много различий во временах доступа; например, твердотельные накопители обладают характеристиками быстродействия, которые во многих отношениях превосходят магнитные диски, но их производительность недостаточно высока, чтобы полностью устранять проблемы, которые мы рассмотрим в этой главе.

Особенности конструкции компьютерного оборудования, такие как разрыв в производительности процессора, иерархия памяти, задержка и пропускная способность, формируют важную основу для понимания того, как конструировать эффективные структуры данных для внешней памяти. Рекомендуем перечитать раздел 1.4 первой главы, чтобы освежить в памяти эти темы. Ниже приведены ключевые особенности, которые делают проблемы, связанные с разработкой алгоритмов внешней памяти, новыми и уникальными:

- объем основной памяти (ОЗУ) значительно меньше, чем объем крупного набора данных, хранящегося на диске, и может одновременно содержать лишь малую порцию всего набора данных. Следовательно, для того чтобы решить задачу (скажем, отсортировать файл), данные должны вводиться в основную память и извлекаться из нее по частям;
- дисковые данные доставляются поочередными порциями (то есть блоками/страницами). Внесение блока данных в основную память – дорогостоящая операция, которая компенсируется тем, что блок содержит много элементов. Независимо от числа элементов в блоке – одного элемента или тысячи – мы платим одинаковую стоимость в размере одной передачи данных;
- выполнение однократного блочного ввода/вывода, то есть передачи с диска в основную память в рамках операции ввода-вывода, выпол-

няется очень медленно и может быть в 100–1000 раз медленнее, чем обычная вычислительная операция в оперативной памяти;

- последовательный порядок доступа к данным на диске быстрее, чем случайный порядок. Последовательный доступ характеризуется высокой пропускной способностью, а произвольный доступ сопряжен с высокой задержкой;
- чтение с диска, как правило, происходит быстрее, чем запись на диск.

В сухом остатке в этой главе вы должны научиться расширять поле зрения, абстрагируясь от оперативной памяти, чтобы видеть общую картину перемещения данных туда-сюда между диском и основной памятью. Понимание того, что аспект передачи данных является узким местом для многих существующих приложений, а все, что происходит в оперативной памяти (например, используемые алгоритм внутренней памяти или структура данных), зачастую является проблемой второго порядка, – один из ключевых уроков этой главы. Переход к такому образу мышления не всегда прост, учитывая, что мы привыкли считать сравнения, арифметические операции и прочее, что происходит в оперативной памяти, а также иметь в распоряжении весь набор данных. Мы продолжим использовать обозначение «О» большое, чтобы давать характеристику времени выполнения алгоритмов, но отныне выражения внутри «О» большого будут отражать число передач данных, а не циклов центрального процессора. Для того чтобы закрепиться в этой новой картине мира, далее мы представим модель внешней памяти и покажем пару базовых алгоритмических примеров, в которых она симулируется.

9.1 Модель внешней памяти: предварительные сведения

Модель внешней памяти, или модель доступа к диску[75], была впервые предложена в 1988 году, когда многие крупные организации начали сталкиваться со своими первыми проблемами при работе с массивными данными. С тех пор она зарекомендовала себя как невероятно полезный инструмент анализа алгоритмов для приложений с интенсивным использованием данных. Указанная модель изображена на рис. 9.1.

В модели внешней памяти компьютер состоит из внешнего хранилища (диска) бесконечного размера и основной памяти ограниченного размера M. Данные первоначально хранятся на диске и передаются между диском и основной памятью блоками размера B. Сразу после того как данные прибывают в основную память, все вычисления в ней и все остальное, что с ними делается, осуществляются бесплатно. Например, внеся блок данных в память, вы можете считать его отсортированным, если вам нужно, чтобы он был отсортирован. Стоимость сортировки взиматься не будет.

[75] Англ. disk-access model (DAM). – *Прим. перев.*

Любое значимое вычисление (которое не требует превышения размера оставшейся памяти!) можно считать уже выполненным. Взимается только стоимость одной передачи данных в оперативную память в рамках операции ввода-вывода.

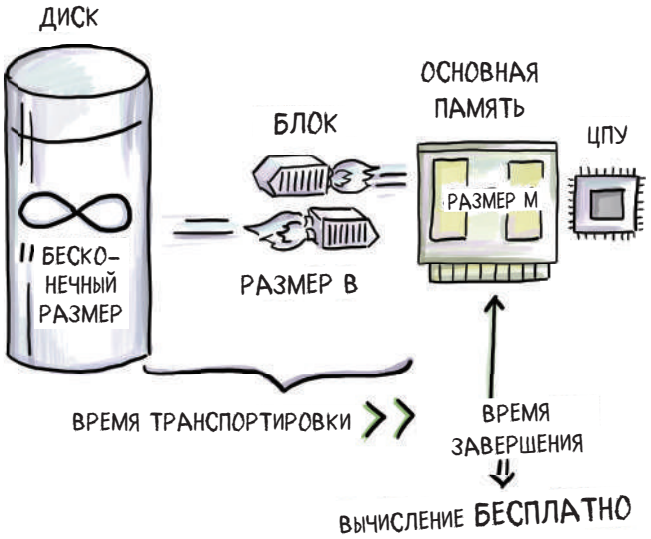


Рисунок 9.1 Модель внешней памяти. Эта модель подходит для анализа приложений по обработке массивных данных, в которых затраты на вычисления в оперативной памяти поглощаются гораздо бóльшими затратами на передачу данных с диска в основную память и обратно. Вычисления не совсем бесплатны, но они настолько дешевле, чем стоимость передачи данных, что во многих приложениях они фактически бесплатны

Параметры M и B можно трактовать как значения, подаваемые в алгоритм. В зависимости от конфигурации оборудования разные компьютеры имеют разные значения параметров M и B. Эти параметры также будут появляться в анализе «О» большое алгоритмов внешней памяти, поскольку они будут использоваться для анализа эффективности. Место появления каждого параметра в границе поможет нам лучше понять роль, играемую каждым из них. Размер наших входных данных на протяжении всей части 3 будет равен N, и он будет обозначать общий объем данных (то есть N элементов размера некой единицы).

Следует учитывать, что даже несмотря на то, что в анализе «О» большое константы не важны, а значения B и M для конкретного компьютера постоянны, мы будем смотреть на них как на параметры, которые могут увеличиваться и изменяться, и время выполнения алгоритмов будет параметризовываться ими. Например, мы не будем упрощать $O(N/B)$ до $O(N)$, даже если B растет не так, как, по нашим ожиданиям, будут расти входные данные, то есть N.

Что означает *B*? *B* бит, *B* целочисленных переменных, *B* 64-битовых слов или что-то совершенно другое? Если мы неукоснительно выбираем одну и ту же единицу измерения для *N*, *M* и *B*, то выбор не так важен, поскольку соответствующие соотношения остаются неизменными; например, *N/B* (число блоков, занимаемых набором данных на диске) или *M/B* (число блоков, умещающихся в основной памяти).

Здесь мы будем считать единицей измерения объем памяти, занимаемый одним конкретным элементом данных в наборе данных, с которым мы работаем, будь то строковое значение, целочисленное значение, значение с плавающей точкой или какой-либо более крупный и сложный объект. Ради простоты мы также допустим, что все элементы в одном наборе данных имеют одинаковый тип и занимают одинаковый объем пространства, даже если реальность в этом случае будет нам противоречить. Например, опыт и исследования показывают, что записи в базах данных могут иметь очень разные размеры, что может существенно влиять на время выполнения алгоритмов, которые слепо предполагают, что все записи имеют одинаковый размер. Если вы хотите узнать подробности, то рекомендуем ознакомиться с исследованиями B-деревьев в части ключей разного размера или сортировки с информацией, касающейся цены каждого размера.

Но для того чтобы чего-то добиться, нужно принять несколько упрощающих допущений. В любом случае, мы исходим из следующих ниже параметров:

- *N* = число записей в наборе данных, хранящемся на диске;
- *M* = число записей, которые могут уместиться в основной памяти;
- *B* = число записей, которые могут уместиться в одном блоке.

Размер *B* обычно равен размеру блока, передаваемого между диском и памятью, и чаще всего составляет от 4 до 64 Кб. На некоторых твердотельных накопителях размер блока составляет порядка пары мегабайт. В любом случае, важно понимать, что один блок содержит тысячи элементов, поочередно размещенных на диске. Размеры памяти также варьируются, и в настоящее время средний компьютер имеет где-то от 8 до 32 Гб. Однако не все это пространство используется для вычислений, поскольку в оперативной памяти хранятся все работающие программы, операционная система и т. д. Размер *N* тоже варьируется, но если мы говорим о данных, хранящихся на диске, то мы должны быть готовы к очень большим наборам данных. Например, создатели системы управления реляционными базами данных Teradata утверждают, что они могут размещать базы данных размером до 50 петабайт (Пб). Подводя итог, будет правильно предположить, что в большинстве ситуаций *N* намного больше, чем *M*, и *M* все еще значительно больше, чем *B*, даже несмотря на то, что разрыв между последними параметрами намного меньше. Давайте приведем эту модель в действие в нескольких примерах.

9.2 Пример 1: отыскание минимума

Довольно часто возникает потребность найти минимальное значение в наборе значений. С точки зрения традиционных алгоритмов для этого требуется линейное сканирование элементов в списке, в котором хранятся элементы. Теперь рассмотрим аналогичный вариант использования для внешней памяти.

9.2.1 Вариант использования: минимальный медианный доход

Допустим, вы работаете в стартапе, который моделирует демографию и данные переписи населения и визуализирует их для своих пользователей (например, Social Explorer в **www.socialexplorer.com**). В большом числе таблиц содержатся агрегированные данные, при этом отдельные записи в табличных данных о доходах агрегируются до уровня демографического блока. Мы будем различать демографические блоки и дисковые блоки, называя их соответственно *демографическими блоками* и просто *блоками*. Территория США разделена на более чем 10 млн демографических блоков, и таблица каждого демографического блока содержит значительный объем информации, упорядоченной последовательно по блокам. Одна из переменных включает медианный доход, и мы заинтересованы найти демографический блок с минимальным медианным доходом во всех США.

Фактически у нас несортированный массив из N целочисленных записей на диске, и нам нужно найти минимальное значение. В мире «обычных» алгоритмов будет достаточно простого цикла `for`, требующего O(N) сравнений. Если применить аналогичный подход, когда данные расположены на диске, то вместо этого мы берем блоки данных, начиная с того места, где начинаются данные, вплоть до самого последнего блока, содержащего любой из наших элементов. Рассмотрим две схемы чтения данных в оперативной памяти и на диске в псевдокоде.

Вот как мы читаем данные из оперативной памяти:

```
min = INT_MAX
for i in range(N)
    if (A[i] < min)
        min = A[i]
```

Вот как мы читаем данные с диска:

```
BLOCK_SIZE = 1024
min = INT_MAX
for i in range(ceil(N/BLOCK_SIZE))
    BLOCK = read_block(filename, file_start + i*BLOCK_SIZE, BLOCK_SIZE)
    for i in range(BLOCK_SIZE):
        if (BLOCK[i] < min):
            min = BLOCK[i]
```

Второй фрагмент псевдокода читает блок данных, указывая имя файла (оно сообщает стартовую позицию файла), смещение, то есть позицию внутри файла, и размер читаемого блока, начиная с этой позиции. В приведенном выше псевдокоде принята пара упрощающих допущений, например что файл начинается (`file_start`) прямо на границе блока, что бывает не так. Поскольку диск подразделен на блоки памяти, а блоки имеют, как правило, довольно большой размер, нет никакой гарантии, что наш файл будет начинаться с начала блока. В функции `read_block` принято допущение, что если искомая позиция находится в середине блока, то функция должна подхватить блок, содержащий нужный нам элемент.

Несколько замечаний о программировании во внешней памяти: например, при работе с крупными файлами на Python можно использовать ряд библиотек, которые позволяют выполнять системные вызовы, такие как `open` (для открытия файла), `seek` (для поиска той или иной позиции в файле) или `write` (для записи в ту или иную позицию в файле). Грубо говоря, `seek` соответствует ранее упомянутому дорогостоящему чтению в рамках операций ввода-вывода, но отслеживать то, как операционная система перетасовывает блоки туда-сюда, очень трудно. Операционная система имеет большое число встроенных оптимизаций, которые работают «под капотом», решая проблемы ввода-вывода. Например, если операционная система замечает, что мы последовательно обращались к нескольким блокам, то она может доставить несколько блоков, которые следуют за ними, еще до того, как мы их запросим, исходя из того, что это то, что мы, возможно, захотим сделать дальше. Кроме того, когда мы считываем блок в основную память и затем его изменяем, операционная система может отказаться от его немедленной записи обратно на диск и вместо этого предпочесть буферизовать его в оперативной памяти и записать обратно позже с рядом других блоков в последовательности.

Это большая и важная тема, но, рассказывая об алгоритмах внешней памяти, мы стремимся добиться понимания концепций с абстрактной точки зрения и связанных с ними алгоритмических хитросплетений. С этой целью мы показываем примеры в Python-подобном псевдокоде, в котором физически берем те или иные блоки, чтобы показывать работу алгоритма, даже несмотря на то, что в реальном исходном коде это обычно пишется не так. Мы считаем, что такой упрощенный взгляд способствует тому пониманию, к которому мы стремимся.

Теперь вернемся к нашему примеру. Когда мы последовательно сканируем блоки на диске, мы вводим блоки один за другим в основную память. Сразу после прочтения блока он нам больше не нужен, поэтому в любое время в памяти нужен только один блок одновременно, а также переменная `min`, которую мы соответствующим образом обновляем. На рис. 9.2 показан процесс на игрушечном примере, где $N = 11$, $M = 6$ и $B = 3$. Поскольку наши данные занимают не более N/B блоков, алгоритм требует $O(N/B)$ переносов в память (или операций ввода-вывода).

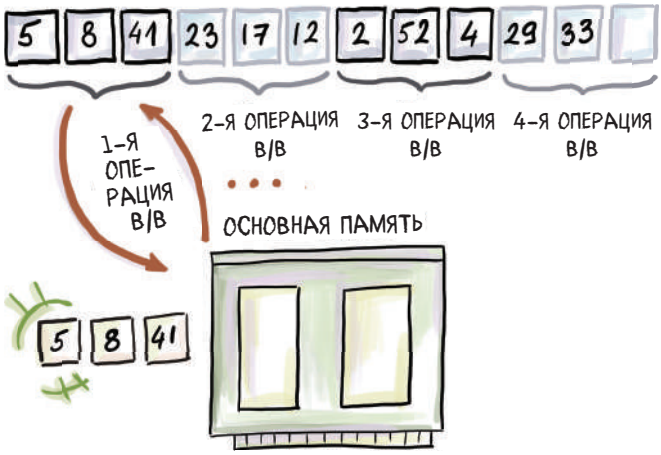


Рисунок 9.2 Отыскание минимума во внешней памяти. Всего имеется 11 элементов, которые занимают четыре блока размера 3. Последний блок не полон. В данном примере начало файла поблочно выровнено

Здесь мы подходим к первому различию во времени выполнения в рамках границ оперативной и внешней памяти (как показано на рис. 9.3). В мире внешней памяти «линейное время» естественным образом становится O(N/B). Это стоимость линейного витка по поочередно упорядоченным данным. Хорошо, если мы сможем достичь часть $/B$, поскольку в наихудшем случае, если данные не упорядочены последовательно или если мы обращаемся к ним случайным образом, то для чтения N элементов может потребоваться до O(N) операций ввода-вывода.

Рисунок 9.3 Разница в границах между алгоритмом в оперативной памяти (слева) и эквивалентным алгоритмом на диске (справа). Очень важно не сравнивать их напрямую, поскольку они представляют разные единицы измерения. Сравнение границ необходимо для того, чтобы понять изменения в алгоритмах вследствие другой конструкционной схемы

Упражнение 9.1

Используя операции Python `open`, `seek`, `read`, `write`, `close` и т. д., создайте файл с 1 млрд целых чисел (по одному в строке) и сохраните его на диске.

Затем используйте те же вызовы, чтобы выполнить следующие ниже задания:

1) вычислить сумму первых 1 млн целых чисел;
2) вычислить сумму случайно выбранных 1 млн целых чисел.

При выполнении обоих заданий засеките время задания по передаче данных (например, вызова `seek`) и вычислительного задания (суммирования данных). Сравните количество времени, которое требуется для каждого из них. Кроме того, сравните количество времени, необходимое для последовательного и случайного чтения. Если хотите, то в задании 2 засеките время, занимаемое вызовом `seek`. Занимает ли он одинаковое количество времени в каждой точке эксперимента, и если нет, то подумайте о причине, из-за которой это не так.

9.3 Пример 2: двоичный поиск

Теперь давайте посмотрим, как адаптировать наш старый добрый двоичный поиск к диску. Для этого есть несколько важных вариантов использования – всегда, когда нужно отыскать то или иное значение в упорядоченном файле, на ум приходит двоичный поиск. Поскольку двоичный поиск ведется по всему файлу, будет интересно посмотреть на число выполняемых переносов разных блоков. Но сначала рассмотрим следующий ниже вариант использования.

9.3.1 Вариант использования в области биоинформатики

Вы работаете специалистом по информатике в биоинформатическом стартапе и трудитесь над задачей секвенирования ДНК. В поставленной конкретной задаче вы получили большое число *K*-мер (подстрок длины *K* из ряда заданных последовательностей ДНК). Каждая *K*-мера имеет свое собственное строковое значение, а также небольшое число других ключевых свойств, важных для дальнейшего изучения. Данные представлены в виде одного кортежа (`K-мера, свойство1, свойство2, ...`) в каждой строке файла. Размер файла превысил 1 Тб. Значения *K*-мер сортируются и дедуплицируются, и вас интересует локализация тех или иных значений *K*-мер в файле. Данные статичны, поэтому вы не заинтересованы в изменении файла либо реорганизации в нем данных; вы просто хотите иметь возможность опрашивать файл максимально быстрым способом, поэтому прибегаете к двоичному поиску.

Как двоичный поиск будет работать в оперативной памяти по сравнению с поиском вне оперативной памяти? Давайте посмотрим на псевдокод и рис. 9.4.

Двоичный поиск в оперативной памяти:

```
binarySearch(arr, left, right, x)
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

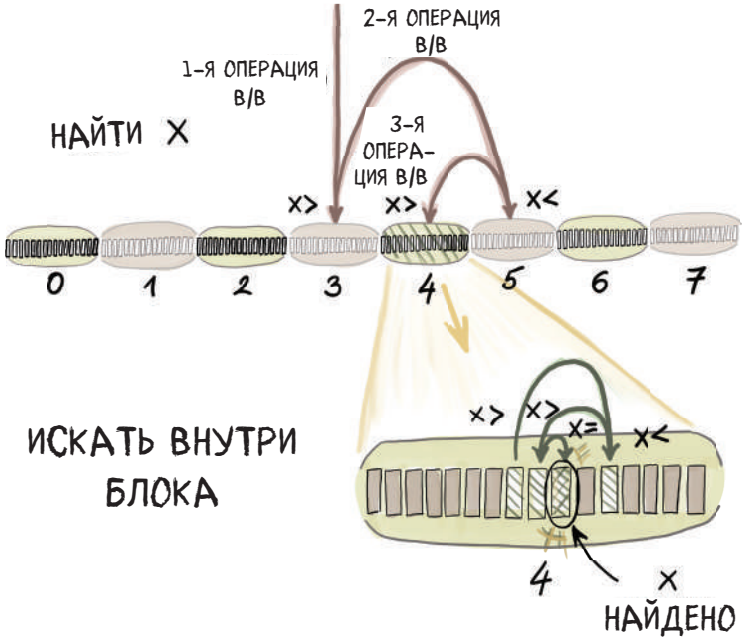


Рисунок 9.4 Двоичный поиск во внешней памяти. Алгоритм обращается к отдельному блоку по каждой имеющейся опорной точке, за исключением нескольких последних опорных точек, все из которых находятся в одном блоке

Двоичный поиск на диске:

```
BLOCK_SIZE = 1024                                                    ❶
def binarySearchExtMem(filename, file_start, left, right, x):
    while left + BLOCK_SIZE <= right:
        mid = left + (right - left) // 2
        BLOCK = read_block(filename, file_start + mid, BLOCK_SIZE):
        if BLOCK[BLOCK_SIZE - 1] < x:
```

```
            left = mid + 1
        elif BLOCK[0] > x:
            right = mid - 1
        else:
            return binarySearch(BLOCK, 0, BLOCK_SIZE - 1, x)
    BLOCK1 = read_block(filename, file_start + left, BLOCK_SIZE)
    return binarySearch(BLOCK1, 0, BLOCK_SIZE - 1, x)
```

❶ BLOCK_SIZE совпадает с B в тексте и анализе времени выполнения

Резидентный двоичный поиск (то есть в оперативной памяти) выполняет $O(\log_2 N)$ сравнений, а также такое же число чтений из ячеек памяти.

Версия того же алгоритма для внешней памяти изменена лишь незначительно. По сути, мы по-прежнему хотим выполнять ту же последовательность сравнений и выбираем блоки, содержащие элементы, которые мы хотим сравнивать. Опять же, здесь при выполнении `read_block(filename, file_start + mid, BLOCK_SIZE)` мы устраняем многие важные детали, потому что это, безусловно, не тот случай, когда `file_start + mid` всегда находится на границе блока. Сразу после того, как блок оказывается в основной памяти, выполняются следующие действия: если *x* меньше самого малого элемента блока, то мы приступаем к двоичному поиску в левой части массива, а если *x* больше самого большого элемента блока, то мы выполняем поиск в правой части. В остальных случаях мы вызываем функцию двоичного поиска в оперативной памяти, чтобы выяснить, не находится ли элемент в блоке.

Однако процесс продолжается только до тех пор, пока размер файла, в котором мы выполняем двоичный поиск, не станет больше блока. Как только размер массива становится меньше размера блока, мы вводим оставшиеся данные, которые умещаются в одном блоке, и все оставшиеся сравнения выполняются в оперативной памяти с использованием изначального алгоритма.

В примере на рис. 9.4 мы имеем $N = 128$, $M = 64$ и $B = 16$. Если бы в этом идентичном алгоритме двоичного поиска мы считали только число сравнений, то для отыскания искомого элемента нам потребовалось бы примерно семь сравнений. Поскольку нас интересует в основном подсчет перемещений блоков, то в худшем случае для поиска нам потребуется около 3 операций ввода-вывода.

9.3.2 Анализ времени выполнения

В большинстве случаев исполнения алгоритма, для того чтобы выполнить сравнение, которое направит нас к левой либо правой стороне массива, мы тратим 1 операцию ввода-вывода, что составляет $O(\log_2 N)$ операций ввода-вывода. Однако как только массив становится размером с блок, мы выполняем еще один ввод блока, и, таким образом, все последние $\log_2 B$ сравнений используют 1 операцию ввода-вывода. В общей сложности мы получаем $O(\log_2 N - \log_2 B + 1) = O(\log_2 N/B)$ операций ввода-вывода.

Важно проанализировать, что происходит, когда блок попадает в основную память. В асимптотическом смысле нам не помогает то, что мы сравниваем x с двумя граничными элементами блока, а не с одним элементом, как в изначальном алгоритме. Рассмотрим ввод первого блока. Сразу после того, как алгоритм решает, основываясь на двух сравнениях, перейти к левой либо правой стороне массива, у нас все еще остается $(N - B)/2 \approx N/2$ элементов. Тем не менее обследование граничных элементов в блоке или даже последовательное сканирование всего блока – разумное решение, учитывая, что блок уже находится в памяти. С практической точки зрения, эта поправка, скорее всего, будет влиять на производительность, даже если она не будет отражена в асимптотическом времени выполнения.

Теперь, когда мы проделали все это в каком-то смысле неуклюжим способом, приведем более естественный взгляд на алгоритм двоичного поиска во внешней памяти. Мы можем рассматривать алгоритм как блочный: вместо того чтобы искать точную позицию x в массиве, мы можем подумать о локализации правильного блока, в котором находится x (или где он должен быть, если он отсутствует). Следовательно, алгоритм выполняет двоичный поиск среди блоков, а не среди элементов. Как только элемент оказывается внутри внешних границ некоторого входного блока, исполнение алгоритма завершается. Это также упрощает анализ. У нас есть $O(N/B)$ блоков, и каждый шаг двоичного поиска стоит 1 операцию ввода-вывода. В результате, как и раньше, мы получаем $O(\log_2 N/B)$ операций ввода-вывода.

Логарифмическое время выполнения в оперативной памяти представляет оптимальную границу поиска, будь то использование двоичного поиска в сортированном массиве либо прохождение вниз по сбалансированному дереву двоичного поиска логарифмической глубины. Однако сейчас возникает следующий вопрос: является ли аналогичный алгоритм двоичного поиска на диске, который выполняется за $O(\log_2 N/B)$ время, оптимальным механизмом поиска в файлах и базах данных на диске?

Если данные выложены просто в сортированном массиве и нам не разрешено каким-либо образом их реорганизовывать или перегруппировывать, то, разумеется, двоичный поиск – это лучший выход. Но если нам будет разрешено каким-либо образом предобрабатывать данные (например, переставлять элементы), чтобы обеспечивать более оптимальное время выполнения запроса, то мы сможем добиться намного большего, чем двоичный поиск. Читайте дальше, чтобы узнать, как это сделать, но прежде чем двигаться дальше, попробуйте выполнить вот это небольшое упражнение.

Упражнение 9.2

Создайте файл на диске с 1 млрд упорядоченных целых чисел, по одному в строке. Напишите программу на Python, которая открывает файл и выполняет в нем двоичный поиск. Используйте системные вызовы `readline()`, `seek()` и `tell()`, чтобы выполнить это задание. Выполните его на не-

скольких примерах. Засеките время выполнения разных частей программы и определите наиболее времязатратные из них.

9.4 Оптимальный поиск

Для того чтобы понять, почему двоичный поиск не является для нас оптимальным алгоритмом поиска во внешней памяти, достаточно рассмотреть любой блочный ввод, происходящий во время двоичного поиска. Даже если мы выполняем реально дорогостоящую операцию ввода-вывода, в результате которой в основную память попадают тысячи элементов, мы фактически используем только один или всего пару элементов из нее. Это все равно, что нанять автобус, который отвезет на работу вас (и только вас). Согласитесь, можно сделать и получше.

Для того чтобы увидеть, как это делается, взгляните на рис. 9.5a и N отсортированных там элементов. С учетом размера блока, $B = 3$, какая подборка элементов будет наилучшей для их упаковки в блок, который мы собираемся поместить в память первым? Так вот, это будут три элемента, которые делят массив на четыре равные части; в данном конкретном примере это элементы 15, 31 и 40. Если бы мы создали такой блок, то после его ввода для поиска у нас остался бы массив размером не N/2, а $N/(B + 1)$. Аналогичным образом можно рекурсивно продолжить работу с каждым оставшимся подмассивом, отбирая B равноудаленных опорных точек. Для того чтобы понять, как строить эти блоки, рассмотрим строительство неявного дерева двоичного поиска поверх сортированного массива (рис. 9.5b), а затем, начиная с вершины дерева, будем группировать верхние B узлов дерева, чтобы формировать один узел (рис. 9.5c). Благодаря такому подходу мы создадим поисковую структуру данных, подобную той, что показана на рис. 9.5d, в которой узел позволяет ответвляться на $B + 1$ (в данном примере четыре, но в реальном мире тысячи) разных направлений, основываясь только на вводе одного блока. Чем выше ветвление, тем меньше уровней в дереве. Каждый уровень представляет собой одну операцию ввода-вывода, подлежащую выполнению, поэтому более высокий коэффициент ветвления сокращает число операций передачи данных в память.

В целях понимания разницы в этом небольшом примере число сравнений, которые нужно выполнить во время двоичного поиска, эквивалентно глубине дерева двоичного поиска в 9.4b, то есть равно четырем сравнениям. Поскольку последние два уровня дерева будут находиться в одном блоке, то нам нужно 4 – 2 + 1 = 3 блочных ввода, если использовать обычный двоичный поиск. Но если мы используем новую структуру, показанную на рис. 9.4d, то понадобятся всего два блочных ввода, поскольку эта структура имеет всего два уровня.

Разница кажется тривиальной, так как набор данных невелик и, что важнее, параметр B невелик. Но на самом деле разница огромна: если при двоичном поиске число операций ввода-вывода равно $O(\log_2 N)$, то при

использовании структуры из рис. 9.5d необходимое число операций ввода-вывода составляет $O(\log_B N)$ операций.

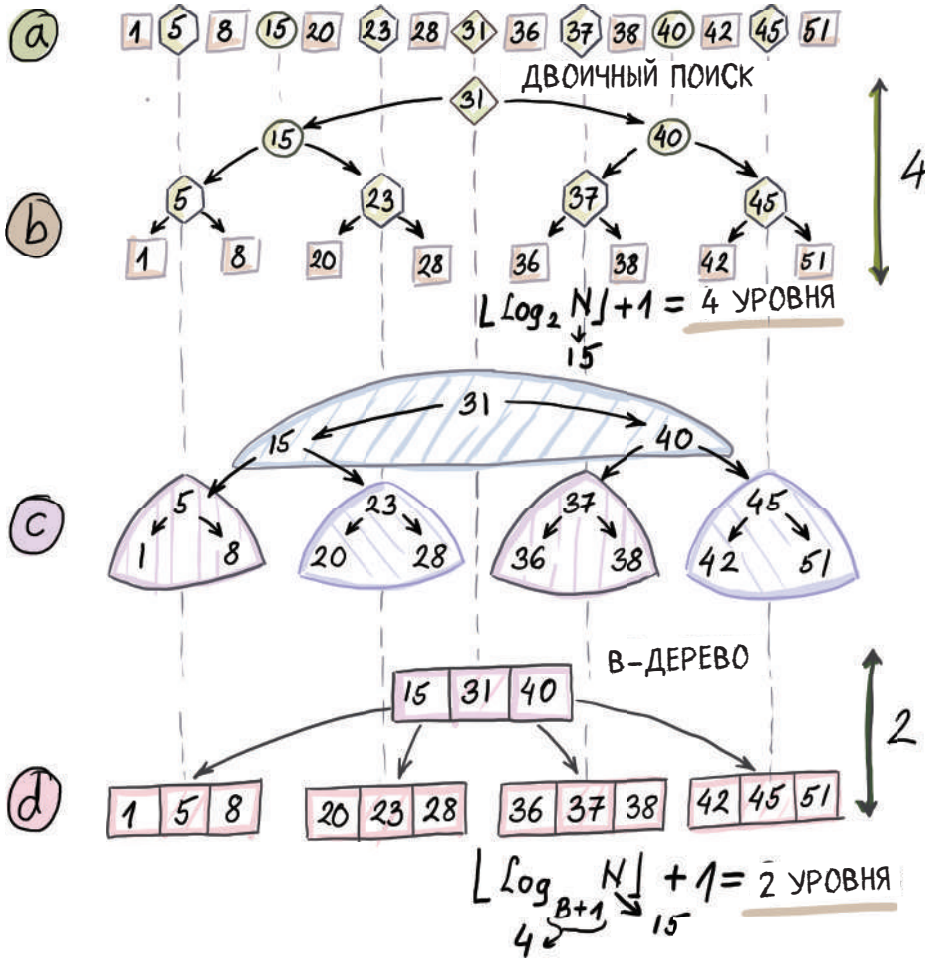


Рисунок 9.5 Как преобразовать сортированный массив в структуру, обеспечивающую оптимальный поиск во внешней памяти (*B*-дерево). Оптимальная поисковая структура данных в оперативной памяти выглядит примерно как *d*, где у каждого узла есть одна опорная точка, основанная на выбранном узлом пути движения вниз по дереву. Оптимальная поисковая структура данных во внешней памяти имеет большие узлы (размером с блок), которые содержат много элементов, потому что данные извлекаются в память поблочно

Обычно основание логарифма асимптотически не имеет значения, если оба основания постоянны. Однако здесь основание логарифма, равное B, имеет гигантское значение.

Возьмем набор данных размером 1 млрд ($\approx 2^{30}$) и блок, который может вместить 1000 ($\approx 2^{10}$) элементов. Например, размер блока составляет 64 Кб,

а каждый элемент занимает 8 байт. Это означает, что в двоичном поиске нужно ≈ 20 блочных вводов, а в новой структуре – только 3.

Структура на рис. 9.5d представляет собой мультяшную версию так называемого *B*-дерева. *B*-дерево образует костяк индексов баз данных в большинстве крупных реляционных баз данных. Несмотря на огромный объем данных, глубина *B*-деревьев редко превышает пять-шесть уровней, в силу этого ограничивая число операций ввода-вывода, которые необходимо выполнять для выполнения запроса (см. рис. 9.6).

Рисунок 9.6 Еще одно отличие в том, как могут выглядеть границы в модели оперативной и внешней памяти. Нередко время выполнения с логарифмом по основанию 2 может превращаться в логарифм по основанию *B*, так как с помощью одного блока можно обследовать *B* разных элементов одновременно. Как мы увидим далее, основание 2 не всегда превращается в основание *B*

В следующей главе мы рассмотрим гораздо больше деталей о *B*-деревьях, их разных реинкарнациях и других структурах данных, которые используются в реляционных базах данных.

Обратите внимание, что до сих пор размер памяти не имел особого значения, лишь бы в ней помещался хотя бы один блок и имелось некоторое дополнительное пространство для хранения нескольких переменных. При сканировании, двоичном поиске или *B*-поиске нужно было вводить много блоков, но по одному за один раз. Однако есть задачи, в которых важен размер памяти и в которых алгоритм внешней памяти может эффективно ее использовать.

9.5 Пример 3: слияние *K* сортированных списков

Давайте обратимся к задаче слияния данных во внешней памяти. Например, популярная задача слияния данных из разных источников сводится к слиянию нескольких списков. В основной памяти она часто решается с по-

мощью кучи, которая многократно извлекает минимумы из тех или иных списков. Далее мы увидим, как задача слияния K сортированных файлов решается во внешней памяти и что это говорит о природе одновременного слияния большого числа списков в оперативной памяти по сравнению с внешней памятью. Эта информация окажется важной позже, когда мы начнем адаптировать сортировку слиянием к внешней памяти.

9.5.1 Слияние журналов времени/дат

Вы работаете в компании, продуктом которой является балансировщик нагрузки, поддерживающий многосерверные приложения с высоким трафиком. Приложение также обладает значительным компонентом безопасности и собирает данные о трафике на множестве разных веб-сайтов. Вас интересует ответ на вопрос, существует ли какая-либо связь между временем и датами совершения определенных атак, поэтому вы анализируете большое число журналов событий, собранных на разных веб-сайтах, каждый из которых отсортирован по времени/дате. Ключевым шагом является слияние файлов в порядке возрастания времени/даты в один гигантский файл. Ваш локальный компьютер имеет 16 Гб оперативной памяти, а общий размер файлов составляет 1 Тб, который распределен между 16 000 разных файлов.

Задача сводится к слиянию K сортированных списков. Давайте допустим, что все списки, вместе взятые, содержат N элементов. Прежде чем перейти к версии, в которой подлежащие слиянию файлы находятся на диске, давайте вспомним решение этой задачи в оперативной памяти.

Версия для оперативной памяти

Для эффективного слияния K сортированных списков в оперативной памяти можно задействовать минимум-ориентированную кучу (min-heap), которая содержит по одному представителю из каждого сортированного списка (в общей сложности K элементов), и извлекать минимальные элементы по очереди. Как только элемент покидает кучу в качестве ее минимума, элемент из его списка поступает обратно в кучу. Индивидуальные операции в куче размером K стоят $O(\log_2 K)$, и поскольку каждый элемент должен в какой-то момент вставляться в кучу размером не более K и удаляться из нее, на решение этой задачи потребуется в общей сложности $O(N \log_2 K)$ сравнений.

Версия для внешней памяти

Теперь давайте посмотрим, что произойдет, если N слишком велико, чтобы уместиться в оперативной памяти. В этом примере мы будем считать, что хотя общий размер файлов и даже каждого отдельного файла может быть слишком большим, чтобы уместиться в ОЗУ, число файлов достаточно мало, чтобы вместить в ОЗУ по одному блоку данных из каждого файла и при этом оставить свободной половину памяти. Другими словами, мы

исходим из допущения, что $K \leqslant M/2B$. Это не очень строгое допущение, поскольку в некоторых распространенных аппаратных конфигурациях оно позволяет иметь до миллиона списков.

Мы воспользуемся этим фактом, резервируя по одному блоку памяти под каждый файл. Вначале мы будем читать первый блок каждого файла.

Теперь можно задействовать резидентное решение для блоков, которые мы только что прочитали. Мы начинаем с того, что каждый блок вставляет свое минимальное значение в минимум-ориентированную кучу. Затем мы начинаем извлекать минимумы из кучи. При каждом извлечении минимума мы добавляем следующий элемент в кучу из того же блока, из которого минимум был взят. Как только у нас заканчивается один блок, мы добавляем следующий блок из того же файла, пока не дойдем до конца файла. Весь процесс показан на рис. 9.7, а псевдокод иллюстрирует более подробную информацию.

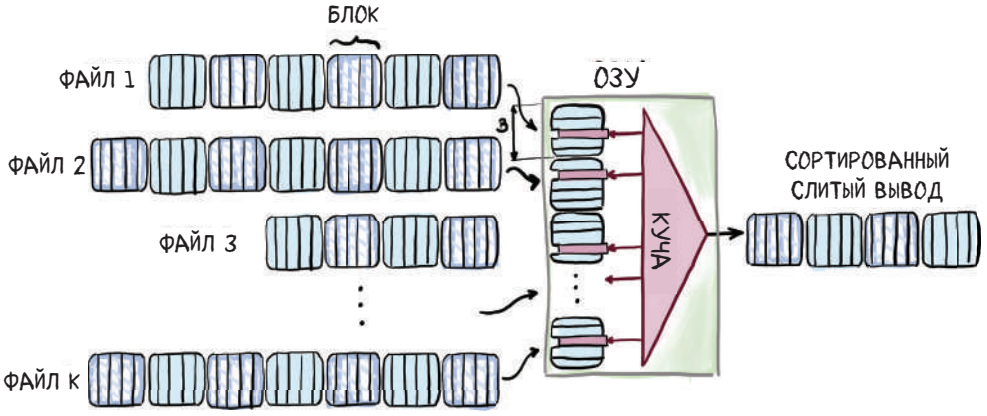


Рисунок 9.7 *K*-путное слияние сортированных файлов. Каждому файлу отведен один блок данных в основной памяти, и через этот блок каждый файл отправляет оставшиеся минимумы в кучу. Минимумы многократно извлекаются из кучи и отправляются в вывод. Эта схема работает независимо от общего накопленного размера файлов, при условии что $K = O(M/B)$

В псевдокоде список `file_names` содержит имена файлов и позволяет обращаться к стартовым позициям каждого файла, а `files_loc` – это список, содержащий текущую позицию, в которой мы находимся внутри каждого файла. Список `buffer_in` хранит индивидуальные блоки в основной памяти (всего их K), и каждый блок может быть проиндексирован как список из `BLOCK_SIZE_ELEMENTS` числа элементов, поэтому список `buffer_in` можно трактовать как двумерный. Список `buffer_out` хранит уже слитые элементы, готовые к выводу, и как только он будет содержать `BLOCK_SIZE_ELEMENTS` число элементов, мы записываем этот блок в местоположение `file_dest` с именем выходного файла `outfile_name`.

Список file_processed указывает на каждый сливаемый файл и на то, все ли его элементы уже израсходованы. Список merge_pos обозначает элемент, на который мы в настоящее время указываем при слиянии мини-списков размером в *K* блоков в оперативной памяти. Как только счетчик достигнет максимального числа элементов в блоке, мы инициируем чтение нового блока из соответствующего файла, если только мы уже не находились в конце файла:

```
BLOCK_SIZE = 1024
ELEMENT_SIZE = 64
BLOCK_SIZE_ELEMENTS = BLOCK_SIZE / ELEMENT_SIZE

buffer_in = []
buffer_out = []
file_processed = []
merge_pos = []
file_dest = 0                                                          ❶
for i in range(K)
    file_processed[i] = False
    files_loc[i] = 0
    buffer_in[i] = readBlock(files_names[i], files_loc[i], BLOCK_SIZE) ❷
    files_loc[i]+=BLOCK_SIZE                                           ❸

for i in range(K)
    H.insert(tuple(buffer_in[i][0], i))                                ❹
    merge_pos[i]=1

while(!H.empty())
    element, i = H.extractMin()
    buffer_out.append(element)
    if buffer_out.size == BLOCK_SIZE_ELEMENTS:                         ❺
        flushBlock(outfile_name, file_dest, buffer_out, BLOCK_SIZE)
        file_dest+=BLOCK_SIZE
        buffer_out.clear()
    if(!file_processed[i]):                                            ❻
        H.insert(buffer_in[i][merge_pos[i]])
        merge_pos[i]+=1
        if merge_pos[i] == BLOCK_SIZE_ELEMENTS && files_loc[i]!=EOF:   ❼
            readBlock(files_names[i], files_loc[i], BLOCK_SIZE)
            merge_pos[i] = 0
            files_loc[i]+=BLOCK_SIZE
        elif file_loc[i] == EOF
            file_processed[i] = True
```

❶ K – это число сливаемых сортированных списков
❷ Читать первый блок каждого списка в основную память
❸ Переместить позицию в файле вперед

❹ Куча H хранит пары (элемент, индекс списка)
❺ Если имеется полный блок слитых элементов, то очистить блок
❻ Случай, когда мы не израсходовали файл до конца
❼ Случай, когда мы достигли конца читаемого блока, но не конца файла

Обратите внимание, что в этой задаче мы используем большой объем оперативной памяти для одновременного слияния большого числа файлов. Время выполнения анализируется довольно просто, так как для каждого блока выполняется всего 1 операция ввода-вывода. В результате мы получаем O(N/B) операций ввода-вывода, включая все записанные блоки.

Это интересный артефакт внешней памяти, потому что во внутренней памяти мы никогда не смогли бы слить более чем постоянное число сортированных списков за линейное время. С другой стороны, во внешней памяти мы можем слить большое (до M/B) число сортированных списков за один линейный проход по входным данным (см. рис. 9.8).

Рисунок 9.8 Разница в границах слияния большого числа сортированных списков в основной памяти и во внешней памяти. Слияние многочисленных сортированных списков во внешней памяти можно выполнить всего за один виток по входным данным. Слияние большего, чем постоянное число списков во внутренней памяти, приводит к более чем линейной стоимости при ряде сравнений. Та же стоимость внутренней памяти сохраняется и в алгоритме K-путного слияния во внешней памяти; и это не самая важная стоимость

Что произойдет, если мы не сможем отводить по одному блоку на каждый файл в основной памяти? Мы оставим этот случай для главы 11, где вернемся к слиянию многочисленных списков в качестве контекста для алгоритма оптимальной сортировки во внешней памяти.

Будем надеяться, что, познакомившись с парой примеров того, как при переходе от оперативной памяти к внешней все совершенно меняется, вы смогли развить интуитивное понимание аспектов производительности, улавливаемых моделью внешней памяти. Однако этой модели не удается отражать важные аспекты, связанные с вводом-выводом. В следующем далее разделе мы обсудим, в чем заключаются эти различия и насколько они важны для правильного предсказания эффективности реального онлайнового приложения.

9.5.2 Модель внешней памяти: простая либо упрощенческая?

Первое, что визуально бросается в глаза при взгляде на изображенную модель внешней памяти (рис. 9.1), – это то, что она содержит только два уровня памяти: оперативную и дисковую. Как мы знаем, компьютерная иерархия гораздо сложнее и содержит много уровней памяти. Однако это не должно обескураживать. Каким бы ни был размер набора данных, мы всегда можем найти наименьший уровень, на который данные могут уместиться, и назвать его «диском», тогда как все остальные меньшие уровни будут образовывать оперативную память. На параметры размера блока B и размера памяти M будет влиять то, где внутри иерархии памяти вписывается размер набора данных. Например, если наши данные умещаются в основной памяти, но не могут уместиться в кеше, то данные между этими двумя уровнями передаются в строках кеша, которые меньше дисковых страниц/блоков.

Изначальная модель внешней памяти также допускает возможность совместного использования базы данных на многочисленных дисках и в этом смысле обеспечивает параллельную передачу данных. Если имеется P дисков, то в большинстве алгоритмов всю стоимость производительности можно разделить на P.

Некоторые упрощающие допущения модели внешней памяти, такие как бесконечность дискового пространства и полная свобода вычислений в оперативной памяти, не отражают реальности. Однако дисковое пространство чрезвычайно дешево, и пренебрежение производительностью центрального процессора будет влиять на нас не столь сильно, как выполнение ненужных операций поиска на диске.

Одним из важных недостатков модели внешней памяти является игнорирование соотношения последовательной и случайной производительности. Читаем ли мы x последовательных блоков или x блоков в совершенно разных местах диска, стоимость остается равной x операциям ввода-вывода. Для большинства технологий хранения данных это далеко не так. Отчасти это объясняется тем, как все устроено с аппаратной точки зрения, а также различными оптимизациями операционной системы. Например, нередко, если мы выполняем последовательное чтение нескольких блоков, то операционная система замечает это и пытается предварительно доста-

вить следующий блок. Несмотря на свои несовершенства, модель внешней памяти на сегодняшний день остается самой популярной моделью для проведения анализа производительности алгоритмов в контекстах интенсивного использования данных.

9.6 Что дальше

В этой главе мы начали отвечать на вопрос о том, как оптимально выполнять запросы на диске, и начали знакомиться с общей идеей *B*-деревьев. Однако их разные реализации и варианты оставлены для следующей главы. В частности, мы планируем ответить на следующие вопросы:

- Какие варианты *B*-деревьев существуют и реализованы в реальных системах?
- Как добавлять, удалять и изменять элементы в *B*-дереве и каковы механизмы для этого?
- Как узнать, что поиск по *B*-дереву оптимален во внешней памяти?
- Является ли *B*-дерево оптимальной структурой данных для вставок и изменений, а также поиска?

Вопрос об оптимальности B-деревьев для вставок также побудит к введению двух других структур данных, о которых мы узнаем: B^{ε}-деревья и LSM-деревья; эти две структуры данных ориентированы на базы данных, оптимизированные под операции записи.

Резюме

- Многие приложения с интенсивным использованием данных хранят крупные объемы данных на диске. Для эффективной работы с крупными файлами и базами данных, хранящимися на диске, требуются другой набор структурных данных и алгоритмические приемы.
- Модель внешней памяти – это полезный инструмент для анализа алгоритмов, предназначенных для крупных наборов данных, которые не умещаются в основной памяти. В рамках этой модели принято допущение о том, что все данные изначально хранятся на диске бесконечного размера и для выполнения вычислений в основную память ограниченного размера заносятся (и выносятся) порции данных.
- Модель внешней памяти отказывается от вычислительной стоимости алгоритма, чтобы подчеркнуть стоимость передачи данных, которая, как правило, в 1000 раз дороже вычислительных операций в оперативной памяти.
- При сканировании последовательных данных алгоритм внешней памяти, как правило, содержит */B*-часть, означающую, что мы упаковали элементы в поочередные блоки.

- Алгоритм двоичного поиска, в отличие от его аналога для внутренней памяти, не является оптимальным алгоритмом поиска во внешней памяти, так как он не очень хорошо использует блочные вводы и выводы. Лучшего времени выполнения можно добиться за счет переупаковки блоков и построения структуры данных, именуемой *B*-деревом.
- Одним из главнейших преимуществ большой оперативной памяти является то, что всего за один линейный виток можно одновременно сливать большое число сортированных списков/файлов, независимо от суммарной величины всех файлов. Этот процесс также является основой для алгоритма оптимальной сортировки во внешней памяти, который мы изучим позже.

Глава 10

Структуры данных для баз данных: B-деревья, B^{ε}-деревья и LSM-деревья

Эта глава охватывает следующие ниже темы:

- изучение внутреннего устройства индексов баз данных;
- обследование структур данных, обитающих внутри MySQL, LevelDB, RocksDB, TokuDB и т. д.;
- введение в B-деревья и изучение принципов работы поиска, вставки и удаления в B-деревьях;
- понимание принципа работы B^{ε}-деревьев и буферизации как вспомогательного средства для операций записи;
- изучение принципа работы журнально-структурированных деревьев слияния (LSM-деревьев) и их преимуществ в производительности.

Правильный выбор подходящей базы данных для своего приложения требует некоторого понимания того, как строятся различные механизмы управления базами данных. В частности, в большинстве баз данных реализуются индексы, которые служат для ускорения поиска в больших и часто опрашиваемых таблицах. Обычно индекс помещается в один из столбцов, чтобы ускорять поиск по этому столбцу. Понимание всех последствий создания индекса для производительности предусматривает понимание того, как разные базы данных строят и поддерживают свои индексы.

В самых общих чертах индекс – это структура данных, обычно отдельная от самой таблицы базы данных, которая помогает эффективно перенаправлять запрос к нужной строке таблицы. Без индекса операция поиска сводится к линейному сканированию ключей в заданном столбце. Имея дело с системами, которые ежедневно записывают 10 млрд новых строк или около того, можно с уверенностью сказать, что линейный поиск с этим не справится.

В этой главе мы узнаем о трех наиболее распространенных структурах данных, используемых для построения индексов в современных системах хранения данных. Каждая структура данных оптимизирована под разный вид рабочей нагрузки с точки зрения соотношения между операциями поиска, вставки/удаления и другими важными операциями. Как обычно, стоимости этих операций будут расходиться друг с другом. В основе эффективных баз данных лежат структуры данных внешней памяти, и, на наш взгляд, они являются прекрасным примером всех алгоритмических хитростей и компромиссов, связанных с работой с данными на диске. Но прежде чем погрузиться на более глубокие, технические уровни внутреннего устройства баз данных, сначала давайте познакомимся с основными принципами работы индексации.

10.1 Принцип работы индексации

Индекс наиболее полезен, когда он строится на основе столбца, к которому мы часто обращаемся с запросами. В ответе на отправляемый запрос, возможно, потребуется вернуть всю строку, в которой ключ совпадает с ключом запроса, но для локализации записи нужен только ключ – то есть значение из обозначенного индексного столбца.

Рассмотрим следующий простой пример построения индексов, показанный на рис. 10.1. Дана таблица, в которой хранится информация о сотрудниках конкретного розничного магазина. В целях ускорения поиска можно создать индекс по столбцу `Имя`, и значения в указанном столбце для этого должны быть уникальными. Другими словами, если мы ищем `Джона`, то индекс должен выдавать одно-единственное местоположение в таблице, где можно найти строку с именем `Джон` (именно поэтому столбец `Имя` сам по себе плохо подходит в качестве основы для построения индекса).

Когда ни один столбец не имеет уникальных значений, можно использовать комбинацию (то есть конкатенацию) столбцов и строить индекс на ее основе. Как показано на рис. 10.1, чтобы ускорять поиск по двум или более независимым столбцам, можно создавать несколько индексов.

Одним из способов реализации индекса является построение отдельной от самой таблицы структуры данных, ключи которой лексикографически отсортированы для быстрого поиска (например, дерево поиска). Ключ в структуре данных – это столбец, на котором мы строим индекс, а значение – местоположение строки в таблице, содержащей данный ключ, как показано на рис. 10.1. Тогда запрос сначала быстро находит ключ в индексе, а затем использует указанное его значением местоположение, чтобы мгновенно извлечь соответствующую строку из таблицы.

То, что мы только что описали, иногда называют некластеризованным индексом, когда фактические данные таблицы не перестраиваются при построении индекса. Если индексов несколько, как в нашем примере, то они должны быть некластеризованными. С другой стороны, при построении кластеризованного индекса он упорядочивает данные внутри табли-

цы, поэтому в каждой таблице может быть только один кластеризованный индекс.

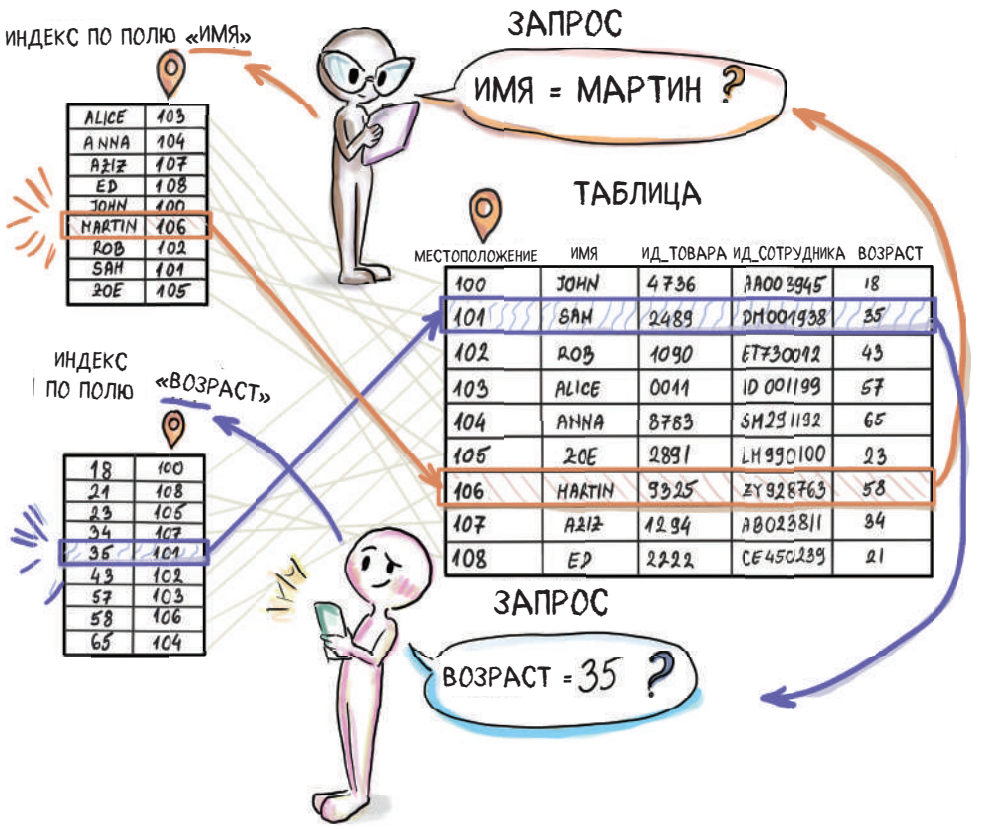


Рисунок 10.1 Формирование двух индексов поверх двух разных столбцов таблицы

Как вы понимаете, наличие одного индекса позволяет ускорять поиск по одному столбцу, но он совершенно бесполезен при поиске по другим критериям. Если в нашем примере из рис. 10.1 мы хотим выполнить запрос по возрасту сотрудника, то нам нужен совершенно новый индекс. Технически, конечно, можно было бы построить индекс на каждом столбце таблицы, просто на всякий случай, но наличие большого числа индексов быстро приводит к точке, откуда отдача начинает убывать. А именно при каждом обновлении таблицы (то есть вставке либо удалении строки или же изменении значения в столбце ключа) индекс тоже должен обновляться. Индексы ускоряют поиск, но замедляют все другие операции, которые изменяют содержимое таблицы. Иметь много индексов на таблицу можно только тогда, когда мы знаем, что данные не будут часто модифицироваться, когда скорость поиска гораздо важнее скорости обновления или когда данных не так много, чтобы беспокоиться о скорости.

Необходимость обновления индекса вместе с таблицей преподносит нам первый важный урок баз данных: стоимость поиска тесно переплетается со стоимостью вставки и удаления. Это не должно удивлять, поскольку мы видели, как аналогичные компромиссы происходят с резидентными индексами/словарями.

Однако в базах данных эта взаимосвязь будет гораздо более сложной, чем та, которую мы увидели на данный момент. Как объясняется в нескольких недавних технических статьях [1], в операцию вставки во многих системах встроена операция поиска. Например, при выполнении дедупликации операция вставки сначала делает запрос на наличие записи с тем или иным ключом и вставляет только в том случае, если ключ не найден. В этой ситуации в худшем случае суммарная стоимость вставки складывается из стоимости поиска плюс стоимость модификации в рамках операции вставки. Поэтому если бы мы настроили систему под молниеносно быструю вставку ценой невероятно медленного поиска, то были бы очень разочарованы итоговой производительностью вставки, увидев, что она включает стоимость поиска. Отыскание оптимальной производительности во многих реально-практических случаях, подобных этому, превращается в тонкую балансировку.

10.2 Структуры данных этой главы

В этой главе мы уделяем наибольшее внимание *B*-деревьям [2], поскольку они составляют основу большинства наиболее популярных движков баз данных, таких как PostgreSQL (**https://www.postgresql.org/docs/13/index.html**) и MySQL (**http://mng.bz/OG8n**). *B*-деревья во многом похожи на деревья двоичного поиска, но с огромными узлами, размер которых соответствует размеру страницы/блока на диске. Поскольку такие узлы могут вмещать большое число ключей, *B*-деревья имеют большой коэффициент ветвления (число дочерних узлов, которое иногда называют разветвлением по выходу[76]), что гарантирует малую глубину и, соответственно, отличную производительность поиска. Помимо изучения механики операций с использованием *B*-деревьев, в этой главе мы математически покажем, что *B*-деревья демонстрируют оптимальную производительность при поиске во внешней памяти. Поэтому неудивительно, что с момента разработки *B*-деревьев в 70-х годах они остаются самым популярным вариантом решения в части конструирования движков баз данных.

B-деревья пользуются всеобщим признанием за их быстрые операции поиска, однако можно получить структуру данных, в которой поиск выполняется лишь немного медленнее, а вставка и удаление имеют гораздо более высокую производительность, чем в *B*-дереве. B^{ε}-дерево [3] – это альтернативная структура данных, лежащая в основе систем хранения данных, таких как TokuDB, которые в последнее время стали популярными благода-

[76] Англ. fan-out. – *Прим. перев.*

ря своей превосходной производительности вставки/удаления. Поскольку данные становятся все более динамичными, многим приложениям необходимо поддерживать гораздо более высокую пропускную способность при вставке/удалении, чем могут предложить базы данных на основе *B*-дерева, при этом поддерживая быстрый поиск. B^ε-деревья являются идеальной структурой данных для таких типов рабочих нагрузок. B^ε-деревьям удается поддерживать ту же асимптотическую стоимость поиска при одновременном улучшении (асимптотически) стоимости вставки/удаления. Другими словами, в их операциях поиска ощущается некоторое замедление, но ускорение в операциях вставки и удаления ощущается гораздо сильнее.

Секрет производительности B^ε-дерева заключается в том, что вставки и удаления выполняются не сразу, как в *B*-деревьях, где одна вставка/удаление/модификация немедленно перемещается вниз к листу дерева и его изменяет (*B*-деревья просто слишком серьезно относятся к жизни). Вставки и удаления в B^ε-деревьях действуют как сообщения, которые буферизуются и задерживаются на пути к листьям. Идея в основе задержки операций заключается в сборе достаточного числа сообщений о вставках/удалениях на одном узле, которые направляются в одном направлении, а затем их отправке вместе в одной передаче в память; эта идея схожа с объединением вагонов, принадлежащих разным дорогам, в один состав. Объединяя вставки и удаления, B^ε-дерево может эффективно использовать свои операции ввода-вывода, чтобы обрабатывать как можно больше вставок и удалений. Это отличается от *B*-деревьев, в которых один элемент спускается вниз по дереву и инициирует множество дорогостоящих операций ввода-вывода исключительно для своей собственной выгоды.

Наконец, мы обсудим LSM-деревья [4]. LSM-деревья – это структуры данных, лежащие в основе LevelDB, RocksDB, Cassandra [5] и некоторых других движков, заботящихся только о высокопроизводительных вставках, которые выполняются быстрее, чем даже те, что находятся в B^ε-деревьях. Преимущество LSM-деревьев заключается в том, что они используют функциональность очень быстрого последовательного сканирования дисков. Если *B*-деревья и B^ε-деревья при спуске по дереву обращаются к случайным блокам, то LSM-деревья организуют свои данные в последовательные отрезки[77], которые эпизодически интегрируются, как при сортировке слиянием. Слияние двух отрезков может выполняться со скоростью сканирования (N/B для всех элементов или $1/B$ для каждого элемента), что является оптимальным, а эпизодическое слияние отрезков гарантирует, что в итоге мы не получим слишком много отрезков, которые придется запрашивать в нужный момент. Как бы то ни было, операции поиска в этой структуре данных действительно бьют по карману, но эту проблему можно несколько смягчить с помощью фильтров Блума.

[77] Англ. run; отрезок содержит данные, отсортированные по индексному ключу, и представлен на диске в виде одного файла либо набора файлов с ненакладывающимися диапазонами ключей.– *Прим. перев.*

10.3 *B*-деревья

B-деревья являются естественным расширением двоичных деревьев на внешнюю память: если деревья двоичного поиска используют по одному ключу на узел с целью ориентирования поиска/вставки/удаления в двух разных направлениях на следующем уровне дерева (<ключ и >ключ), то *B*-деревья используют гораздо больше ключей на узел. В остальной части изложения мы иногда используем термин *точка разворота*[78] взаимозаменяемо с ключом в узле *B*-дерева.

В частности, *B*-дерево порядка d имеет в каждом узле по меньшей мере d ключей (с коэффициентом ветвления $d + 1$) и не более $2d$ ключей (с коэффициентом ветвления $2d + 1$). Коэффициенты ветвления в узлах могут различаться в зависимости от числа ключей, которые они содержат. Единственным узлом, который не должен подчиняться требованию о минимальном числе ключей, является корень, который может иметь меньше, чем d ключей, но не более $2d$ ключей.

Значение d в узле *B*-дерева задает размер узла, поскольку у узла всегда есть пространство для размещения $2d$-ключей и $2d + 1$ указателей на поддеревья ниже, независимо от числа ключей, которое он хранит фактически. Как мы увидим, узлы обычно имеют немного пустого пространства.

Для того чтобы разобраться во внутренней структуре узлов *B*-дерева, взгляните на рис. 10.2, на котором показаны два узла *B*-дерева порядка 2. Слева мы видим минимально заполненный узел с двумя ключами и тремя указателями на непустые дочерние узлы. Справа мы видим полный узел с четырьмя ключами и пятью указателями на непустые дочерние узлы.

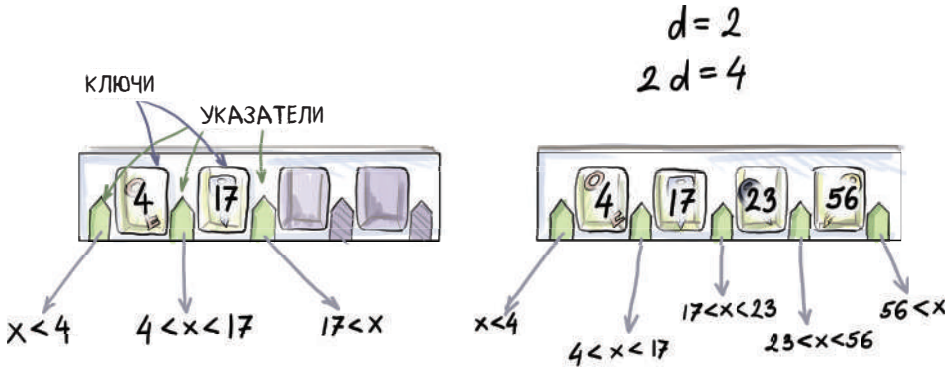


Рисунок 10.2 Структура узла B-дерева. У каждого узла есть пространство для размещения $2d$ ключей и $2d + 1$ указателей, независимо от его заполненности. Точки разворота направляют поиск; то есть при поиске элемента x мы сравниваем его со значениями в точках разворота, и следующий узел (то есть указатель на него) выбирается на основе значения ключа. Узел слева заполнен минимально, а узел справа – максимально

[78] Англ. pivot. – *Прим. перев.*

Каждый ключ внутри узла *B*-дерева, помимо значения, используемого для маршрутизации запроса на следующий уровень, также содержит указатель на местоположение строки в таблице, как показано на рис. 10.1. Начиная с рис. 10.2 эта деталь будет скрыта и будет показываться только ключ, поскольку в оставшейся части главы мы не будем обращаться к изначальным таблицам базы данных. Мы просто допустим, что в тот момент, когда мы локализуем ключ в узле дерева, который является ответом на запрос, у нас будет необходимая информация, чтобы автоматически перепрыгнуть к таблице и доставить остальную часть записи. Однако важно понимать, что внутри каждого узла *B*-дерева выполняется большая служебная и связующая работа, которая занимает значительную часть его пространства. Эти соображения важны при выборе размера узла. В целом на протяжении всей этой главы мы будем считать, что размер узла связан с размером блока *B*, поэтому *d* может рассматриваться как некая доля размера блока (например, *B*/2, *B*/4 и т. д.).

Рассмотрим случай, когда размер блока $B = 1024$. Если указатели, ведущие на следующий уровень дерева, занимают примерно половину пространства, то оставшаяся половина остается для ключей и указателей на таблицу. Давайте допустим, что это пространство снова разделено поровну, поэтому ключи занимают *B*/4 пространства, а указатели на таблицу занимают оставшиеся *B*/4. Несмотря на то что изначальный узел содержит 1024 слова и теоретически может хранить 1024 ключа, на самом деле он хранит до 256 ключей. Это все равно выгодная сделка, учитывая, что при максимальной заполненности всех узлов на четырех уровнях такое *B*-дерево может хранить более 4 млрд ключей на листовом уровне. На практике многие *B*-деревья имеют гораздо более крупные узлы, иногда порядка мегабайтов.

10.3.1 Балансирование *B*-дерева

В целях поддержания низкой стоимости поиска, как и в случае со сбалансированными деревьями двоичного поиска, мы должны быть уверены, что ни один путь от корня к листу в *B*-дереве не становится слишком длинным. У *B*-деревьев эта проблема прекрасно решена, поскольку каждый путь от корня к листу всегда имеет одинаковую длину. *B*-дерево – плоское снизу (то есть все листья находятся на одном уровне).

Операции вставки и удаления могут нарушать ограничения размера узла, например при вставке в полный узел или удалении из минимально занятого узла. Когда это происходит, переполненный узел можно расщеплять на две части, перераспределяя ключи, или же соединить два узла, у которых недостаточно ключей. Это может спровоцировать изменения вверх по дереву, требующие расщеплений/слияний на верхних уровнях, где в крайнем случае дерево может в итоге вырасти или уменьшиться с вершины. То есть мы могли бы в итоге расщепить корень на два узла, наложив новый корень сверху, или же опустить существующий корень на более низкий уровень, слив его с узлами под ним.

Если это звучит сбивчиво, то не волнуйтесь; вскоре эти операции будут наглядно показаны и подробно описаны. На данный момент важно понимать, что глубина *B*-дерева растет и уменьшается сверху, а все листья внизу остаются на одном уровне. Сравните это с деревьями двоичного поиска, где новый элемент вставляется в качестве листа снизу дерева, причем не требуется, чтобы все листья находились на одном уровне. Далее мы рассмотрим механику операций поиска и вставки/удаления.

10.3.2 Поиск

Алгоритм поиска довольно прост и имитирует логику поиска в дереве двоичного поиска. Выполняя поиск, мы сначала читаем в корневом узле *B*-дерева и находим местоположение ключа запроса в сортированном порядке среди корневых ключей. В случае если ключ равен одному из корневых ключей, то возвращается `True`; в противном случае мы следуем по соответствующему указателю вниз по дереву и применяем тот же алгоритм рекурсивно до тех пор, пока либо не вернем `True`, либо не достигнем указателя `null` и не вернем `False`. Если элемент найден до того, как он достигнет листьев, то спускаться по дереву дальше нет необходимости. В зависимости от реализации можно возвращать не булево значение, а хранящееся значение, но идея та же.

Поскольку верхние уровни дерева зачастую достаточно малы и сидят в оперативной памяти, можно было бы экономить некоторое число операций ввода-вывода при поиске на верхних уровнях дерева. Однако большинство ключей будут находиться на нижних уровнях, поэтому вероятность того, что запрашиваемый ключ находится в одном из узлов в оперативной памяти, довольно мала.

В наихудшем случае при поиске, возможно, потребуется выполнять чтение в каждом блоке на пути от корня к листу, что приведет к стоимости поиска $O(\log_d N)$ для *B*-дерева порядка d. Поскольку мы обычно исходим из допущения, что $d = \theta(B)$, это означает, что стоимость поиска в наихудшем случае будет равна $O(\log_B N)$. Тот факт, что некоторые узлы будут более пустыми, чем другие, не нарушит асимптотику, так как коэффициент ветвления по-прежнему будет равен $\theta(B)$.

Наихудший случай будет происходить, когда искомый элемент находится на листовом уровне, поэтому его отыскание предусматривает обследование каждого блока на пути от корня к листу. Наихудший случай часто возникает в ситуациях, когда считается, что большинство элементов сидит в листьях, а также когда поиск сообщает, что элемент отсутствует.

10.3.3 Вставка

Вставка несколько запутаннее, чем поиск. Сначала мы выполняем поиск, чтобы найти лист, в который данный элемент должен быть вставлен (мы всегда вставляем в лист). Если лист не заполнен (имеет < $2d$ ключей),

то элемент просто добавляется в нужную позицию в надлежащем листе, а измененный узел записывается обратно на диск. Рассмотрим пример, показанный на рис. 10.3, в котором 80 вставляется в *B*-дерево порядка $d = 2$. Помимо добавления элемента в лист, никаких других изменений в дереве не требуется.

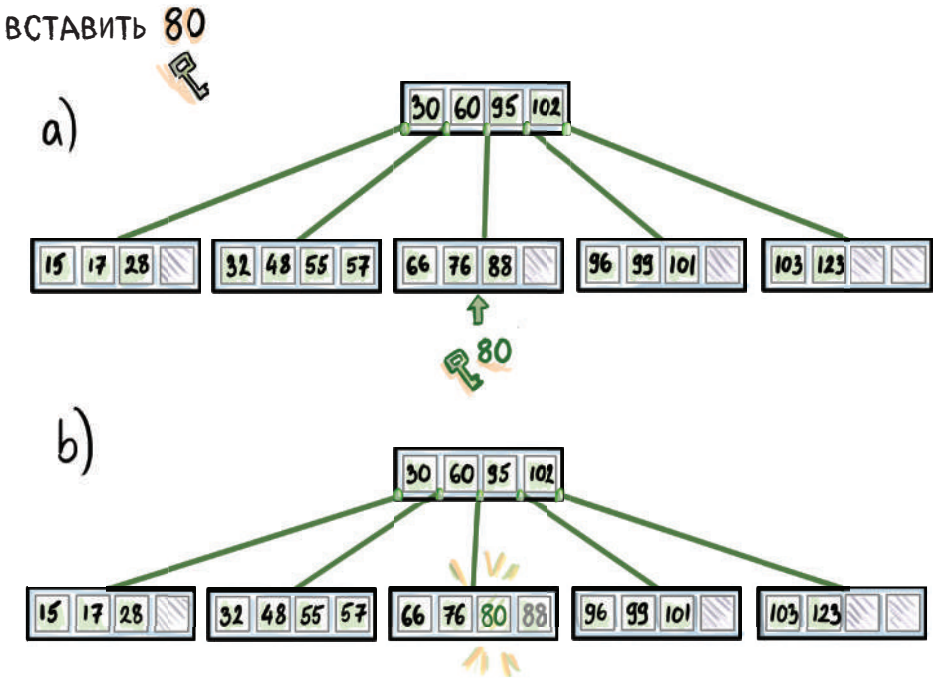


Рисунок 10.3 Вставка в *B*-дерево, когда указанный лист не заполнен

Однако может случиться так, что назначенный лист будет заполнен (у него уже есть $2d$ ключей), и тогда после размещения нового ключа у узла будет $2d + 1$ ключей. Поскольку теперь это нарушает ограничение по размеру, переполненный лист расщепляется на два листа следующим образом: левый лист будет содержать наименьшие d ключей, правый лист – наибольшие d ключей, а медианный ключ будет вставлен в родительский узел, чтобы служить разделителем для двух вновь созданных узлов. Этот процесс может спровоцировать дальнейшее расщепление вверх по дереву. Например, если все узлы на заданном пути от корня к листу заполнены, то все узлы будут расщеплены, поднимаясь к корню, включая корень, и дерево увеличится в высоту на 1.

Таков пример, показанный на рис. 10.4, в котором мы вставляем 69 в *B*-дерево порядка $d = 2$. После размещения 69 в лист он будет состоять из пяти элементов и будет расщеплен на два листа, которые будут отделены медианой 76; два элемента переходят в левый лист, и два элемента – в правый, а 76 вставляется в родительский узел, чтобы служить разделителем

между двумя вновь сформированными листами. В этом случае родитель является корнем дерева, и он тоже заполнен, поэтому вставка инициирует новое расщепление. Корень расщепляется на два узла, каждый с двумя ключами, а медиана переносится выше во вновь созданный корень.

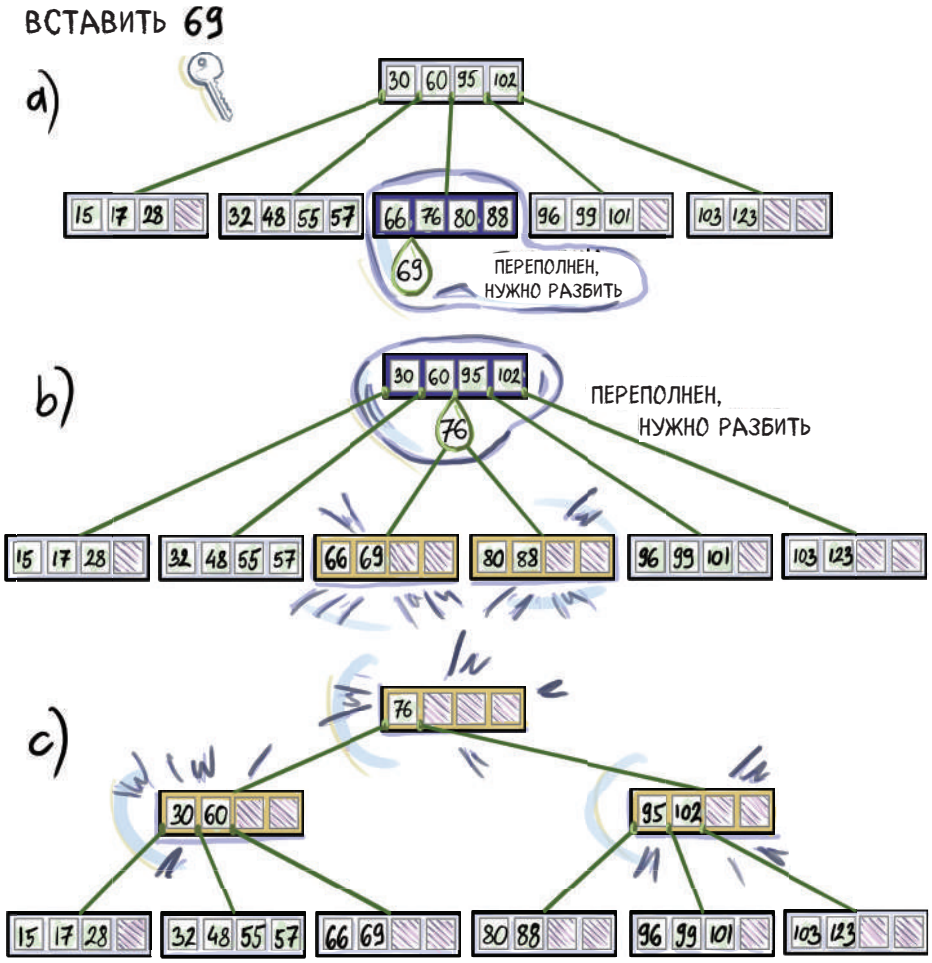


Рисунок 10.4 Вставка в заполненный узел, приводящая к увеличению глубины *B*-дерева на 1

Какова стоимость операции вставки? Общую стоимость вставки можно разделить на стоимость поиска – стоимость, необходимую для отыскания места вставки, – и стоимость модификации, включая расщепления узлов, перераспределение ключей и т. д. Стоимость поиска всегда равна $O(\log_B N)$ операциям ввода-вывода, так как при каждом поиске, требуемом вставкой, нужно обращаться к листу. Стоимость модификации варьируется в зависимости от дальности, с которой приходится выполнять расщепления вверх по дереву, но эта стоимость в наихудшем случае составляет O(1) операций

ввода-вывода в расчете на уровень дерева. Создание нового узла и перемещение по некоторым ключам требует не более чем доступа и записи в постоянное число блоков. Следовательно, в наихудшем случае стоимость модификации асимптотически не снижает общую стоимость вставки, поскольку она тоже, самое большее, равна $O(\log_B N)$.

Вместе с тем обратите внимание на то, что поскольку *B*-дерево имеет очень большие узлы, разрыв между минимально заполненным узлом (d ключами) и полным узлом ($2d$ ключами) довольно велик. Это означает, что расщепление узлов, инициированное вставкой в заполненный узел, происходит не так часто. Некоторые регулярности вставки могут вызывать большее число расщеплений узлов; например, многочисленные вставки в один и тот же лист будут приводить к нарушению *B*-дерева. Вот почему многие практические реализации *B*-дерева пытаются распознавать появление таких вставок и по-разному их обрабатывают (они вставляются одним большим пакетом и т. д.).

10.3.4 Удаление

Удаление элемента из *B*-дерева в некоторой степени аналогично вставке. Однако удаление имеет два следующих случая:

- удаление ключа из внутреннего узла;
- удаление ключа из листа.

Алгоритм удаления сводит оба случая ко второму случаю следующим образом: если ключ, подлежащий удалению, находится во внутреннем узле, то он удаляется из своего узла, а его преемник помещается на его место. Напомним, что преемником элемента x в дереве является наименьший элемент в дереве, который больше x.

Возможно, вы захотите на этом остановиться и убедиться, что у каждого ключа в нелистовом узле *B*-дерева есть преемник и что преемник произвольного ключа внутреннего узла находится в листе. Преемника ключа x можно найти во внутреннем узле, следуя по указателю p справа от ключа и находя минимум поддерева, на которое указывает p. Визуально, следуя по p, мы уходим один раз вправо, а затем продолжаем двигаться влево, пока не достигнем листа. Самый левый (наименьший) ключ в этом листе и есть преемник элемента x.

Заменяя элемент его преемником (на рис. 10.5 так 60 заменяется на 66), мы поддерживаем то же число ключей во внутреннем узле, из которого происходит удаленный элемент, а также поддерживаем лексикографический порядок элементов в дереве, так что там проблем нет. Однако мы только что потеряли элемент из листа.

Как удалять элемент из листа? Если лист y содержит больше ключей, чем d, то элемент можно безопасно удалить из листа, и на этом все (см. пример удаления 99 на рис. 10.5).

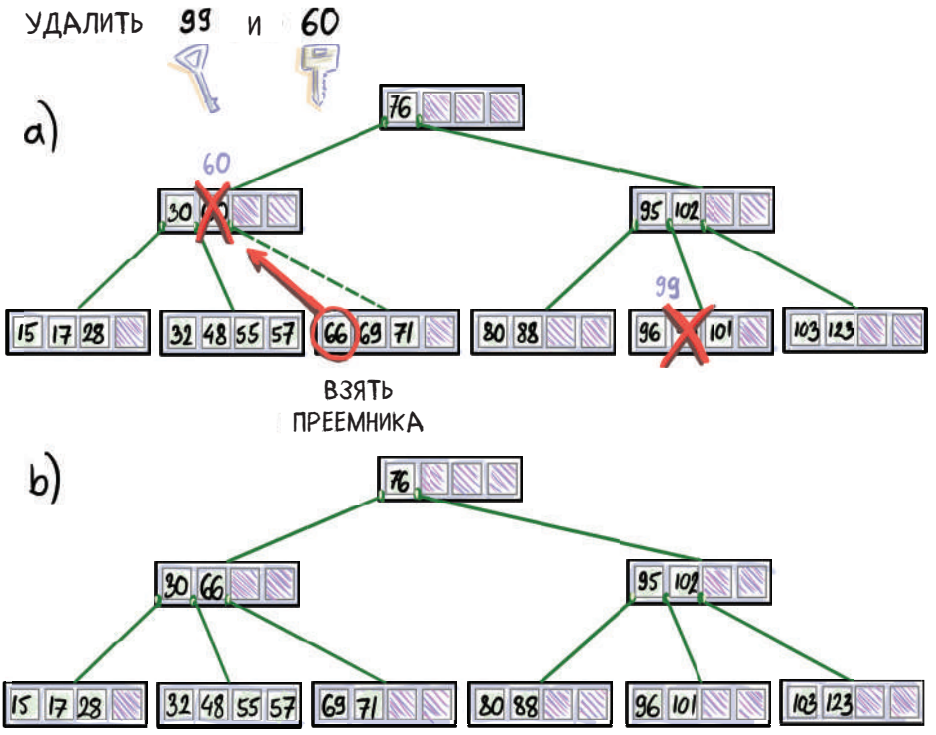


Рисунок 10.5 Обработка удалений из внутреннего узла по сравнению с листовым узлом

С другой стороны, если в листе *y* есть *d* ключей, то удаление ключа будет вызывать ошибку удаления из минимально занятого узла[79]. Тогда мы обращаемся к левому/правому соседу листа *y*, чтобы проверить наличие возможности позаимствовать у него несколько ключей. Если у левого либо правого соседа больше *d* ключей, то можно позаимствовать хотя бы один ключ, чтобы компенсировать ошибку удаления из минимально занятого узла.

В идеале если у одного из соседей имеется достаточный запас ключей, то нужно равномерно распределить ключи между этим соседом и листом *y*, но перестановка элементов между листами приводит к изменению разделителя. В примере, показанном на рис. 10.6, в котором мы удаляем 69, после удаления 69 левый узел содержит 32, 48, 55 и 57, разделитель равен 66, а правый узел содержит 71. Мы переставляем эти элементы таким образом, чтобы левый узел содержал 32, 48 и 55, разделителем стало 57, а содержимым правого узла стало 66, 71.

Возможно, это не совсем очевидно в *B*-дереве порядка $d = 2$, но равномерное перераспределение элементов между двумя листьями играет важ-

[79] Англ. underflow; син. потеря значимости. – *Прим. перев.*

ную роль (например, подумайте об узлах с тысячами ключей). Распределяя ключи равномерно, мы отодвигаем следующее потенциальное перераспределение ключей еще дальше в будущее.

Может случиться так, что оба соседа заполнены по минимуму своей емкости и не могут одалживать никаких ключей. В этом случае листья конкатенируются. Мы конкатенируем лист y (теперь он содержит $d - 1$ ключей) с соседом по нашему выбору (содержит d ключей) и более ранним разделителем между двумя листами, чтобы сформировать новый узел, содержащий $2d$ ключей, таким образом формируя полный узел. Опуская разделитель вниз, мы практически удаляем ключ из внутреннего узла, что может инициировать дальнейшее перераспределение ключей или конкатенацию узлов вверх по дереву.

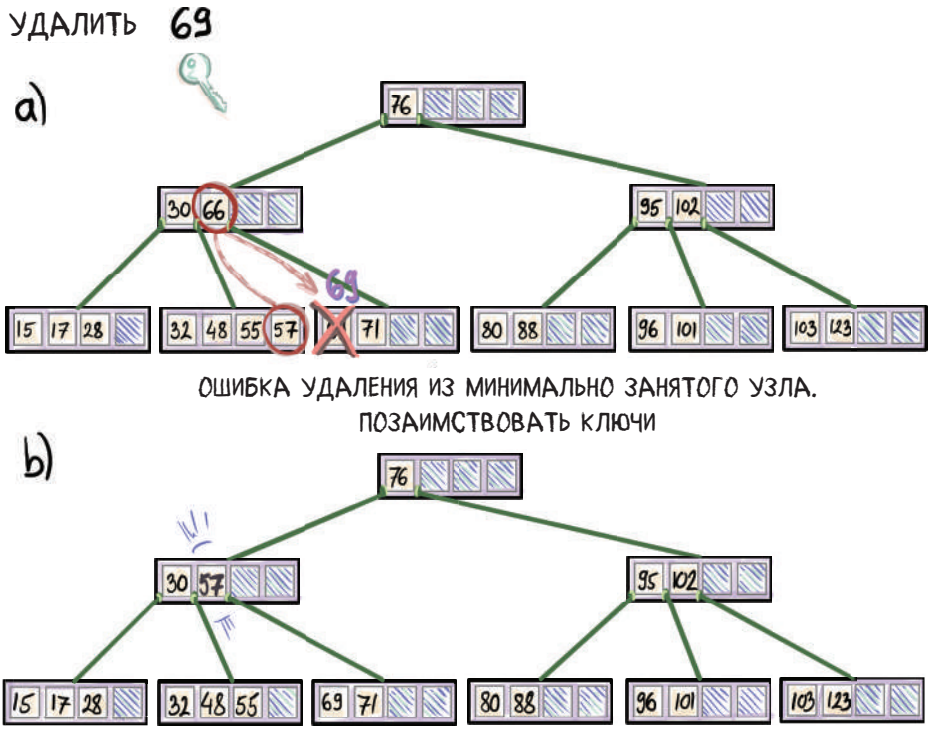


Рисунок 10.6 Удаление элемента может приводить к ошибке удаления из минимально занятого узла. Если соседи заполнены по минимуму своей емкости, то узел заимствует ключи у одного из соседей

Рассмотрим пример на рис. 10.7, где удаление 88 вызывает конкатенацию на листовом уровне. Сразу после конкатенации листа со своим правым соседом предыдущий разделитель двух листов (95) опускается в новый конкатенированный узел. За счет этого инициируется ошибка удаления из минимально занятого узла на втором уровне дерева, вызывая еще одну конкатенацию и в конечном счете уменьшая глубину B-дерева на 1.

Удаление, как и вставка, требует поиска удаляемого ключа и потенциально может потребовать модификации узла. Аналогично вставкам, стоимость модификации дерева во время удаления асимптотически не ставит под угрозу общую стоимость и составляет $O(\log_B N)$.

Несмотря на то что мы анализируем операции в *B*-дереве асимптотически, глубина *B*-дерева редко превышает 6 или 7 уровней. Верхние уровни *B*-дерева также зачастую могут умещаться в основной памяти. Например, для узла, в котором $d = 512$, а общий размер узла составляет порядка пары килобайт, стандартная оперативная память может вместить два-три верхних уровня дерева. За счет этого происходит экономия на операциях доступа, которые будут использоваться только для двух-четырех нижних уровней дерева.

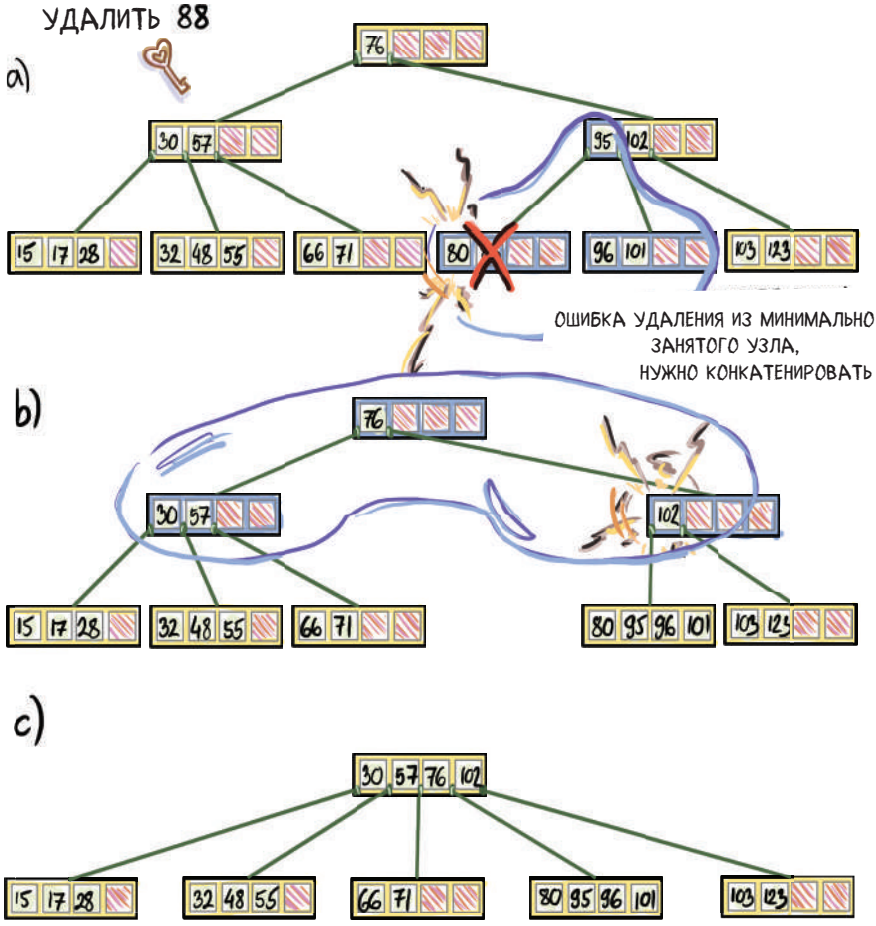


Рисунок 10.7 Удаление элемента из листа может приводить к конкатенации узлов, если соседние узлы заполнены по минимуму своей емкости. Конкатенации могут распространяться вверх по дереву и в конечном итоге уменьшать высоту дерева на 1

10.3.5 B^+-деревья

Большинство используемых сегодня реализаций B-деревьев на самом деле являются B^+-деревьями. Основное различие между простыми B-деревьями, которые мы только что описали, и B^+-деревьями заключается в том, что B^+-деревья хранят все свои данные в листьях. Внутренние узлы содержат ключи, главным предназначением которых является маршрутизация запроса к правильному листу, но они не обязательно отражают содержимое фактического набора данных. Это также означает, что все запросы стоят $O(\log_B N)$, поскольку даже если во время поиска мы обнаружим запрашиваемый ключ во внутреннем узле, мы все равно продолжим поиск до самого листа.

Есть несколько причин, по которым нам выгодно такое устройство. Во-первых, листовой уровень организован в виде связного списка, как показано на рис. 10.8. За счет этого обеспечивается быстрый последовательный доступ к данным в сортированном порядке и быстрые диапазонные запросы. Подумайте о диапазонном запросе или потребности просканировать все данные в сортированном порядке – они инициируют симметричный обход[80] классического B-дерева, который скачет вверх-вниз по различным уровням дерева. Если дерево размещено на диске поуровнево, то переключение между уровнями вызывает неизбежный случайный доступ. Диапазонные запросы и потребность в выводе всех данных за одно быстрое сканирование важны в таких системах, как базы данных и файловые системы. Следовательно, способность обеспечивать быстрые обходы B^+-дерева становится как нельзя кстати.

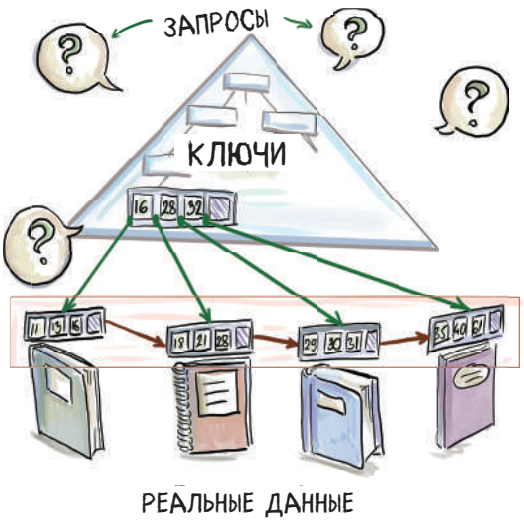


Рисунок 10.8 Организация B^+-дерева. Внутренние узлы просты и содержат ключи, которые маршрутизируют запросы к листьям, в которых содержится более подробная информация о каждом ключе

[80] Англ. in-order traversal. – *Прим. перев.*

Вдобавок отсутствие указателей на фактические данные во внутренних узлах освобождает много пространства для хранения большего числа ключей. Это дает более высокий коэффициент ветвления и меньшую глубину, что приводит к меньшему числу операций ввода-вывода для обычных операций, чем в классическом *B*-дереве.

10.3.6 Чем отличаются операции на B^+-дереве

Первоначально B^+-дерево может быть построено таким образом, чтобы ключи внутренних узлов были дублированными версиями реальных ключей данных. Другими словами, вначале все элементы, существующие в листьях, существуют и во внутренних узлах. Однако когда элемент удаляется, он удаляется только из листа и остается во внутреннем узле в качестве ориентира (если только больше нет необходимости в этом ключе-разделителе из-за слияния узлов и т. д.). Например, если элемент 28 должен был быть удален из B^+-дерева на рис. 10.8, то он был бы удален из листового уровня, но остался бы разделителем между двумя листами во внутреннем узле.

Аналогично этому, во время более сложной вставки узел расщепляется на два, и ключ из листа повышается до внутреннего уровня; в B^+-дереве он дублируется, в результате чего он по-прежнему остается на уровне листа и повышается до уровня внутреннего узла, чтобы служить разделителем.

Поиск всегда ведется вплоть до уровня листьев дерева, поэтому не может быть случая, когда при поиске выполняется ноль операций ввода-вывода, поскольку элемент был найден на одном из более высоких уровней дерева, кешированных в оперативной памяти. С другой стороны, после того как элемент найден, операция следования[81] выполняется за амортизированное $O(1/B)$ время, потому что для каждой $\theta(B)$ операции мы доставляем новый блок, а во все остальные моменты времени операция следования свободна. Как мы увидим позже в этой главе, B^+-дерево является полезным компонентом для построения более крупных структур данных, в которых слияние компонентов происходит эпизодически. В этом случае возможность быстро просканировать все данные двух компонентов и объединить их способом, подобным сортировке слиянием, значительно повышает производительность.

10.3.7 Вариант использования: *B*-деревья в MySQL (и многих других местах)

B-деревья формируют основу для многих движков баз данных, в частности PostgreSQL, MySQL и многих других. Файловые системы, такие как файловая система Apple HFS+ (**http://mng.bz/YgoN**) и BTRFS от Linux (**http://mng.bz/GGYq**), используют *B*-деревья. Если ваша компания использует какую-либо базу данных, то, скорее всего, это база данных основана на *B*-деревьях. Многие из указанных реализаций на самом деле являются B^+-деревьями.

[81] Англ. successor operation. – *Прим. перев.*

В качестве примера (и для разнообразия) рассмотрим онлайновое приложение, не имеющее дела с огромным набором данных. Веб-сайт содержит информацию обо всех складах индивидуального хранения вещей в Соединенных Штатах. База данных содержит около 50 000 складов индивидуального хранения вещей, но каждый склад имеет большое число типов хранилищ, которые можно арендовать (различные категории размеров хранилища; различные функциональные возможности, такие как климат-контроль в помещении, доступ к лифту или акции по ценам), таким образом, по сути, у нас есть около 10 млн индивидуальных записей, включая исторические записи.

Пользователи могут зайти на веб-сайт, чтобы проверить доступность определенных типов складских помещений в их районе, и фильтровать по различным критериям (например, по почтовому индексу, размеру хранилища и т. д.). Каждый день подписывается более 100 000 новых договоров аренды, поэтому можно допустить, что на веб-сайте размещается еще большее число запросов.

С другой стороны, изменения в базе данных тоже происходят, но не так быстро, как запросы; например, старый склад может закрываться и/или открываться новый; также может изменяться информация о ценах, но все это происходит с частотой в пару раз в неделю. Для ускорения поиска база данных должна храниться в виде *B*-дерева, и мы можем строить индексы на разных столбцах таблицы (например, почтовом индексе).

В следующем далее разделе мы коснемся математических основ оптимальности поиска в *B*-дереве. Этот раздел в первую очередь предназначен для читателей, интересующихся математикой, и в противном случае его можно пропустить.

10.4 Немного математики: почему поиск в *B*-дереве оптимален во внешней памяти?

Приступая к определению оптимального способа выполнения запросов во внешней памяти, давайте вернемся к оперативной памяти и оптимальному поиску в оперативной памяти. Мы знаем, что деревья двоичного поиска (а также двоичный поиск в сортированном массиве) могут оптимально выполнять запросы за $\sim \log_2 N$ сравнений; другими словами, нижняя граница поиска в оперативной памяти равна $\Omega(\log_2 N)$ сравнениям. Но откуда мы это знаем? Другими словами, откуда мы знаем, что в один прекрасный день кто-нибудь не придет и не изобретет новый алгоритм, который будет быстрее двоичного поиска?

Отвечая на этот вопрос, нужно произвести нижнеграничный аргумент, который помещает все потенциальные алгоритмы под один зонтик процедур, выполняющих последовательность сравнений (например, $a < 3$?), ответы на которые могут быть утвердительными либо отрицательными, и проанализировать информацию, которую мы узнаем из каждого ответа.

То есть мы работаем в мире алгоритмов, которые могут выполнять только сравнения (в противном случае хеш-таблицы превзойдут нижнюю границу поиска). Затем мы вычисляем минимальное число вопросов, которые эта процедура общего определения должна поставить, чтобы решить задачу.

В целях иллюстрации этого момента давайте обратимся к детской игре, которая, возможно, вам знакома: допустим, вы загадываете число x в диапазоне от 1 до 1 000 000, а ваш друг/подруга пытается угадать это число. Ему разрешается задавать такие вопросы, как «x меньше, больше или равно 30 000?», и вы должны дать ему правдивый ответ. Если x равно упомянутому им числу, то игра прекращается; в противном случае вы отвечаете, что его догадка слишком велика либо слишком мала, и игра продолжается до тех пор, пока он не угадает правильное число. Цель состоит в том, чтобы он угадал правильное число за наименьшее возможное число вопросов.

Вы можете заключить, что самым лучшим первым вопросом будет вопрос «x меньше, больше либо равен 500 000?». Благодаря такому подходу даже в наихудшем случае число потенциальных вариантов сокращается с 1 000 000 до 500 000. Если ваш друг выберет меньшее либо большее число, то это будет полезно для вас, но не для друга, так как вы сможете подстроить свои ответы под те варианты, которые оставят большее число кандидатов, оставаясь при этом последовательными в своих ответах (например, если в качестве первого вопроса он спрашивает, не является ли число меньше, равно либо больше 900 000, то вы обязательно ответите «меньше»).

Вывод таков: один вопрос/сравнение помогает сокращать число вариантов максимум в два раза; вопрос может сокращать варианты в меньшее число раз или вообще их не сокращать, если он плохо поставлен, но самое большее он поможет сокращать варианты в два раза. Это означает, что для перехода из поискового пространства N к 1 придется задать как минимум $\Omega(\log_2 N)$ вопросов, поэтому при $N = 1\,000\,000$ наша игра называется «20 вопросов». Теперь давайте перенесем эту аналогию во внешнюю память.

Если в оперативной памяти мы подсчитываем *вопросы* (то есть число сравнений, которые алгоритм должен выполнить для решения задачи), то в модели внешней памяти мы подсчитываем *операции* ввода-вывода. Следовательно, нам нужно вычислить максимально возможную выгоду (то есть сколько информации мы узнаем из одного блочного ввода в память).

Поскольку передача в память содержит не более B элементов, ввод одного блока – это как бы изменение игры, позволяющее нашему другу задавать немного более сложный вопрос, содержащий значения B. Примером такого вопроса с $B = 4$ может быть «Куда бы вы поместили x между следующими четырьмя числами: 23, 31, 56 и 88?» Если x равно одному из чисел, то игра прекращается; в противном случае у вас будет пять вариантов ответа ($x < 23$, $23 < x < 31$, $31 < x < 56$, $56 < x < 88$ и $x > 88$), и игра будет продолжаться до тех пор, пока одно из предложенных чисел не станет равным x. Каким будет оптимальный первый вопрос нашего друга, если, скажем, $B = 4$? Это будут четыре равноотстоящих числа, так что какой бы из пяти вариантов

мы ни выбрали, расстояние между ними будет равным. Это предохраняет нас от затягивания игры.

В теоретической информатике данный метод доказательства называется *аргументацией соперника*[82]. В нашей игре мы являемся соперниками, потому что при наличии элементов *B* мы всегда будем размещать *x* в подпространстве, которое позволяет игре продолжаться дольше всего. Именно так мы и тестируем наихудший вариант алгоритма. Единственный способ одержать решительную победу над соперником – это чтобы алгоритм делал все подпространства одинакового размера. Когда мы внедряем алгоритмы в реальный мир, у нас нет реальных соперников; правильнее сказать, метафора соперника существует для того, чтобы помогать нам осознавать асимптотическую сложность задачи.

Таким образом, лучшее, что может произойти, – это если *B* элементов в блоке помогут сократить число вариантов в *B* + 1 раз. Опять же, обратите внимание, что наш друг может составить плохой блок, что позволило бы нам, сопернику, сократить пространство на величину, меньшую, чем *B* + 1 (см. рис. 10.9, на котором показано, как можно составлять хороший/плохой блок).

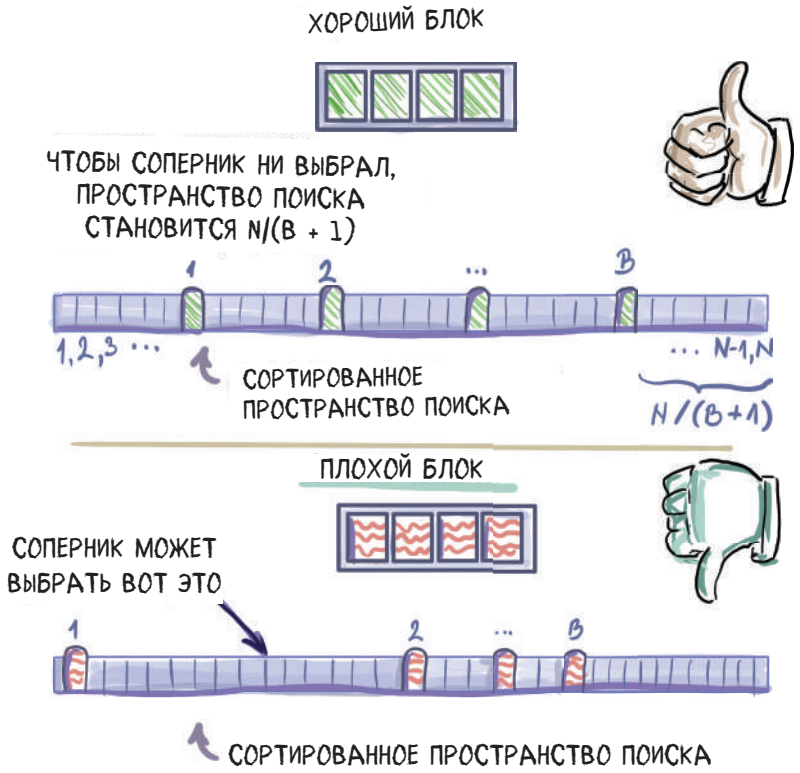


Рисунок 10.9 Расчет нижней границы предполагает наличие хороших блоков

[82] Англ. adversary argument. – *Прим. перев.*

Поскольку каждая операция ввода-вывода помогает сокращать общее число вариантов не более чем в $B + 1$ раз, то для выполнения поиска во внешней памяти нам требуется $\Omega(\log_{B-1} N) = \Omega(\log_B N)$ операций ввода-вывода, и поиск в B-дереве удовлетворяет этой нижней границе.

10.4.1 Почему вставки/удаления в *B*-дереве не являются оптимальными во внешней памяти

Теперь, когда мы знаем, что B-деревья оптимальны по отношению к запросам, давайте рассмотрим операции модификации, такие как вставка и удаление, которые требуют того же асимптотического числа перемещений в память, что и поиск.

Однако операции вставки и удаления существенно отличаются от операций поиска, поскольку вставка не требует немедленного подтверждения того, что новый элемент был сохранен в листе. Аналогично этому, удаление не требует немедленного подтверждения того, что элемент был физически удален из дерева. Единственное подтверждение приходит в результате более поздней операции поиска, когда она должна привести к удовлетворительному ответу для вставленного элемента и отрицательному для удаленного элемента. Поиск – это единственная операция, требующая немедленной обратной связи, и как таковую ее нельзя задерживать. С другой стороны, вставки и удаления можно задерживать и буферизовать. Благодаря такому подходу структура данных может обрабатывать эти операции более эффективно в пакетном режиме. В остальной части главы мы увидим две такие структуры данных: B^ε-деревья и LSM-деревья.

10.5 *B*$^\varepsilon$-деревья

B^ε-дерево было разработано Бродалом (Brodal) и Фагербергом (Fagerberg) [6] как структура данных, которая воплощает компромисс между скоростью операций вставки и поиска во внешней памяти. Компромисс отражается в диапазоне значений параметра $\varepsilon = [0,1]$, который можно настраивать, и при $\varepsilon = 0$ структура данных полностью оптимизирована под операции вставки/удаления; при $\varepsilon = 1$ она полностью оптимизирована под операции поиска (это B-дерево).

Однако когда в этой главе мы будем говорить о B^ε-деревьях, мы обычно будем ссылаться на «половинчатую» структуру данных, которая возникает при $\varepsilon = 1/2$. Указанное значение ε интересно тем, что в данной точке спектра мы получаем структуру данных с операциями поиска, которые хуже только на постоянный коэффициент, чем у B-деревьев, и операциями вставки, которые асимптотически лучше, чем у B-деревьев. Это означает, что B^ε-дерево значительно лучше подходит для рабочих нагрузок, оптимизированных под операции записи, чем B-дерево, и поддерживает асимптотически оптимальный поиск.

10.5.1 B^{ε}-дерево: принцип работы

Ключевой конструкционной особенностью B^{ε}-деревьев является то, что, помимо ключей, каждый внутренний узел имеет буфер. Буферы призваны временно хранить вставки и удаления, которые действуют как сообщения на пути к назначенному листу. Операция удаления в B^{ε}-дереве работает не так, как в B-дереве, путем прямого перехода к местоположению элемента и физического его удаления. Вместо этого в буфер корневого узла первоначально вставляется сигнальное сообщение[83] «Удалить *x*», и оно постепенно перемещается по буферам вниз по пути от корня к листу, который хранит *x*. Как только сигнальное сообщение достигает листа, содержащего *x*, *x* физически удаляется из дерева вместе с сигнальным сообщением. Аналогичный процесс существует и со вставками.

Как и в B^{+}-дереве, все элементы B^{ε}-дерева находятся в листьях, поэтому все вставки и удаления в конечном итоге влияют на листья, а ключи во внутренних узлах используются только в качестве точек разворота для направления поиска. Сообщения о вставке/удалении ожидают в буфере до тех пор, пока не будет собрано достаточное число других сообщений, которые можно будет отправить одному из дочерних элементов всего за одну операцию ввода-вывода. Это отличается от B-деревьев, в которых для одной вставки/удаления используется одна операция ввода-вывода, чтобы перейти на следующий уровень дерева, и, следовательно, несколько операций ввода-вывода, чтобы завершить работу. Задерживая операции в B^{ε}-дереве, мы можем ускорить их выполнение в амортизированном смысле. Позже мы увидим, как поддержание всех этих сообщений влияет на алгоритм поиска.

Внутренняя структура узла B^{ε}-дерева выглядит следующим образом: каждый узел содержит ключи, а оставшееся $B - B^{\varepsilon}$ пространство используется для буфера (см. рис. 10.10 с узлом, где $B = 16$ и $\varepsilon = 1/2$). В рамках нашей общей конфигурации $\varepsilon = 1/2$ мы имеем $\sqrt{B}$ ключей и $B - \sqrt{B}$ буферное пространство. Следовательно, буфер занимает бóльшую часть пространства узла. Также обратите внимание, что глубина дерева определяется структурой узла, где $\sqrt{B}$ ключей в расчете на узел дают глубину дерева, равную $\log_{\sqrt{B}} N = 2\log_B N$. Несмотря на то что число ключей значительно меньше, глубина дерева всего в два раза больше, чем в B-дереве. Это влияет на производительность поиска лишь отчасти, поскольку мы пожертвовали пространством узла, чтобы разместить буферы, но асимптотическая стоимость поиска остается равной стоимости B-дерева.

10.5.2 Механика буферизации

Буфер можно трактовать как отдельную область узла, в которой накапливаются сообщения, и как только буфер становится переполненным, мы его очищаем. То есть мы очищаем только те элементы, которые предназначены для дочернего узла, у которого больше всего ожидающих обновлений.

[83] Англ. tombstone message; син. надгробное сообщение. – *Прим. перев.*

Возможно, более чистым способом визуального представления буфера является его разбивка на $B^\varepsilon + 1$ разных подбуферов, где каждый подбуфер содержит сообщения, предназначенные для одного конкретного дочернего элемента, на основе значений ключей (например, как на рис. 10.10). Мы не подразделяем пространство между подбуферами в явной форме, и разные подбуферы могут делиться своим пространством с другими, так что очистка инициируется только после заполнения всего буфера. Однако как только буфер заполняется, очищается только самый полный подбуфер. Буфер обычно реализуется в виде сбалансированного дерева двоичного поиска, в котором можно быстро добавлять элементы и по ним перемещаться, держа их в сортированном порядке. Если подбуферы реализуются в виде отдельных деревьев двоичного поиска, то следует беспокоиться только об их общем размере, чтобы не превышать емкость буфера, а не о размерах отдельных подбуферов.

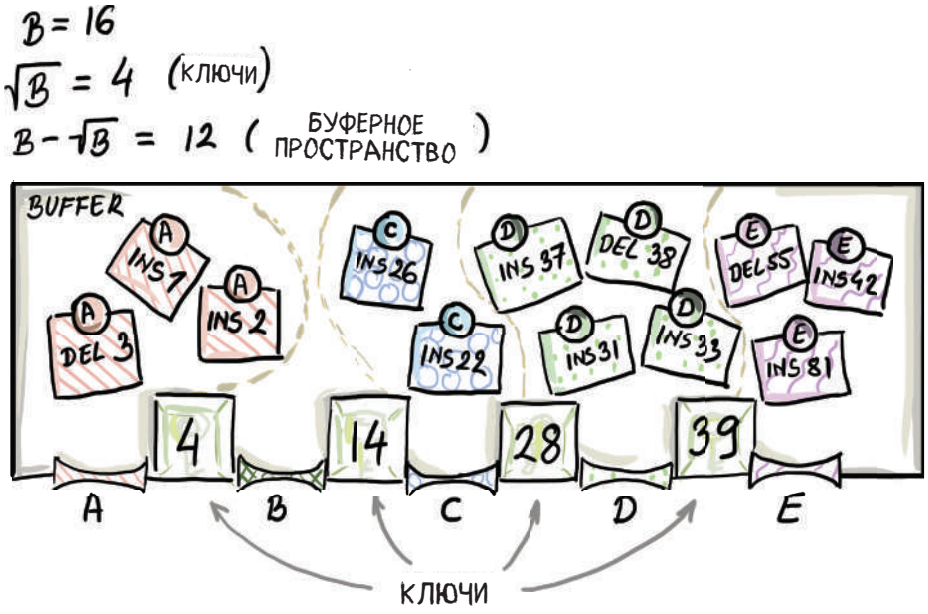


Рисунок 10.10 Узел B$^\varepsilon$-дерева имеет ключи и буферы. В настоящее время буфер полон и больше не может вмещать обновления

На рис. 10.11 показан момент, когда буфер переполняется после нового обновления (`Del 8`) и очищается. Сообщения из самого полного подбуфера очищаются и распределяются по соответствующим подбуферам дочернего элемента. Другие сообщения из буфера остаются в буфере (например, сообщение `Del 8`, которое инициировало очищение, не очищается).

На рис. 10.11 в буфере дочернего узла уже имелось несколько предыдущих ожидающих обновлений (`Del 29` и `Ins 36`), но вместе с поступающими сообщениями узел не превысил емкость буфера, поэтому процесс на этом

останавливается. Тут не показана одна важная деталь: с каждым сообщением об обновлении связана временна́я метка. Временная метка помогает восстанавливать историю и, значит, правильно выполнять алгоритм поиска.

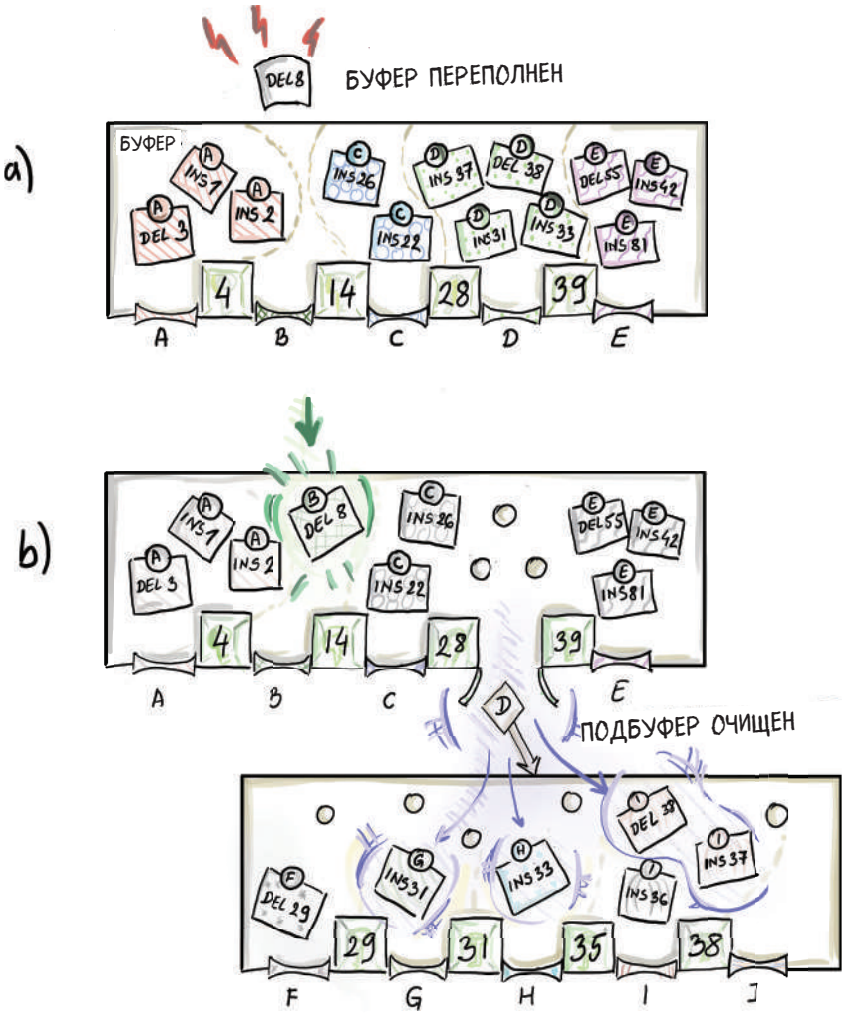


Рисунок 10.11 Когда буфер становится переполненным, мы очищаем самый полный подбуфер соответствующего узла

10.5.3 Вставка и удаление

Теперь, когда у нас есть механизм буферизации, давайте рассмотрим весь процесс вставки/удаления до конца. Сообщение о вставке/удалении первоначально помещается в буфер корня дерева. Если сообщение не инициирует переполнения буфера, то дело сделано. В противном случае,

если сообщение инициирует очистку корневого буфера, мы его очищаем и выполняем любые каскадные очистки вниз по дереву, потенциально вплоть до листового уровня, с соответствующими вставками/удалениями в листе.

Если во время очисток достигается листовой уровень, то мы выполняем вставку/удаление сообщений, которые поступили на листовой уровень, вставляя/добавляя элемент физически, как это делается в B^{+}-дереве, и удаляя эти сообщения вставки/удаления из их буфера. Узлы B^{ε}-дерева обладают тем же свойством «порядка d», что и B-деревья. Когда лист превышает свою емкость, он расщепляется точно так же, как это делается в B^{+}-деревьях. Когда он становится слишком пустым, он сливается таким же образом, как и в B^{+}-деревьях. Таким образом, типичная вставка/удаление, которая продвигается вниз по буферам и в конечном итоге достигает листа, может затем инициировать операцию расщепления/слияния в листе, которая, в свою очередь, может инициировать новые расщепления/слияния вверх по дереву. Весь этот процесс работает точно так же, как было описано для B-дерева, за исключением того, что теперь мы расщепляем/объединяем ключи и перераспределяем сообщения в буферах.

10.5.4 Поиск

Поиск в B^{ε}-дереве выполняется аналогично поиску в B-дереве, поскольку он следует по пути от корня к листу, который может содержать запрашиваемый элемент. Однако поиск в дереве также должен учитывать сообщения о вставке/удалении, с которыми он сталкивается на своем пути, поскольку они влияют на окончательный результат поиска.

Например, допустим, мы ищем элемент 10, который был вставлен в прошлом; однако недавно для него была выполнена операция удаления. Если с тех пор в отношении 10 не было выполнено никаких других операций, то поиск должен сообщить, что элемент отсутствует. Однако этот элемент все еще может существовать в листе дерева, поскольку сигнальное сообщение, возможно, до него не дошло.

По этой причине операция поиска должна собирать все сообщения (с их временны́ми метками), которые относятся к запрашиваемому элементу на пути от корня к листовому элементу. Затем, когда он определяет, что элемент в листе присутствует, он применяет любые потенциальные сообщения о вставке/удалении в правильном хронологическом порядке. На рис. 10.12 поиск элемента 7 собирает сообщения на своем пути от корня к листовому элементу, и после достижения листового уровня и применения всех сообщений он приходит к выводу, что 7 присутствует.

Следует учитывать, что поиск никогда не приводит к очистке каких-либо буферов. Внутренне он собирает соответствующие сообщения, чтобы правильно ответить на запрос. Вся работа, связанная с очисткой буферов, возлагается на операции вставки/удаления.

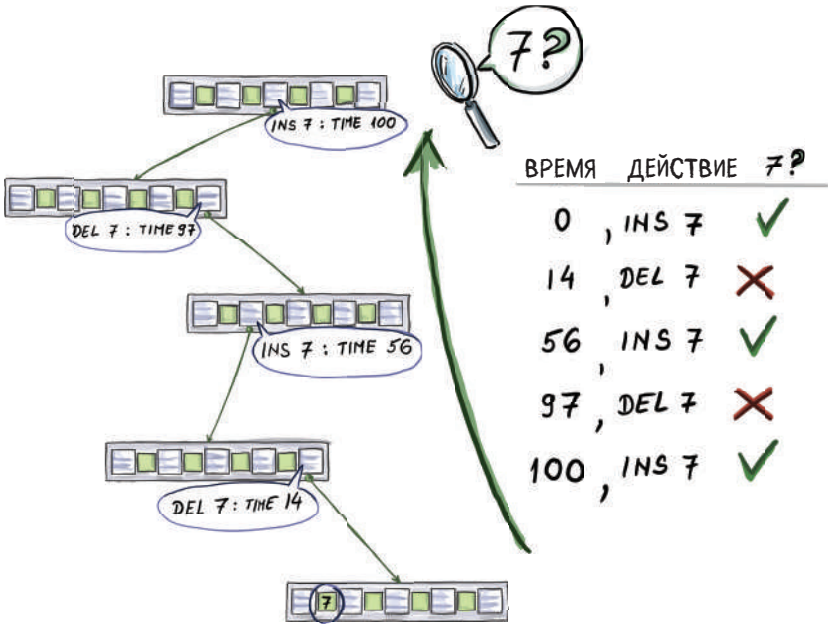


Рисунок 10.12 Вставка и удаление сообщений по пути от корня к листовому элементу 7

10.5.5 Анализ стоимости

В этом разделе мы проанализируем стоимость операций поиска, вставки и удаления в B^ε-дереве. Мы сосредоточимся на анализе интересующей нас половинчатой структуры данных ($\varepsilon = 1/2$), хотя его легко обобщить на любое значение ε.

B^ε-дерево имеет $O(\log_B N)$ уровней, поэтому поиск должен прочитывать $O(\log_B N)$ узлов на своем пути от корня к листу. В этом смысле поиск асимптотически стоит столько же, сколько поиск в B-дереве. Точнее, в два раза медленнее, потому что B^ε-дерево в два раза глубже.

Вставки и удаления можно анализировать вместе, поскольку они работают схожим образом. Сперва нам нужно проанализировать стоимость спуска одного сообщения с одного уровня дерева на следующий. Это зависит от числа элементов, которые перемещаются вместе за одну операцию ввода-вывода при очистке буфера. Когда буфер становится полным, самый полный подбуфер заполнен по меньшей мере так же, как и все остальные подбуферы; следовательно, он содержит по меньшей мере $(B - \sqrt{B}) / (\sqrt{B} + 1) \sim \sqrt{B}$ сообщений. Это означает, что за 1 операцию ввода-вывода мы транспортируем приблизительно $\sqrt{B}$ обновлений на следующий уровень дерева; следовательно, каждое обновление стоит $O(1/\sqrt{B})$ в расчете на уровень. Дерево имеет $O(\log_B N)$ уровней, поэтому одна вставка/удаление обходится в целом в $O(\log_B N/\sqrt{B})$ операций ввода-вывода, что в $\sqrt{B}$ раз

дешевле, чем в *B*-дереве! Пример тому показан на рис. 10.11, где $B = 16$, но мы очистили четыре элемента (самый полный подбуфер содержит четыре элемента), поэтому на каждый элемент мы использовали ¼ операции ввода-вывода. Если учесть, что *B* зачастую выражается в тысячах или даже миллионах, то удешевление в $\sqrt{B}$ раз может означать значительное снижение стоимости.

Помимо стоимости буферизации, существует также классическая стоимость физического расщепления и слияния узлов, которые подобны узлам в *B*-дереве. Но на этот раз нам нужно проанализировать эту стоимость более тщательно, учитывая, что мы не хотим превышать $O(\log_B N / \sqrt{B})$ стоимость вставки/удаления. Другими словами, с *B*-деревом можно быть более свободным в анализе и допускать, что в наихудшем случае на каждом уровне происходит одно расщепление либо слияние, и эта стоимость все равно будет покрыта уже существующей стоимостью вставки/поиск, связанной просто с перемещением вниз по дереву. С B$^{\varepsilon}$-деревом приходится быть более бережливыми. К счастью, число ожидаемых расщеплений и слияний работает в нашу пользу.

Возьмем наихудшую возможную рабочую нагрузку всех вставок, направленных к одному листу; эта рабочая нагрузка максимизирует число расщеплений узлов. Начнем с дерева порядка $d = \theta(\sqrt{B})$, узлы которого минимально заполнены, причем каждый $\theta(\sqrt{B})$ вставляет, и нам приходится расщеплять узлы. После $\theta(\sqrt{B})$ таких расщеплений, которые также являются вставками на более высокий уровень, нам нужно сделать расщепление одним уровнем выше. То есть после $\theta((\sqrt{B})^2)$ вставок/удалений мы будем затрагивать только уровень, расположенный выше листового уровня. Однако эта стоимость уже покрыта стоимостью гораздо более часто встречающихся расщеплений на листовом уровне. Можно было бы продолжить аргументацию в пользу более высоких уровней, но в итоге можно сказать, что стоимость, влекомая расщеплением и слиянием, пренебрежимо мала и амортизируется постоянной по стоимости операцией ввода-вывода. Это означает, что над стоимостью расщеплений/слияний в B$^{\varepsilon}$-дереве доминирует стоимость очистки и транспортировки сообщений вниз по дереву.

10.5.6 B$^{\varepsilon}$-дерево: спектр структур данных

Как упоминалось ранее, в зависимости от выбранного ε можно получать более высокую производительность поиска либо более высокую производительность вставки, чем в обычной конфигурации при $\varepsilon = 1/2$. На рис. 10.13 показаны три точки в спектре: (a) *B*-дерево (все ключи, без буферов), (b) B$^{\varepsilon}$-дерево при $\varepsilon = 1/2$ (несколько ключей и большинство буферного пространства) и (c) буферизованное репозиторное дерево[84] (один ключ и все буферное пространство).

Буферное репозиторное дерево – это интересная структура данных, которая позволяет оптимизировать производительность вставки/удаления

[84] Англ. repository tree. – *Прим. перев.*

даже больше, чем B^ε-дерево с $\varepsilon = 1/2$. Поскольку буфер большой и сообщения в каждом узле могут направляться только в двух разных направлениях, на следующий уровень могут совместно перемещаться $\theta(B)$ элементов, снижая производительность вставки до $O(1/B)$ в расчете на уровень и $O(\log_2 N)/B)$ операций ввода-вывода в целом (буферно-репозиторное дерево имеет $O(\log_B N \log_2 N)$ уровней, точно так же, как в дереве двоичного поиска, что делает его неприемлемым для производительного поиска).

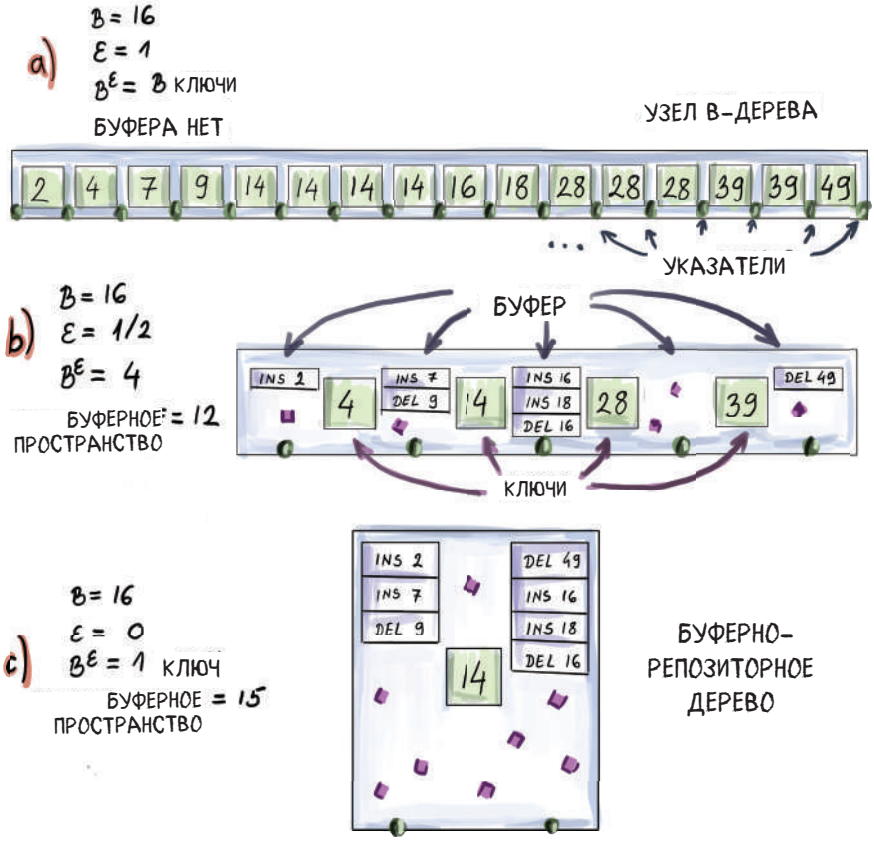


Рисунок 10.13 Спектр B^ε-древовидных структур данных, от наиболее оптимизированных под чтение до наиболее оптимизированных под запись

10.5.7 Вариант использования: B^ε-деревья в TokuDB

B^ε-деревья были реализованы в движке хранения Percona TokuDB для сервера Percona под MySQL. В том же ключе были реализованы файловые системы, такие как BetrsFS [7], которые внутренне выполняют B^ε-деревья. Поскольку B^ε-деревья помогают улучшать качество вставок, они могут способствовать упрощению и ускорению поддержания индексов, позволяя таким образом нескольким индексам сосуществовать, не давая вставкам

становиться слишком медленными. То есть, по иронии судьбы, вся история сводится к тому, что мы ухудшаем поиск, чтобы помогать вставке, которая, в свою очередь, помогает поиску.

Типичный вариант использования, когда B^{ε}-деревья могут оказаться полезными, – это высокодинамичные приложения, в которых как вставка, так и поиск должны быть быстрыми. Рассмотрим следующее высокопроизводительное приложение: ваша компания размещает веб-запросы для крупнейшего издателя онлайновых журналов. Пользователи постоянно загружают новый контент и реагируют на него (например, добавляя новые комментарии), и в то же время большой объем нового контента и статей публикуется, модифицируется и одновременно запрашивается.

Самое трудное в приложениях такого типа – публиковать новый актуальный контент не за счет снижения удобства чтения потребителями. Аналогичные варианты использования возникают и в социальных сетях, в которых новый контент должен усваиваться с высокой скоростью; однако контент также должен быстро доставляться пользователям.

10.5.8 Торопитесь медленно, как операции ввода-вывода

Одно из главнейших различий между B-деревьями и B^{ε}-деревьями заключается в том, что B-деревья выполняют обновления прямо на месте; то есть, например, при поступлении операции модификации/вставки/удаления изменение немедленно вносится прямо там, где находится соответствующий элемент. С другой стороны, B^{ε}-деревья выполняют обновления не прямо на месте, то есть сообщение о модификации/вставке/удалении вносится в структуру данных в другом месте, а не там, где находится соответствующий элемент. В них нет никакой спешки с немедленным отысканием элемента и применением к нему необходимых изменений. Обратите внимание, что обновления не на месте увеличивают объем пространства, требуемый структурой данных, поскольку число элементов в структуре данных измеряется не числом несовпадающих элементов, а числом обновлений в ней. Мы храним элементы и сообщения, относящиеся к этим элементам.

Вместе с тем обратите внимание, что именно функциональность «не прямо на месте» помогает операциям модификации/вставки/удаления выполняться быстрее, чем в B-дереве. Для того чтобы выполнить обновление, нам не нужно сразу же искать точное местоположение элемента и сжигать при этом большое число операций ввода-вывода. Обновления не торопятся путешествовать вниз по дереву, когда спускаться дешевле всего. Вставки/удаления ожидают применения дольше, но по этой причине выполняются быстрее; это объясняется тем, что мы измеряем эффективность операций не временем, затраченным на применение операции, а числом операций ввода-вывода, необходимых для применения изменения. В следующем далее разделе мы увидим структуру данных, которая расширяет

понятие быстрых операций записи (и операций записи не прямо на месте) еще дальше, чем B^{ε}-деревья.

10.6 Журнально-структурированные деревья слияния (LSM-деревья)

Для того чтобы понять, как появились LSM-деревья[85], давайте начнем с самостоятельной разработки простой структуры данных, оптимизированной под операции записи. Каков оптимальный способ реализации индекса во внешней памяти с невероятно быстрой вставкой/удалением не прямо на месте, без учета скорости поиска?

На ум приходит простая регистрация сообщений о вставках/удалениях в одном последовательном журнале. Одним из вариантов такой структуры данных является резидентный буфер, в котором накапливаются сообщения о вставках, удалениях или модификациях записей. Как только буфер заполняется, мы последовательно сбрасываем его на диск. Затем мы снова заполняем резидентный буфер операциями записи (говоря «операции записи», мы подразумеваем операции вставки, удаления и модификации) и сбрасываем содержимое памяти, добавляя новые данные в конец журнала.

Описанная выше простая система гарантирует идеальную производительность вставки/удаления, равную $1/B$ на элемент. Это амортизированная стоимость записи элементов на диск, поэтому нетрудно понять, почему невозможно добиться лучшего. Разумеется, запросы будут ужасны, так как нам придется сканировать весь дисковый файл, чтобы ответить на запрос (N/B операций ввода-вывода).

Теперь попробуем немного модифицировать эту идею без ущерба для производительности записи. Допустим, что при каждом заполнении резидентного буфера все элементы в резидентном буфере внутренне сортируются. Для этого буфер может представлять собой своего рода сбалансированное двоичное дерево и сбрасывать отсортированный диапазон в отдельный файл или таблицу на диске. При следующем заполнении буфера мы снова сортируем все резидентные данные и сбрасываем их в другую таблицу рядом с первой таблицей и т. д. Теперь мы имеем чуть более организованную систему, содержащую множество отдельных таблиц с данными, причем каждая таблица внутренне упорядочена. Операции вставки/удаления по-прежнему выполняются оптимально за $O(1/B)$, амортизируемых в расчете на элемент, поскольку сортировка буфера происходит во внутренней памяти и не требует никаких дополнительных операций ввода-вывода. Запросы не стали астрономически лучше: теперь нам нужно обследовать каждую таблицу, чтобы локализовывать обновления, связанные с интересующим нас элементом. Если размер таблицы аналогичен размеру основной памяти M, а общее число обновлений равно N, то всего у нас будет N/M таблиц. Поскольку каждая таблица отсортирована, мы можем использовать

[85] Англ. log-structured merge-tree (LSM-tree). – *Прим. перев.*

двоичный поиск, чтобы управлять поиском внутри отдельных таблиц, что помогает избегать полного линейного сканирования таблицы. В результате мы получаем $O(N/M \times \log_2(M/B))$ операций ввода-вывода в стоимости в расчете на запрос.

Давайте немного усовершенствуем нашу простую конструкцию: учитывая, что таблицы немутируемы (мы не будем изменять их после сброса на диск), почему бы не построить индекс с использованием B^+-дерева поверх каждой таблицы и не повысить производительность запросов до $O(N/M \times \log_B(M/B))$ операций ввода-вывода? Наша результирующая структура данных показана на рис. 10.14.

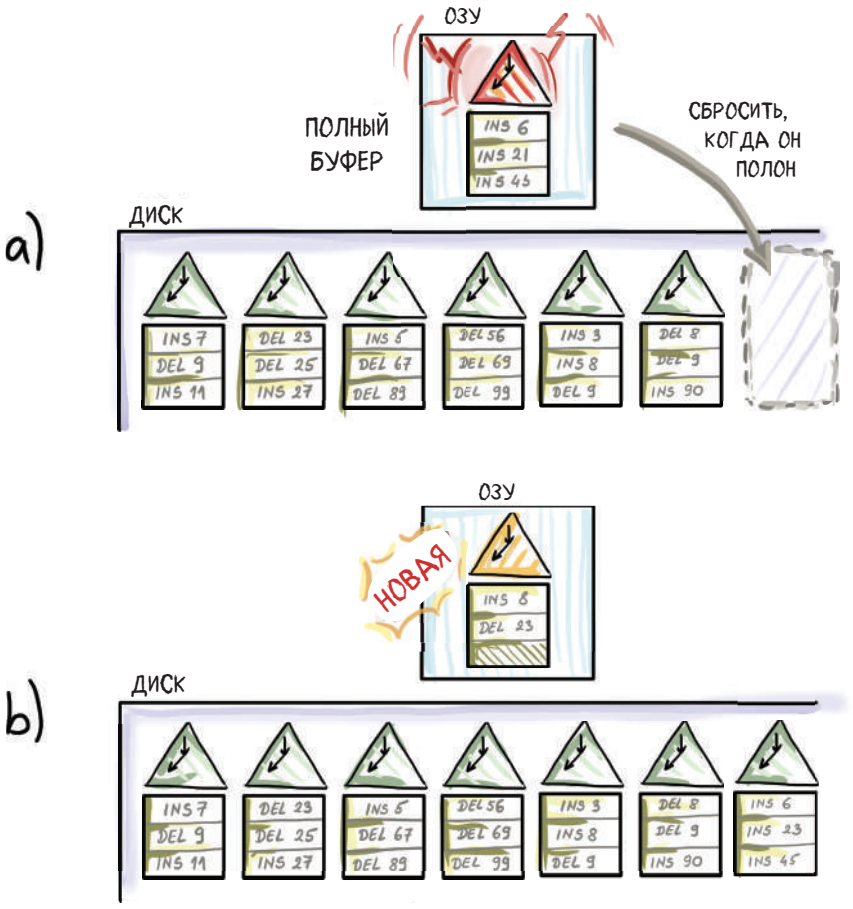


Рисунок 10.14 Простая структура данных, оптимизированная под операции записи, но не являющаяся LMS-деревом

Со временем растущее число таблиц усугубит и без того плохую производительность поиска. Поддержание нескольких фильтров Блума в оперативной памяти (пример использования описан в главе 3), по одному на

каждую таблицу, позволяет отказаться от поиска на диске в таблицах, не содержащих обновлений для запрашиваемого элемента. Однако, как вы помните, фильтры Блума тоже растут пропорционально общему объему данных, так что не успеем мы оглянуться, как заполним оперативную память тонной мини-фильтров Блума. Фильтры Блума позволяют выиграть время, но ненадолго, если мы имеем дело с невероятно высокой частотой вставок.

Разработанная в 1996 году структура данных, именуемая журнально-структурированным деревом слияния (LSM-дерева), воплощает идею нашей упрощенной структуры, оптимизированной под операции записи, и добавляет к ней механизм, который ограничивает число таблиц на диске путем их эпизодического слияния и сжатия. LSM-дерево было успешно реализовано в ряде баз данных, оптимизированных под операции записи, таких как LevelDB, используемой Google, RocksDB, поддерживаемой Facebook, и др. Давайте посмотрим на принцип работы LSM-деревьев.

10.6.1 LSM-дерево: принцип работы

Существует несколько вариантов базового устройства LSM-дерева, а также множество разных реализаций [8]. Первоначально LSM-дерево составлялось из k компонентов C_0, C_1, ..., C_{k-1}, где C_0 находится во внутренней памяти, а все остальные на диске. Однако есть несколько важных отличий от нашей упрощенной структуры данных: в LSM-дереве мы исходим из того, что размер C_0 примерно равен размеру M памяти, а C_1 на f (обычно $f \geqslant 2$) больше компонента C_0. На самом деле соотношение f поддерживается между размерами любых двух поочередных компонентов, поэтому размеры компонентов в возрастающем порядке равны M, fM, f^2M и т. д.

Компоненты, расположенные на диске, изначально задумывались как B^+-деревья, но, как мы увидим, в современных реализациях используются разные структуры данных, такие как списки пропусков[86] или простые таблицы сортированных пар ключ-значение и файлы. Экспоненциальное увеличение размеров между компонентами гарантирует, что в дальнейшем мы будем иметь управляемое число опрашиваемых компонентов. Самый большой компонент должен хранить все N элементов, поэтому общее число компонентов равно $k = O(\log_f N/M)$, если наименьший компонент имеет размер $\theta(M)$.

Каждый компонент находится на своем уровне, и каждый уровень имеет ограничение на максимальную емкость. При нарушении верхнего порога емкости на одном из уровней соответствующий компонент C_i сливается в компонент C_{i+1}. Это, в свою очередь, может привести к переполнению компонента C_{i+1} и инициировать каскадные слияния на нижестоящих уровнях. В оригинальной конструкции LSM-дерева это достигалось путем слияния диапазона ключей из меньшего компонента в больший. Современные реализации LSM-дерева отдают предпочтение подходу, при котором однажды

[86] Англ. skip list; син. список переходов. – *Прим. перев.*

записанные компоненты (так называемые *отрезки*) не мутируемы. Таким образом, даже если окончательный эффект слияния уровня C_i в C_{i+1} такой же, как и в изначальном подходе слияния LSM-дерева, современная политика слияния между уровнями никогда не подвергает мутированию однажды записанные структуры. Вместо этого создается новый объединенный компонент, а старые высвобождаются. На рис. 10.15 показан пример LSM-дерева и политики слияния, которую мы только что описали, широко известной как *политика поуровневого слияния*[87], устраняющая детали физического слияния данных на диске.

В примере на рис. 10.15 мы устанавливаем $M = 4$ и $f = 2$. В левой части рисунка мы делаем снимок структуры данных в какой-то момент обработки рабочей нагрузки. Компонент C_1 заполнен и должен быть слит в C_2. Для того чтобы слить C_1 в C_2, мы последовательно сканируем диапазон элементов в C_1 и C_2 и выполняем их слияние таким образом, каким они были бы слиты во время сортировки слиянием. Это возможно благодаря тому, что элементы внутри индивидуальных компонентов отсортированы. До слияния в компоненте C_2 было 8 элементов, а теперь их 16 (правая часть рисунка). Компонент C_2 также будет слит в C_3, так как он достиг максимальной емкости.

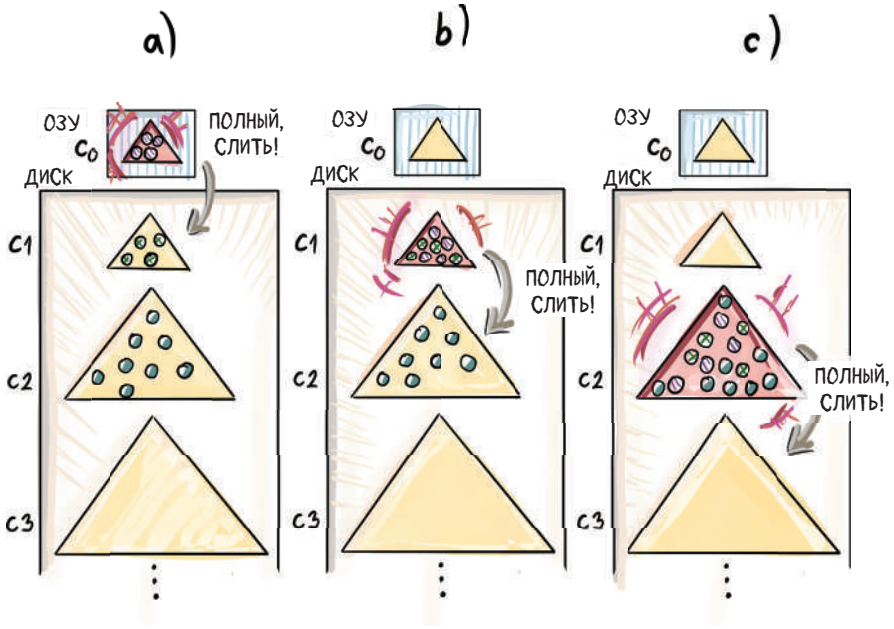


Рисунок 10.15 Слияние меньшего компонента в больший в политике поуровневого слияния LSM-дерева. В примере показаны B^+-древоподобные единичные компоненты на каждом уровне, но в современных реализациях LSM-дерева вместо B^+-деревьев используются различные структуры данных или даже простые таблицы и файлы

[87] Англ. leveling merge policy; см. статью «О стабилизации производительности в системах хранения на основе LSM»: **https://www.vldb.org/pvldb/vol13/p449-luo.pdf**. – *Прим. перев.*

Обратите внимание, что при таком слиянии для спуска одного элемента на следующий уровень мы записываем этот фрагмент данных многократно: один раз, когда он сливается в более низкий уровень, а затем позже, когда другие элементы сливаются в его уровень. Этот процесс повторяется для каждого уровня, и этот так называемый эффект *усиления операции записи* проявляется в большей степени при бóльших коэффициентах роста, так как для заполнения меньшего компонента нам необходимо более f раз сливать его в больший. Термин *усиление операции записи* используется для измерения объема данных, который записывается внутри структуры данных на единицу вставленного элемента. Можно с уверенностью сказать, что в рамках политики поуровневого слияния величина усиления операции записи довольно высока.

Политика поярусного слияния[88] – еще один популярный механизм уплотнения компонентов в современных реализациях LSM-дерева. В данной политике ярусы эквивалентны уровням, за исключением того, что каждый ярус содержит f компонентов одинакового размера. Как только f компонентов на ярусе i заполнены, все они сливаются в один новый компонент на ярусе $i + 1$. Благодаря такому подходу для спуска на новый ярус каждый элемент записывается только один раз. Пример политики поярусного слияния приведен на рис. 10.16, в котором в качестве компонентов используются не B^+-деревья, а сортированные отрезки, и $f = 2$. В данном примере две таблицы на ярусе 1 заполняются и сливаются в одну таблицу на ярусе 2. Процесс слияния происходит быстро и последовательно. На рисунке это не показано, но теперь два полноценных компонента на ярусе 2 будут слиты в один компонент на ярусе 3.

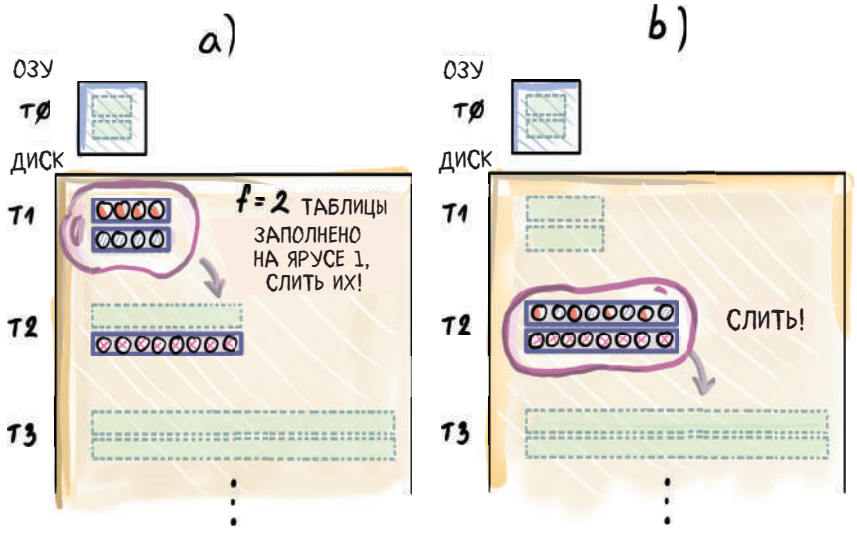


Рисунок 10.16 Политика поярусного слияния в LSM-дереве с $f = 2$. Когда f компонентов на ярусе i заполняются, они сливаются в один компонент на ярусе $i + 1$

[88] Англ. tiering merge policy. – *Прим. перев.*

10.6.2 Анализ стоимости LSM-дерева

Обратите внимание, что при обеих политиках слияния мы теряем часть первоначальной O(1/*B*) производительности записи во время сжатия. Поскольку при политике поуровневого слияния каждый элемент записывается примерно O(*f*) раз, чтобы спуститься на один уровень, общая стоимость спуска одного элемента на один уровень ниже равна O(*f*/*B*), а общая стоимость спуска вниз LSM-дерева – это стоимость, накопленная на всех уровнях: O((*f*/*B*) $\log_f N/M$) операций ввода-вывода. При политике поярусного слияния нам нужно только O(1/*B*) на каждый уровень, и в совокупности по всем уровням требуется O($\log_f N/M$) операций ввода-вывода, что в *f* раз меньше, чем при политике поуровневого слияния.

Однако запросы говорят о другом. Без учета размера самого компонента для проверки наличия элемента требуется примерно постоянное число операций ввода-вывода в расчете на компонент. Этого можно достичь, даже если сам компонент или отрезок намного больше блока. Примером могут служить таблицы сортированных строк (SST), содержащие сортированные пары ключ-значение, а также небольшой индекс ключей. Если мы сначала хотим выяснить, где в таблице может находиться потенциальный элемент, мы доставляем индекс отдельной таблицы (компонент), которая достаточно мала, чтобы уместиться в блоке. Узнав местонахождение элемента, затем нам понадобится еще одна операция ввода-вывода, чтобы доставить элемент. Таким образом, в целом на один компонент приходится одна операция ввода-вывода.

Поскольку в политике поуровневого слияния число компонентов равно числу уровней, для поиска требуется O($\log_f N/M$) операций ввода-вывода. В политике поярусного слияния приходится проверять в *f* раз больше компонентов, что увеличивает стоимость запроса. Однако наибольший прирост производительности запросов достигается за счет использования фильтров Блума, которые помогают перенаправлять запрос в нужную таблицу, тем самым в большинстве случаев доводя производительность запроса до O(1). Эта оптимизация может применяться к обеим политикам слияния, и на этот раз она работает, поскольку общее число компонентов логарифмично по отношению к общему размеру набора данных, и, следовательно, поддержание фильтров Блума в основной памяти вполне управляемо. Диапазонным запросам повезло меньше, так как они не могут так же эффективно использовать фильтры Блума, как точечные запросы.

10.6.3 Вариант использования: LSM-деревья в Cassandra

LSM-деревья были реализованы в ряде крупных движков баз данных, таких как Cassandra, LevelDB, RocksDB и т. д. В частности, LSM-дерево поярусного слияния в Cassandra использует фильтры Блума, чтобы избегать ненужных операций дискового поиска. Рисунок 10.17 является отсылкой к

главе 3, где мы приветствовали применение фильтров Блума в контекстах распределенного хранения. На рисунке показано LSM-дерево, компонентами которого являются таблицы. Наглядно видно, что при $f = 4$ таблицы SST 1-4 относятся к первому ярусу, SST 5-8 – ко второму ярусу и т. д.

Типичным вариантом использования LSM-дерева является приложение, которому требуется невероятно высокая производительность операций записи. Примером такого приложения является продукт резервного копирования, который делает снимки данных через регулярные промежутки времени и хранит петабайты данных, но редко пересматривает их историю. Еще один вариант использования – сетевое приложение для мониторинга трафика, которое отслеживает сотни миллионов запросов в час. Запросы хранятся в базе данных, но случаи явного поиска того или иного запроса бывают редко.

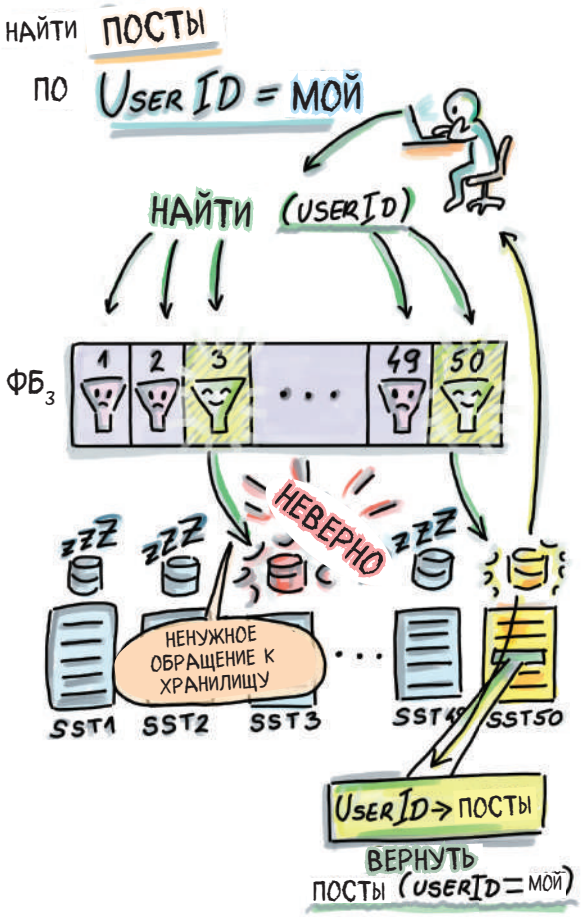


Рисунок 10.17 LSM-деревья используют фильтры Блума для устранения ненужных дисковых запросов к таблицам, которые не содержат запрашиваемых элементов

Резюме

- Индекс базы данных – это структура данных, построенная поверх таблицы базы данных и предназначенная для ускорения производительности запросов к большим таблицам. Индексы строятся с использованием структур данных, которые могут выполнять эффективный поиск во внешней памяти.
- *B*-деревья формируют костяк наиболее распространенных систем хранения данных, таких как MySQL. *B*-дерево – это оптимальная структура данных для выполнения поиска на диске. Узлы *B*-дерева имеют большой размер и обычно связаны с размером блока. Все операции в *B*-дереве логарифмичны с основанием *B*. Когда узлы в *B*-деревьях становятся слишком полными / слишком пустыми, узлы могут расщепляться/сливаться, а дерево – расти вверх.
- *B*-деревья оптимизированы под операцию чтения, и существуют другие структуры данных, которые лучше подходят для записи больших рабочих нагрузок. Операции вставки и удаления, в отличие от операций поиска, могут задерживаться или обрабатываться пакетами вместе. Эта идея используется структурами данных, оптимизированными под операцию записи, для задержки и буферизации операций вставки/удаления, чтобы достигать гораздо более быстрых вставок, чем в *B*-деревьях.
- B^{ε}-дерево – это оптимизированная под операцию записи структура данных, в которой вставки/удаления выполняются асимптотически быстрее, чем в *B*-деревьях, а поиск выполняется с постоянным коэффициентом медленнее, чем в *B*-деревьях. B^{ε}-деревья используют в своих узлах буферы, чтобы временно хранить сообщения о вставке/удалении и в удобный момент обрабатывать их пакетно. Значение параметра ε определяет степень, с которой структура данных отдает предпочтение операции записи перед операцией чтения.
- LSM-дерево – это оптимизированная под операцию записи структура данных, состоящая из сортированных отрезков, которые эпизодически интегрируются в быстром последовательном порядке. LSM-деревья могут обеспечивать чрезвычайно быстрое обновление за счет операций поиска, которые выполняются медленнее, чем в *B*-дереве и B^{ε}-дереве.

Глава 11

Сортировка во внешней памяти

Эта глава охватывает следующие ниже темы:

- понимание важности эффективной сортировки на диске;
- ревизия двух наиболее классических алгоритмов сортировки в оперативной памяти: сортировки слиянием и быстрой сортировки;
- изучение принципа работы сортировки слиянием во внешней памяти;
- понимание принципа работы быстрой сортировки во внешней памяти;
- понимание взаимосвязи между поиском и сортировкой во внутренней и внешней памяти.

В предыдущей главе мы познакомились с разными способами конструирования индексов в базах данных. В информатике индексы воплощают в себе основополагающую задачу о поиске. Еще одна основополагающая задача, которая возникает в базах данных – и практически везде, где бы то ни было еще, – это сортировка. Вспомните, сколько раз вы использовали функцию `sort()` в своем исходном коде для упорядочивания набора данных.

Помимо очевидных применений сортировки, существует большое число алгоритмов, которые используют сортировку в качестве своей подпрограммы. Например, в главе 2 мы обсуждали задачу о дедупликации (то есть об устранении дубликатов) и говорили о различных эффективных решениях с использованием хеширования. Хеширование дает хорошую производительность в среднем случае; однако если мы стремимся к наилучшей производительности наихудшего случая, предполагающей реальные сравнения элементов (а не сопоставление с помощью хеширования), устранение дубликатов влечет за собой сортировку данных. Это не означает, что оптимальный алгоритм дедупликации должен сортировать данные в явной форме, но он должен выполнять хотя бы тот объем работы, который требуется для сортировки, – поэтому с тем же успехом эту задачу можно решить сортировкой, а затем сканированием массива на наличие дублика-

тов. Другой несколько иной версией этой задачи является задача об уникальности элементов, которая на входе принимает неупорядоченный массив данных и выдает «да», если все элементы в массиве уникальны, и «нет» в противном случае. Задача об уникальности элементов аналогично задаче о дедупликации тоже требует сортировки в том смысле, что оптимальный алгоритм для этой задачи будет иметь время выполнения по крайней мере такое же высокое, как в алгоритме сортировки ($\Omega(n \log_2 n)$).

В этой главе мы сначала поговорим о разных контекстах, в которых возникает сортировка, и о проблемах, возникающих при сортировке крупных файлов в условиях ограниченного объема оперативной памяти. Затем рассмотрим два известных алгоритма сортировки, сортировку слиянием и быструю сортировку, а точнее методы их адаптации к внешней памяти. Мы будем делать это постепенно, чтобы продемонстрировать алгоритмические уловки, которые могут быть полезны в других задачах, похожих на сортировку. Наконец, мы покажем, как анализировать нижние границы для сортировки во внутренней и внешней памяти. Используя этот инструмент, мы сможем убедиться, что сортировка слиянием является оптимальным алгоритмом сортировки во внешней памяти.

11.1 Варианты использования сортировки

Сортировка распространена в вычислительных приложениях из многих прикладных областей. В мире геометрии сортировка точек по координатам довольно распространена и необходима для многих основополагающих процедур, таких как вычисление ближайшей пары точек на двумерной плоскости, алгоритмы заметающей прямой[89] и др. Рассмотрим следующее ниже применение сортировки в вычислительной геометрии и робототехнике.

11.1.1 Планирование движений робота

Представьте, что вы разрабатываете робота, который будет передвигаться по кухонному столу, избегая препятствий (допустим, роботу нужно собирать крошки со стола). У робота есть карта объектов и их 2D-следы в его окрестности, которые должны помогать роботу плавно перемещаться по столу, не врезаясь в объекты на нем. Объекты могут иметь различные формы, а фактические следы могут иметь сложную форму, что может затруднять планирование движений, поэтому с целью упрощения вычислений вместо фактических следов робот вычисляет так называемую *выпуклую оболочку* двумерного следа каждого объекта, то есть наименьший выпуклый многоугольник, содержащий след (см. рис. 11.1 с пояснениями).

Многие алгоритмы построения выпуклой оболочки используют сортировку, сортируя точки по координатам x и y. Один из популярных алгоритмов построения выпуклой оболочки можно наглядно представить как

[89] Англ. sweep line algorithm. – *Прим. перев.*

обертывание двумерного следа объекта подарочной бумагой. Процесс определения угла, который следует обогнуть следующим, предусматривает сортировку углов от текущего угла к другим углам следа. Более подробную информацию см. в алгоритме Джарвиса Марча (Jarvis March), также именуемом алгоритмом подарочной упаковки[90] для выпуклых оболочек (сортировка используется и во многих других алгоритмах для вычисления выпуклых оболочек, без подарочной упаковки в названии).

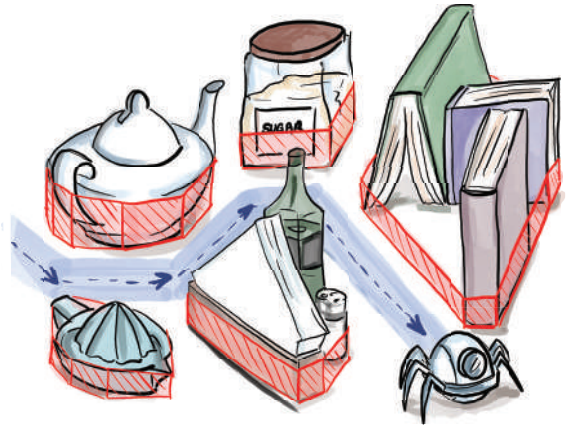


Рисунок 11.1 Алгоритмы планирования движения робота часто предусматривают вычисление выпуклых оболочек близлежащих объектов

В базах данных сортировка также широко используется для создания индексов, чтобы выполнять операции группировки по условию, сортировку результатов запросов и т. д. [1]. Помимо использования сортировки для реализации базовых операций с базами данных, крупным базам данных обычно требуется упорядочивать данные в соответствии с некоторыми критериями, которые предусматривают вычисления по разным столбцам. Рассмотрим пример базы данных из области биоинформатики как приложение сортировки.

11.1.2 Онкогеномика

У вас есть большая база данных геномов (полный генетический код человека), которые вы хотели бы упорядочивать в соответствии со склонностью к определенному типу злокачественных опухолей. Вы используете свою базу данных для проверки гипотезы недавнего исследования о том, что частота встречаемости определенных последовательностей, X и Y, в геноме играет роль в возникновении рака, и последовательность X делает это в два раза чаще, чем последовательность Y. Для этого мы упорядочиваем геномы в соответствии с оценочным баллом, который задействует число появлений указанных последовательностей, и используем оценочный балл на входе в функцию сравнения, как показано на рис. 11.2.

[90] Англ. giftwrap algorithm; см. https://en.wikipedia.org/wiki/Gift_wrapping_algorithm. – *Прим. перев.*

	ПОСЛЕДОВАТЕЛЬНОСТЬ X #	ПОСЛЕДОВАТЕЛЬНОСТЬ Y #	ОЦЕНОЧНЫЙ БАЛЛ 2·X + Y	РАНГ
ACTGGGTCACCGTGCA...	52	4	2·52 + 4 = 108	2
GGCCTATTGCGCGTGCA...	33	3	2·33 + 3 = 69	3
AGAGCCTTCTCCTTTTGA...	12	23	2·12 + 23 = 47	5
GCTTATCGCGAGCTAAA...	Ø	12	2·Ø + 12 = 12	6
GCTTGCTCGCTCATCTTC...	27	90	2·27 + 90 = 144	1
TCAAAGCGCAATCTCCTT...	19	10	2·19 + 10 = 48	4
GATCATGCTAGCTGATCC...	1	8	2·1 + 8 = 10	8
CGATTGACCTATTTCTAG...	5	1	2·5 + 1 = 11	7

Рисунок 11.2 В биоинформатике геномы нередко упорядочиваются в соответствии с различными критериями. В данном конкретном случае мы упорядочиваем геномы по числу появлений последовательностей *X* и *Y*. Присутствие последовательности *X* оценивается в два раза больше, чем присутствие последовательности *Y*

При сортировке принято предоставлять настраиваемую функцию сравнения, которую мы использовали для определения понятий *меньше, чем* и *равно*. Это особенно полезно для непримитивных типов данных, в которых порядок между элементами зависит от контекста и является более сложным. Например, функция сортировки в Python позволяет передавать настраиваемую функцию сравнения.

При определенных диапазонах и типах данных, размере набора данных и многих других параметрах может применяться другой алгоритм сортировки. Исследования сортировки и различных реализаций алгоритмов сортировки довольно обширны. Всесторонний обзор сортировки заслуживает отдельной главы или книги, поэтому, за исключением нескольких рассматриваемых в этой главе алгоритмов, мы не будем обсуждать многие тонкости сортировки.

Мы сосредоточимся на аспекте сортировки, когда данные становятся слишком большими, чтобы уместиться в оперативной памяти. Когда мы планируем отсортировать большой файл, который находится на диске, а в основной памяти может умещаться только по небольшой порции за раз, возникает главный вопрос: как определить высокоуровневую процедуру сортировки, которая будет сортировать весь файл, имея возможность работать только с малой порцией данных за раз? В частности, основное внимание в этой главе будет уделено выяснению того, как это сделать, минимизируя число переносов данных с диска.

11.2 Трудности сортировки во внешней памяти: пример

Представьте, что вы работаете в хостинговой компании, которая собирает данные о веб-запросах для своих клиентов. Допустим, вы хотите упорядочить все запросы за последний месяц, чтобы определить распределение времен обращения и найти запросы, которые заняли больше всего времени. Ваша компания собирает много данных, и данные организованы в одну большую таблицу, каждая строка которой представляет один запрос и всю связанную с ним информацию: IP-адрес, браузер, время обращения и т. д. Файл, подлежащий сортировке, занимает в общей сложности около 512 Гб, но вы располагаете лишь 4 Гб оперативной памяти.

Первое, что приходит на ум, – это то, что мы можем сортировать по 4 Гб данных за один раз. Если разделить изначальный файл на порции по 4 Гб и читать каждую порцию целиком в основную память, затем ее сортировать и записывать обратно, то мы получим частично отсортированный набор данных.

Этот шаг создания мини-сортированных списков, по сути, является отличной отправной точкой для применения алгоритма сортировки слиянием только во внешней памяти. Иногда мы будем использовать термин *двупутная сортировка слиянием*[91], чтобы обозначать традиционный алгоритм сортировки слиянием в качестве противопоставления многопутной сортировке слиянием, которую мы разработаем для внешней памяти. Давайте посмотрим на принцип работы двупутной сортировки слиянием (или просто сортировки слиянием) при ее слепом переносе во внешнюю память.

Но сначала краткий обзор: двупутная сортировка слиянием в оперативной памяти работает тривиальным делением массива на малые подмассивы сверху вниз до размера 1 и выполняет всю работу путем слияния этих массивов снизу вверх, по одной паре за раз. Слияние – это, по сути, сортировка. Указанное рекурсивное слияние превращает n сортированных списков размера 1 в $n/2$ сортированных списков размера 2, $n/4$ списков размера 4 и т. д., а затем, наконец, в 1 список размера n. Время выполнения сортировки слиянием описывается с использованием следующей ниже рекурсивной формулы $T(n)$, которая при развертывании рекурсии представляет число сравнений, требуемых для сортировки слиянием:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n).$$

Член $O(n)$ представляет время, необходимое для слияния на одном уровне (например, $n/2$ списков размера 2 в $n/4$ списков размера 4). Базовый случай рекурсии равен $T(1) = 1$, так как сортировка одноэлементного списка тривиальна. Разворачивая эту рекурсию с помощью мастер-метода или простого дерева, построенного распутыванием рекурсии, мы определяем, что время выполнения сортировки слиянием равно $O(n \log_2 n)$.

[91] Англ. two-way merge-sort. – *Прим. перев.*

11.2.1 Двупутная сортировка слиянием во внешней памяти

Прежде чем начать думать о том, как адаптировать двупутную сортировку слиянием к внешней памяти, давайте рассмотрим параметры, которые мы используем для анализа алгоритмов во внешней памяти. Значение N представляет размер входных данных (число записей), M представляет общий размер основной памяти, а B – размер блока.

Выгода от внешней памяти в части сортировки заключается в том, что при одном витке по всем данным (N/B передач блоков) можно получать N/M сортированных списков размера M, поэтому тривиальная разбивка на списки размером меньше M не имеет особого смысла. Естественно, это будет транслировано в базовый случай нашего алгоритма. После создания N/M сортированных списков, каждый размером M, алгоритм работает аналогично внутренней сортировке слиянием, в которой мы сливаем пары списков (см. пример с сортировкой игральных карт на рис. 11.3а).

При слиянии двух сортированных списков мы зачастую не можем одновременно хранить оба списка целиком в основной памяти, но для слияния нам нужно иметь лишь по одному блоку каждого из двух списков в основной памяти и выбирать наименьший оставшийся элемент из двух блоков до тех пор, пока один из них не будет полностью исчерпан; затем мы читаем следующий блок из списка. Процесс аналогичен слиянию k сортированных списков, как ранее было показано на рис. 9.7, который мы повторяем здесь (рис. 11.4), но при двупутной внешней сортировке слиянием $k = 2$.

Это означает, что время выполнения двупутной внешней сортировки слиянием составляет

$$T_{2\text{внеш}}(N) = 2T_{2\text{внеш}}\left(\frac{N}{2}\right) + O\left(\frac{N}{B}\right),$$

и базовый случай равен $T_{2\text{внеш}}(M) = O(M/B)$ передачам, необходимым только для чтения данных. Общая стоимость сортировки преобладает над линейной стоимостью создания первоначальных сортированных списков размера M, поэтому она в формулу не включена. Для того чтобы понять происходящее при двупутной внешней сортировке слиянием, важно понимать, что каждое чтение всех данных требует N/B передач. Для каждого витка по всем данным, который увеличивает размер списка в 2 раза (и сокращает число списков в 2 раза), требуется N/B операций ввода-вывода. Всего нужно $O(\log_2 N/M)$ таких витков, чтобы перейти от N/M списков размера M к 1 списку размера N, удваивая размер списка при каждом прогоне. Этот анализ (а также развертывание рекурсии) дает нам $O(N/B \log_2 N/M)$ операций ввода-вывода.

В целях проверки своих знаний о перемещении блоков туда-сюда во время двупутной внешней сортировки слиянием во внешней памяти сначала выполните следующее ниже упражнение.

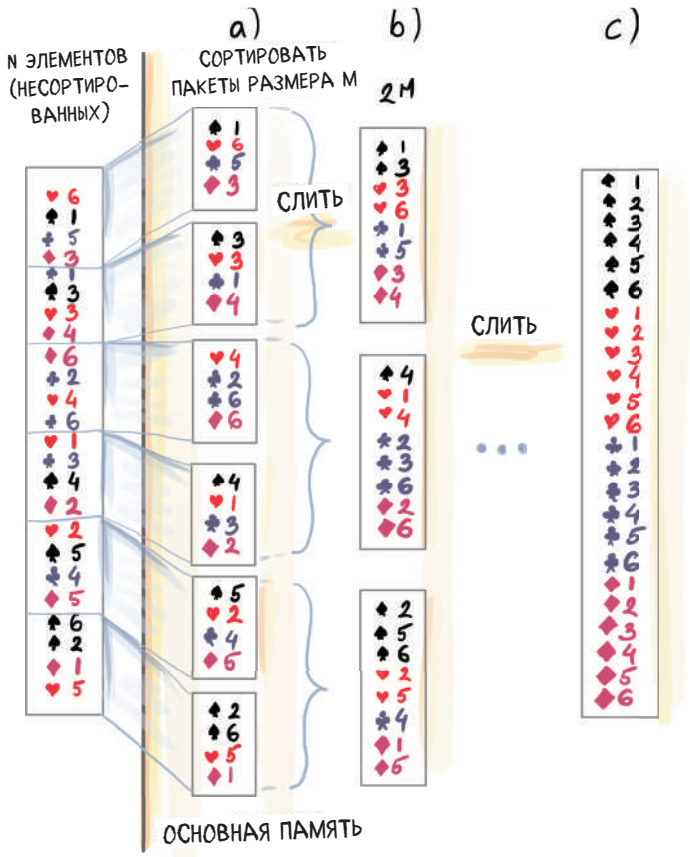


Рисунок 11.3 Двупутная сортировка слиянием, адаптированная к внешней памяти. При первом прогоне *N* элементов обрабатываются в *N* / *M* пакетах размера *M*, где сортируется каждый пакет размера *M*. Это наш «базовый случай» сортировки слиянием во внешней памяти (сортировка слиянием во внутренней памяти обычно начинается со списков размера 1). Затем, одна за другой, пары списков размера *M* обрабатываются и сливаются в списки размера 2*M*, потом 4*M* и т. д. В конце концов мы приходим к окончательному списку размера *N*

Упражнение 1

Проанализируйте число блочных запросов, необходимых для сортировки данных запросов из предыдущего примера, используя двупутную сортировку слиянием во внешней памяти. Распространенные размеры блоков варьируются от 8 до 64 Кб.

Мы можем добиться большего, чем двупутная внешняя сортировка слиянием, поэтому давайте вернемся к одновременному слиянию *k* сортированных списков. Рисунок 11.4 показателен тем, что демонстрирует процесс слияния сортированных списков, общий размер которых не умещается в оперативной памяти.

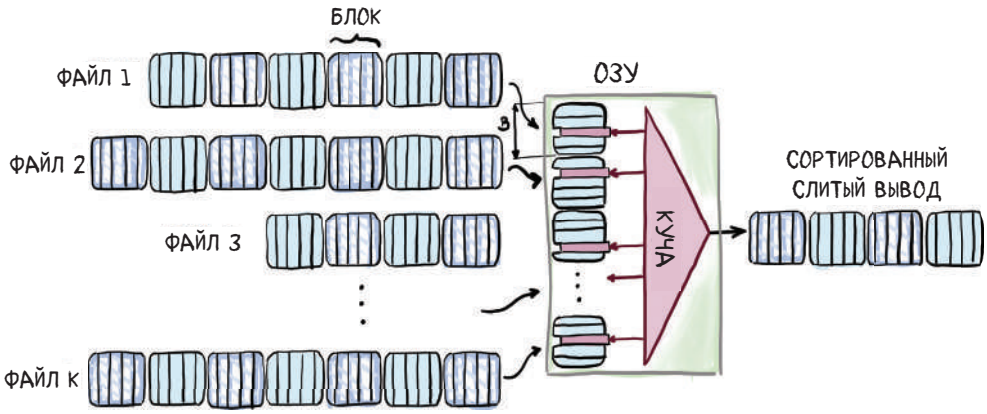


Рисунок 11.4 Слияние *k* сортированных списков во внешней памяти. Каждый список содержит один репрезентативный буферный блок, все время находящийся в памяти. Минимум каждого блока первоначально вставляется в кучу, откуда минимумы многократно извлекаются. При каждом извлечении элемента из кучи в нее по очереди вставляется следующий элемент из того же блока, из которого был получен минимум. Когда заканчиваются элементы одного блока, из того же списка вводится следующий блок

Согласно этому рисунку, мы можем слить до O(*M*/*B*) списков за раз, так как для представления каждого списка в оперативной памяти требуется всего один блок. Слияние множества списков за один раз дает значительный выигрыш, поскольку мы можем превратить *M*/*B* списков размера *x* в 1, и для этого мы используем то же число передач в память, что и при двупутной внешней сортировке слиянием для превращения *M*/*B* списков размера *x* в *M*/2*B* списков. Привнесение идеи слияния множества списков в двупутную внешнюю сортировку слиянием открывает дорогу для самого популярного алгоритма сортировки во внешней памяти – *M*/*B*-путной сортировки слиянием.

11.3 Сортировка слиянием во внешней памяти (*M*/*B*-путная сортировка слиянием)

В сортировке слиянием во внешней памяти, или *M*/*B*-путной сортировке слиянием, впервые представленной еще в 1980-х годах [2], используется идея одновременного слияния множества списков. Все начинается с создания базового случая в виде сортированных списков размера *M* для последующего слияния. Затем алгоритм переходит к одновременному слиянию *M*/*B* списков в один список, тем самым увеличивая размер списка между прогонами в *M*/*B* раз. Другими словами, мы начинаем со списков размера *M*, затем M^2/B, потом M^3/B^2 и т. д. до тех пор, пока не дойдем до одного списка размера *N*. Соответствующий пример приведен на рис. 11.5.

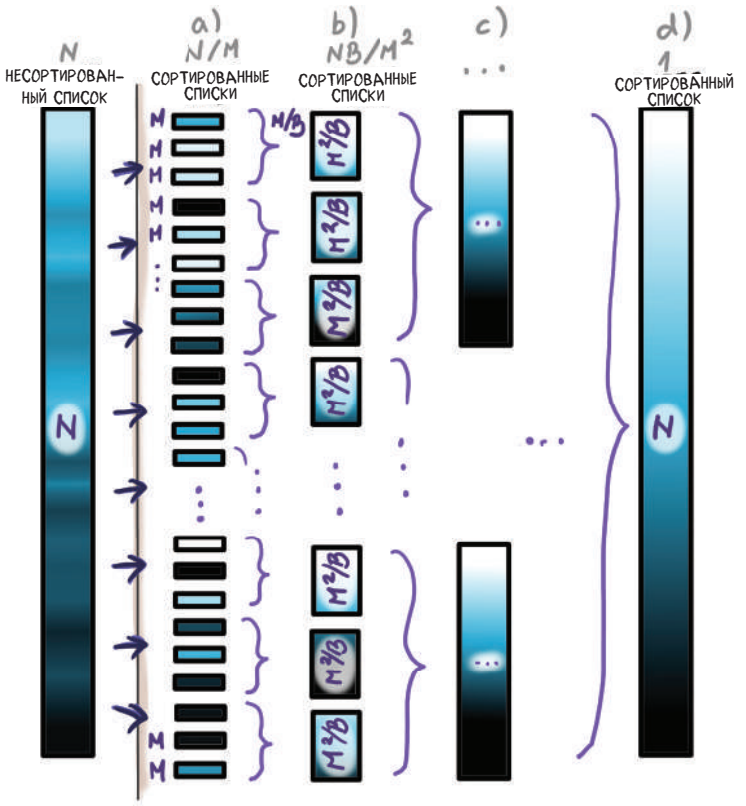


Рисунок 11.5 В *M/B*-путной сортировке слиянием мы начинаем с создания, за один прогон, *N/M* сортированных списков размера *M*. Затем эти списки объединяются дальше, по *M/B* за раз, чтобы в конечном итоге создать один сортированный список

Алгоритм похож на классическую внутреннюю сортировку слиянием в том смысле, что он рекурсивный, поэтому он тривиально делит списки до тех пор, пока они не достигнут размера *M*, сортирует каждый из них по отдельности и затем их рекурсивно объединяет. Рекурсивная формула, описывающая *M/B*-путную внешнюю сортировку слиянием, выглядит следующим образом:

$$T_{(\frac{M}{B})\text{внеш}}(N) = \frac{M}{B} T_{(\frac{M}{B})\text{внеш}}\left(\frac{N}{\frac{M}{B}}\right) + O\left(\frac{N}{B}\right).$$

Базовый случай такой же, как и для двупутной внешней сортировки слиянием, $T(M) = O(M/B)$. Давайте развернем эту рекурсию. Для того чтобы перейти от списков размера *M* к списку размера *N*, всегда увеличивая размер списка в *M/B* раз, требуется $\log_{M/B} N/M$ шагов. Каждый шаг нуждается в одном витке по всем данным, каждый из которых стоит $O(N/B)$ операций

ввода-вывода. Таким образом, общая стоимость M/B-путной внешней сортировки слиянием составляет $O(N/B \log_{M/B} N/M)$ операций ввода-вывода.

Сами по себе выражения для времен выполнения, возможно, мало что значат, но если визуально сравнить формулу *M/B*-путной внешней сортировки слиянием и двупутной внешней сортировки слиянием, то ключевое различие проявится в основании логарифма (2 против *M/B*). Насколько велико различие на самом деле? Мы привыкли пренебрегать основанием логарифма, так как обычно оно представляет собой разность с постоянным коэффициентом. Однако здесь задействованы параметры *M* и *B*, и мы не рассматриваем их как константы. Эти два времени выполнения отличаются в $\log_2 M/B$ раз. Для многих распространенных вариантов размера памяти и размера блока коэффициент разности может достигать даже 30 раз. Убедитесь в этом сами, выполнив следующее ниже упражнение и сравнив результаты с результатами, приведенными в упражнении 1.

Упражнение 2

Подсчитайте число передач блоков, которое используется алгоритмом *M/B*-путной внешней сортировки слиянием в нашем примере данных запросов. Напомним, что объем памяти составляет 4 Гб, а общий размер набора данных – 512 Гб. Используйте тот же размер блока, который вы использовали в упражнении 1.

Теперь, когда вы решили упражнение 2, у вас есть хорошее понимание числа операций ввода-вывода, требуемых алгоритмом сортировки. Однако, несмотря на то что при *M/B*-путной внешней сортировке слиянием мы пытаемся оптимизировать стоимость, связанную с диском, не следует пренебрегать внутренней памятью. Способ обработки операций в основной памяти сильно влияет на окончательное исполнение алгоритма. Наш пример слияния из главы 9 с описанием применения кучи в основной памяти для слияния *k* сортированных списков служит примером эффективного использования памяти в алгоритмах такого типа. В частности, при внешней сортировке слиянием можно поддерживать кучу размера *M/B*, чтобы поддерживать минимумы из каждого блока, представляющего его список. Благодаря такому подходу можно достигать оптимальности как во внешней, так и во внутренней памяти.

11.3.1 Поиск и сортировка: оперативная память по сравнению с внешней памятью

Давайте на минутку сделаем паузу и подумаем о связи между поиском и сортировкой и о том, как она меняется при переходе от внутренней памяти к внешней. Поиск и сортировка во внутренней памяти тесно связаны в следующем смысле: сбалансированное дерево двоичного поиска, структура данных, созданная для эффективного поиска (например, АВЛ-дерево, красно-черное дерево и т. д.), тоже может использоваться для оптимальной

сортировки данных. Вставка (а также поиск и удаление) в сбалансированном дереве двоичного поиска обходится в $O(\log_2 n)$, поэтому n вставок в дерево эффективно сортируют данные за $O(n \log_2 n)$ сравнений, и один симметричный обход может выводить данные в массив в линейном порядке. Тогда поэлементная стоимость сортировки равна поэлементной стоимости поиска, $O(\log_2 n)$.

Если, перенеся эту аналогию на внешнюю память, попытаться выполнить сортировку с использованием *B*-дерева, то мы получим производительность, далекую от оптимального алгоритма сортировки: N вставок в B-дерево стоит $O(N \log_B N)$ операций ввода-вывода, или если мы думаем о верхних уровнях *B*-дерева как о резидентах в основной памяти, то $O(N \log_B N/M)$ операций ввода-вывода. Поэлементная стоимость сортировки для M/B-путной внешней сортировки слиянием составляет всего лишь $O(1/B \log_{M/B} N/M)$ операций ввода-вывода, что существенно меньше времени на вставку в *B*-дерево. Другими словами, поэлементная стоимость сортировки намного меньше, чем поэлементная стоимость поиска во внешней памяти. То есть мы не можем перенести аналогию во внешнюю память.

Это различие важно тем, что оно показывает, что в пакетных задачах, таких как сортировка, можно эффективно использовать большой объем памяти (хорошим примером является слияние множества списков). Под пакетными задачами мы подразумеваем задачи, в которых нужно обрабатывать тонну данных и предоставлять результат только в самом конце. Например, пакетный поиск означает получение группы запросов и доставку ответов на эти запросы один раз в конце. В пакетной версии задачи поиска мы оптимизируем общее количество времени на решение всей задачи, и когда это становится целью, задачу о многочисленных запросах можно представлять в целом (то есть ответ на один запрос помогает ответить на еще один запрос и т. д.). В рамках такой конфигурации одновременная обработка большого объема данных бывает очень полезной, и большая оперативная память нам в этом помогает.

Это существенно отличается от условий, когда нам направляются одни и те же запросы и нам нужно сообщать ответы один за другим, и при этом мы оптимизируем сумму времени, затрачиваемого на ответ на каждый запрос. Эта последняя версия больше похожа на последовательность классических задач о поиске, и такой поиск не способен эффективно использовать большую память, за исключением хранения верхних уровней *B*-дерева в основной памяти, потому что результат сравнения с элементами из одного блока определяет следующий вносимый блок.

Если последние два абзаца кажутся вам чересчур философскими, то, вероятно, вы правы. Но это наша последняя глава, и мы чувствуем себя вправе в какой-то степени пофилософствовать. Дело в том, что при сортировке и других связанных с ней задачах мы и впрямь выигрываем от наличия большой оперативной памяти, тогда как в некоторых других задачах увеличение объема основной памяти помогает, но не настолько существенно.

Вы можете убедиться в этом, посмотрев на время выполнения сортировки по сравнению со временем выполнения поиска и на улучшение стоимости операций ввода-вывода, если объем основной памяти удваивается (то есть M становится $2M$).

Существует целый ряд других пакетных задач, которые могут выигрывать от наличия большой памяти так же, как это делает сортировка. До сих пор мы рассматривали только пример слияния множества списков как преимущество большой памяти. Далее мы рассмотрим внешнюю версию быстрой сортировки и увидим, как большой объем памяти позволяет ускорять обычную двупутную быструю сортировку (то есть выбор опорных точек).

11.4 Как насчет внешней быстрой сортировки?

Давайте начнем с краткого обзора внутренней быстрой сортировки. В отличие от сортировки слиянием, быстрая сортировка бóльшую часть своей работы выполняет сверху вниз, тщательно разбивая данные на разделы. Разбивка происходит на основе выбранной опорной точки, в которой данные далее делятся на элементы, меньшие или равные по размеру элементу в опорной точке, и элементы, превышающие элемент в опорной точке. Как только подлежащие делению массивы становятся размером 1, работа быстрой сортировки практически завершается.

Быстрая сортировка во внутренней памяти имеет более высокую репутацию, чем сортировка слиянием, а библиотеки сортировки чаще используют быструю сортировку, чем сортировку слиянием. Это может показаться необычным, если учесть, что быстрая сортировка не обеспечивает гарантий оптимальности в наихудшем случае, как это делает сортировка слиянием. Детерминированная быстрая сортировка, которая выбирает случайный опорный элемент в фиксированной точке (скажем, всегда с первой позиции в массиве), может варьироваться от $O(n \log_2 n)$ до $O(n^2)$, как и рандомизированная быстрая сортировка, которая выбирает опорную точку случайным образом. Тем не менее метод рандомизированной быстрой сортировки гораздо более безопасен, поскольку он эффективно обрабатывает случай почти отсортированных данных или любой регулярности в данных, которая может оказаться неблагоприятной для выбора опорной точки.

Быстрая сортировка может принудительно выполняться за $O(n \log_2 n)$, используя линейно-временной селекционный алгоритм медианы из медиан с гарантией оптимальности в наихудшем случае[92] [3], но этот алгоритм имеет различные сложности в практическом плане; с другой стороны, для получения асимптотически оптимального времени выполнения нам не нужны идеальные медианы.

Одним из главных преимуществ алгоритма быстрой сортировки является то, что он выполняется прямо на месте, поэтому все рекурсивные вы-

[92] Англ. median-of-medians worst-case linear-time selection algorithm. – *Прим. перев.*

зовы работают с одной и той же частью памяти, с изначальным массивом, подлежащим сортировке. Это означает, что мы не тратим время на копирование данных и отведение дополнительной памяти – на задания, которые замедляют сортировку слиянием. Экономия пространства также экономит время на быструю сортировку во внутренней памяти, но давайте посмотрим, можно ли эти эффекты оттранслировать во внешнюю память.

Для того чтобы разобраться в том, как эффективно транслировать быструю сортировку во внешнюю память, наше первое упражнение состоит в прямом переводе обычной двупутной быстрой сортировки без каких-либо существенных изменений в алгоритме.

11.4.1 Двупутная быстрая сортировка во внешней памяти

Прямая адаптация (рандомизированной) двупутной быстрой сортировки к внешней памяти довольно прямолинейна. Мы случайно выбираем местоположение опорной точки, вносим блок, содержащий опорную точку, а затем прокручиваем весь файл в памяти, блок за блоком, определяя для каждого элемента, меньше ли он, равен или больше элемента в опорной точке. В основной памяти находятся два буферных блока, которые накапливают элементы, принадлежащие двум группам, и когда блок заполняется с одной стороны, мы записываем его обратно на диск на соответствующую ему «сторону». После линейного числа переносов в память мы выполним один уровень разбивки на разделы (см. рис. 11.6).

Показанный на рис. 11.6 шаг разбивки требует $O(N/B)$ операций ввода-вывода. Затем мы рекурсивно выполняем тот же самый алгоритм на двух отдельных порциях файлов. Наш базовый случай срабатывает, когда размер сортируемого файла равен размеру памяти (M) или меньше. В этом случае мы извлекаем весь файл целиком, сортируем его в памяти и записываем обратно.

Давайте на минутку допустим, что выбранная опорная точка всегда разбивает данные на две равные половины. Тогда рекурсия, описывающая время выполнения двупутной внешней быстрой сортировки, идентична рекурсии двупутной внешней сортировки слиянием, и она дает $O(N/B \log_2 N/M)$ в качестве времени выполнения.

11.4.2 На пути к многопутной быстрой сортировке во внешней памяти

Следуя аналогии с сортировкой слиянием, в целях улучшения параллелизма в этом алгоритме можно было бы подумать об увеличении числа находимых опорных точек и выполнении M/B-путной разбивки вместо двупутной. Давайте на минутку предадимся этой идее. Допустим, что за $O(N/B)$ операций ввода-вывода можно отыскать $O(M/B)$ опорных точек, которые разбивают данные на $O(M/B)$ подмассивов. Это позволит нам перейти от

вот такого типа рекурсии для двупутной внешней быстрой сортировки:

$$T_{2q\text{внеш}}(N) = 2T_{2q\text{внеш}}\left(\frac{N}{2}\right) + O\left(\frac{N}{B}\right)$$

к вот такому типу рекурсии:

$$T_{(\frac{M}{B})q\text{внеш}}(N) = \frac{M}{B}T_{(\frac{M}{B})q\text{внеш}}\left(\frac{N}{\frac{M}{B}}\right) + O\left(\frac{N}{B}\right),$$

что приведет нас ко времени выполнения, эквивалентному времени выполнения *M/B*-путной внешней сортировки слиянием. Но не такой быстрой: конверсия из двупутной внешней быстрой сортировки в многопутную внешнюю быструю сортировку делается не так прямолинейно.

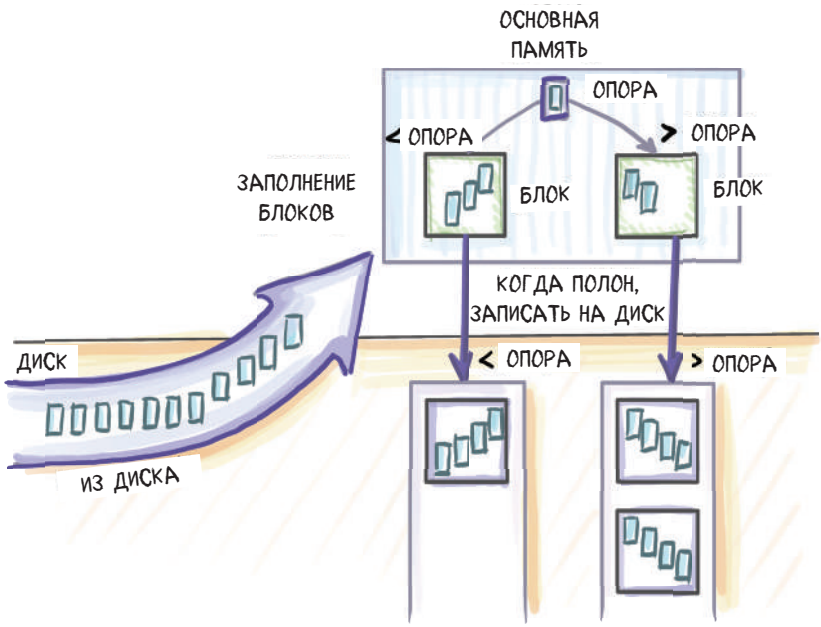


Рисунок 11.6 Снимок во время разбивки в рамках двупутной быстрой сортировки во внешней памяти. Данные последовательно пропускаются через основную память, и каждый элемент сравнивается с элементом в опорной точке. У нас один блок буферного пространства, служащий для накопления элементов меньшего размера, чем элемент в опорной точке, и один блок буферного пространства, служащий для накопления элементов большего размера, чем элемент в опорной точке. Сразу после того, как какой-либо из буферных блоков заполняется, он записывается обратно в надлежащее место на диске, куда добавляется либо левая, либо правая часть массива. Для рекурсивных вызовов, которые будут выполняться позже, важно, чтобы все элементы, которые меньше или больше элемента в опорной точке, были расположены рядом

Главная трудность, с которой мы сталкиваемся, заключается в том, что совсем не очевидно, как находить O(*M*/*B*) хороших опорных точек и выполнять разбивку на линейное число переносов блоков данных. Это можно сделать, если прибегнуть к рандомизированным опорным точкам, но рандомизированные опорные точки не дадут хорошей разбивки.

Еще одна идея заключается в использовании алгоритма медианы из медиан, который при переносе во внешнюю память требует O(*N*/*B*) передач, чтобы отыскать одну медиану. При рекурсивном применении этот алгоритм может находить O(*M*/*B*) медиан за O($N/B \log_2 M/B$) операций ввода-вывода; это противоречит нашему предыдущему плану, в котором мы обещали, что разбивка сработает (нерекурсивной частью рекурсивной формулы $T_{(M/B)q\text{внеш}}(N)$ было бы O(*N*/*B*)). Однако есть обходной путь.

11.4.3 Отыскание достаточного числа опорных точек

В размышлениях, изложенных в предыдущем разделе, оказалась лазейка. Мы сказали, что для достижения времени выполнения *M*/*B*-путной сортировки слиянием с использованием быстрой сортировки нужно выполнять *M*/*B*-путную разбивку (то есть уметь находить *M*/*B* хорошо распределенных опорных точек за *N*/*B* операций ввода-вывода). Мы можем обойтись гораздо меньшим числом опорных точек, и вот почему: чего бы мы ни достигли с помощью O(*M*/*B*) опорных точек с точки зрения времени выполнения, мы также можем достичь (асимптотически выражаясь) с помощью $(M/B)^c$ опорных точек, где c – это некая константа, такая что $0 < c < 1$. Поэтому отыскание $\sqrt{M/B}$ опорных точек или даже $\sqrt[3]{M/B}$ опорных точек по-прежнему дает время выполнения, асимптотически равное времени выполнения $T_{(M/B)q\text{внеш}}(N)$. Наличие $\sqrt{M/B}$ опорных точек будет удваивать глубину дерева рекурсии, но это не будет влиять на время выполнения асимптотически. Это будет наше первое ослабление условия задачи: отыскивать $\sqrt{M/B}$ опорных точек вместо *M*/*B*-опорных точек.

Вторым ослаблением будет то, что мы должны были узнать из внутренней быстрой сортировки: асимптотически оптимальная работа алгоритма вовсе не нуждается в делении данных на равные по размеру разделы. Для того чтобы облегчить нашу жизнь, мы перенесем эту идею во внешнюю память и попытаемся находить опорные точки, которые не обязательно должны иметь точно разнесенные ряды. Они будут достаточно хороши в том, что будут делить данные на O(*s*) подмассивов, где $s = \sqrt{M/B}$, и все подмассивы будут находиться в пределах величины постоянного коэффициента друг от друга. Некоторые подмассивы могут быть в два-три раза больше других подмассивов, и это будет нормально.

Давайте сделаем паузу, чтобы понять причину, по которой это не будет представлять проблем. Возвращаясь к обычной внутренней быстрой сортировке, напомним, что если всякий раз, когда мы выбираем опорную точку, она попадает точно в середину упорядоченного массива, то мы будем получать производительность O($n \log_2 n$). Если опорная точка всегда

находится где-то в средней половине рядов (то есть она никогда не находится в наименьших 25 % или наибольших 25 % данных), то время выполнения в наихудшем случае описывается с использованием следующего ниже рекуррентного соотношения:

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + n.$$

Оно также приводит к O($n \log_2 n$). Фактически даже если опорная точка разделяет данные на 1 % и 99 % ряды, время выполнения, генерируемое рекуррентным соотношением

$$T(n) = T\left(\frac{n}{100}\right) + T\left(\frac{99n}{100}\right) + n,$$

по-прежнему приводит к O($n \log_2 n$). Пока разделы находятся в пределах постоянных размеров друг от друга, это не должно давать асимптотически более низкую производительность, чем та, которую дают идеальные разделы. Мы воспользуемся этим фактом при отыскании s приближенных опорных точек в рамках внешней многопутной быстрой сортировки (теперь ее можно называть $\sqrt{M/B}$-путной быстрой сортировкой).

11.4.4 Отыскание достаточно хороших опорных точек

Разобьем изначальный набор из N элементов на N/M порций и отсортируем каждую порцию. Затем из каждой порции выберем каждый α-й элемент. Берем $\alpha = s/4\alpha = \sqrt{M/B}/4$. Обозначим множество этих выбранных элементов через $R \subseteq N$ (то есть представителей); множество будет иметь кардинальное число ~ N/α. Теперь рекурсивно задействуем селекционный алгоритм медианы из медиан, чтобы найти s опорных точек в R.

Сперва нужно доказать, что это возможно сделать за линейное число перемещений в память. Когда алгоритм медианы из медиан применяется рекурсивно ко множеству размера N/α, чтобы рекурсивно отыскивать $s = 4\alpha$ опорных точек, это стоит O($N/\alpha B \log_2 4\alpha$), что в сумме не стоит дороже O(N/B) операций ввода-вывода.

Далее нужно показать, что s медиан, выбранных из R, являются приближенными медианами в N. s медиан разбивают R на ~s разделов размера $k = N/s\alpha$, и каждый из этих элементов является представителем, который мы выбрали из порции размера M в изначальном наборе. Однако порции взаимно не упорядочены, поэтому в одном разделе могут содержаться элементы из разных порций. Например, первый раздел (содержащий наименьшие элементы) может содержать одного представителя из первой порции, пять представителей из второй порции, четыре из третьей порции и т. д. В любом случае, эти представители несут в себе элементы, которые идут до и после них в изначальном наборе.

Максимальное число элементов, которые k представителей из раздела могут нести с собой, равно, для каждого представителя, α элементам, которые идут после, и, для первого элемента в порции, элементам, которые идут перед ним. Оно равно не более

$$C_1 = k \times \alpha + \left(\frac{N}{M}\right) \times \alpha = \frac{N}{s} + \frac{N\alpha}{M}$$

элементам из изначального набора, а наименьшее число, которое может нести один раздел, схожим образом равно

$$C_2 = k \times \alpha - \left(\frac{N}{M}\right) \times \alpha = \frac{N}{s} - \frac{N\alpha}{M}.$$

Поскольку $s = q(\alpha)$, то $C_1 = C_2$, таким образом показывая, что s медиан, найденных в R, являются приближенными и достаточно хорошими медианами для изначального набора размера N (**http://mng.bz/zQva**).

11.4.5 Сведение всего воедино

Теперь, когда мы знаем, как отыскивать опорные точки за линейное время, давайте посмотрим, как работает эта версия внешней $\sqrt{M/B}$-путной быстрой сортировки (см. рис. 11.7).

Очень важно иметь возможность одновременно помещать s опорных точек в основную память вместе с $s + 1$ блоками, которые действуют как буферы для сбора элементов. Сразу после заполнения каждого блока он записывается обратно на диск. Как только все элементы будут обработаны, мы рекурсивно продолжим работу на $s + 1$ разделах.

Время выполнения этого алгоритма равно времени выполнения алгоритма внешней M/B-путной сортировки слиянием, $O(N/B \log_{M/B} N/M)$ операциям ввода-вывода. В следующем далее разделе мы увидим, что эта граница является оптимальной для любого алгоритма сортировки во внешней памяти.

11.5 Немного математики: почему сортировка слиянием во внешней памяти оптимальна?

Для того чтобы понять причину, по которой M/B-путная сортировка слиянием и $\sqrt{M/B}$-путная быстрая сортировка во внешней памяти являются оптимальными, сначала важно решить этот вопрос во внутренней памяти. Откуда мы знаем, что граница $O(n \log_2 n)$ является оптимальной для сортировки?

В самом начале, когда данные передаются алгоритму сортировки, мы не знаем, какая перестановка данных представляет правильно отсортированный порядок. Сложность задачи сортировки можно проанализировать, если подумать о том, насколько одно сравнение способствует устранению

некоторых перестановок, которые не соответствуют отсортированному порядку. Например, предположим, что наш набор данных состоит всего из трех элементов. Тогда 3! = 6 потенциальных перестановок могли бы дать окончательный отсортированный порядок: (a_1, a_2 a_3), (a_1, a_3, a_2), (a_2, a_1, a_3), (a_2, a_3, a_1), (a_3, a_1, a_2) и (a_3, a_2, a_1).

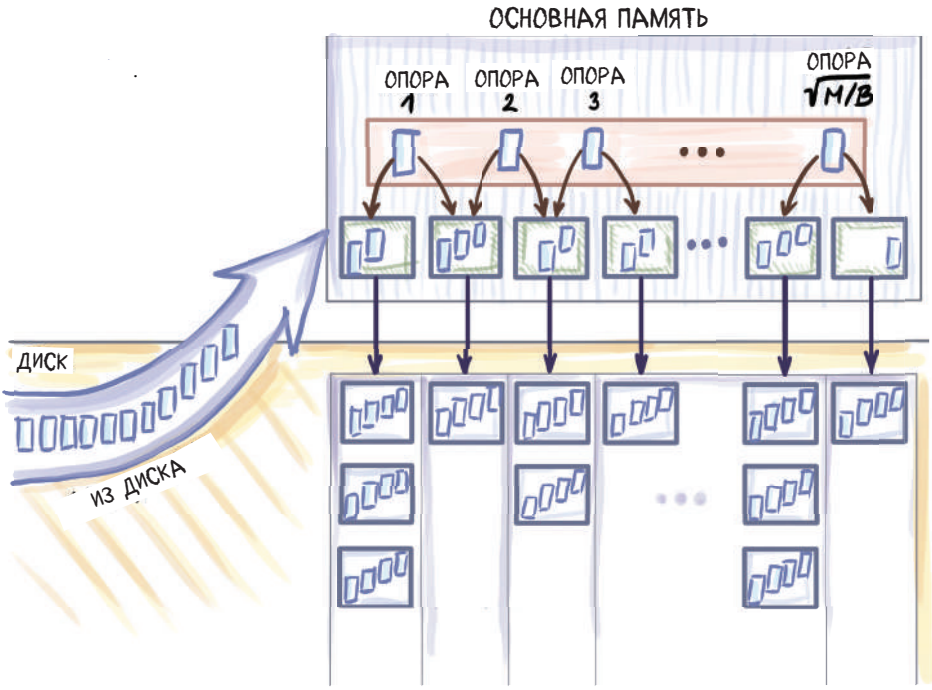


Рисунок 11.7 Снимок многопутной быстрой сортировки во внешней памяти. Вместо одной опорной точки мы отыскиваем O($\sqrt{M/B}$) опорных точек и прокручиваем весь набор данных в памяти. Каждый элемент маршрутизируется на основе сравнений с опорными точками в правильный буферный блок. Сразу после заполнения буферного блока любого раздела он записывается обратно на диск, а буфер опустошается. После прокручивания всех данных в основной памяти мы создаем O($\sqrt{M/B}$) разделов, на которых может быть снова рекурсивно применена быстрая сортировка. После достижения разделом размера *M* весь раздел считывается в основную память, сортируется и записывается обратно

Допустим, мы сравниваем a_2 с a_3 и узнаем, что $a_2 < a_3$. Это означает, что три перечисленные перестановки, где a_3 стоит перед a_2, должны быть исключены как потенциальные исходы. Поскольку перестановки симметричны в этом смысле, можно допустить, что хорошее сравнение может исключать не более половины оставшихся перестановок. Обратите внимание, что если наш алгоритм не очень хорош, то сравнение может сокращать время не так сильно или вообще не сокращать (представьте, что вы проводите одно и то же сравнение снова и снова). Но в том случае, если алгоритм предлагает осмысленные сравнения, потребуется как минимум $\log_2(n!)$

сравнений, чтобы получить одну перестановку. Упростив это выражение, мы получаем, что нижняя граница задачи сортировки равна $\Omega(n \log_2 n)$.

Как это работает во внешней памяти? Здесь наша единичная операция представляет собой передачу блока, поэтому возникает вопрос, насколько передача одного блока может способствовать сокращению числа возможных перестановок. Это будет в значительной степени зависеть от содержимого вводимого блока и содержимого основной памяти. Но что касается нижней границы, то нас интересует, *насколько* один блок может способствовать во время сортировки. Когда вводится один блок, он содержит не более B элементов, а память содержит не более $M - B$ резидентных элементов (см. рис. 11.8).

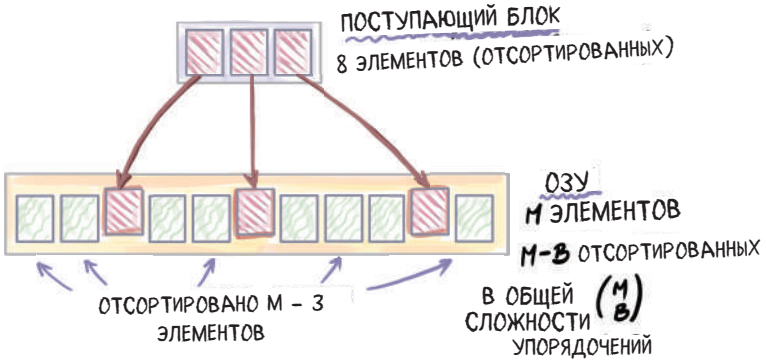


Рисунок 11.8 Мы анализируем число потенциальных упорядочений элементов B (содержимое одного блока), которое имеется в памяти, заполненной элементами, чтобы понять, насколько передача одного блока может способствовать сортировке. В общей сложности существует (M по B) упорядочений, и это тот коэффициент, с которым одна передача в память может уменьшать общее число перестановок, оставшихся для проверки

В целях упрощения расчетов мы примем допущение о том, что каждый отдельный блок отсортирован. Это уменьшает общее число перестановок с $N!$ до $N!/(B!\, N/B)$. Теперь, когда вносимый в основную память блок отсортирован и сама память отсортирована, общее число вариантов того, куда могут попадать B элементов, равно (M по B). Основываясь на этих двух величинах, мы получаем, что нижняя граница сортировки во внешней памяти равна

$$\log \binom{M}{B} \frac{N!}{B!^{\frac{N}{B}}},$$

которая после нескольких алгебраических манипуляций приводит к границе $\Omega(N/B \log_{M/B} N/M)$, соответствующей алгоритмам сортировки во внешней памяти. Приведенный выше анализ был адаптирован из статьи Эриксона

(Erickson) (**http://mng.bz/zQva**), с которой вы можете ознакомиться, если хотите получить более подробную информацию об алгебраических манипуляциях с этой нижней границей и нижними границами в целом.

11.6 Подведение итогов

Мы подошли к концу этой книги. Независимо от причины, по которой вы ее взяли: будь то попытка реализовать вероятностные структуры данных для решения задачи из конкретной предметной области или упрочение своих знаний в области крупномасштабных систем и ноу-хау в области устройства алгоритмов для собеседования в компании, занимающейся обработкой больших данных, – мы надеемся, что эта книга была хорошей инвестицией вашего времени. Если нет, то будем надеяться, что вам по меньшей мере понравились иллюстрации.

Если вы только начинаете работать в области крупномасштабных алгоритмов, то надеемся, что после прочтения этой книги вы лучше поймете спектр алгоритмических проблем, привносимых крупными наборами данных в современные системы, и, что важнее, найдете их интересными. Будем надеяться, что мы убедили вас в том, что такие задачи, как принадлежность множеству, поиск, сортировка, оценивание кардинального числа, взятие выборок и индексация баз данных для массивных наборов данных, являются интригующими и стимулирующими задачами и что размышления о способах их решения помогли вам развить и/или углубить новый, более тонкий взгляд на эффективность и производительность.

В конечном счете компромиссы, возникающие из-за ограниченности пространства и времени при работе с крупными данными, заставляют подходить к задачам более творчески, чем когда-либо прежде, и охватывать ошибки и несовершенства. Работа с массивными наборами данных учит тому, что мы не можем иметь все и сразу (и вовсе нам не нужны были массивные наборы данных, чтобы это понять!). В условиях растущего разрыва между нашими ресурсами и объемом данных, обрабатываемых приложениями, становится ясно, что успех многих приложений сегодня будет определяться тем, насколько хорошо они справляются с проблемами масштабируемости. Для того чтобы успешно с ними справляться, нужны инженеры, способные носить много шляп и сочетать ноу-хау в области алгоритмики и программирования со знаниями предметной области и математическими основами структур данных и алгоритмов. Эта книга – наш небольшой вклад в образование такого универсального инженера.

Резюме

- Сортировка – одна из самых известных задач в области информатики, и большой объем исследований посвящен оптимизации алгоритмов сортировки под различные контексты.

- Когда данные не умещаются в основной памяти, алгоритм сортировки должен переносить небольшие порции данных в основную память и поочередно сортировать каждую порцию.
- *M/B*-путная внешняя сортировка слиянием – это предпочтительный алгоритм, когда данные слишком велики, чтобы уместиться в оперативной памяти. Этот алгоритм выполняет слияние множества списков одновременно, в результате задействуя большой объем имеющейся памяти.
- Быстрая сортировка может быть адаптирована аналогичным образом под оптимальную работу во внешней памяти, выбирая более крупный набор опорных точек и в результате разбивая данные на множество отдельных подразделов вместо всего двух.
- Пакетные задачи, такие как сортировка, имеют более дешевую поэлементную стоимость, чем у поиска во внешней памяти. Это важное различие между оперативной и внешней памятью: сортировка в оперативной памяти может выполняться оптимально путем вставок в поисковую структуру, а ее выполнение во внешней памяти приводит к субптимальному алгоритму.
- Для того чтобы понять, является ли алгоритм сортировки оптимальным, важно понимать принцип работы нижних границ сортировки. Ключевым моментом во внутренней памяти является понимание того, насколько одно сравнение способствует устранению перестановок, которые не соответствуют отсортированному порядку; то же самое делается и во внешней памяти, но анализируя, насколько ввод одного блока помогает устранять перестановки.

Справочные материалы

Глава 1

1. Входная/выходная сложность сортировки и смежные проблемы, Aggarwal and J. S. Vitter, «The Input/Output Complexity of Sorting and Related Problems», Communications of the ACM, vol. 31, no. 9, pp. 1116–1127, 1988.
2. Реально-временная аналитика: методы анализа и визуализации потоковых данных, Ellis, Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data, Wiley, 2014.
3. Вероятностные структуры данных и алгоритмы для приложений с использованием больших данных, G. Andrii, Probabilistic Data Structures and Algorithms for Big Data Applications, Books on Demand, 2019; B. Ellis, Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data, Wiley, 2014.
4. Дисковые алгоритмы для больших данных, C. G. Healey, Disk-Based Algorithms for Big Data, CRC Press, 2016.
5. Извлечение знаний из массивных наборов данных, A. Rajaraman and J. D. Ullman, Mining of Massive Datasets, Cambridge Univer-sity Press, 2011.
6. Конструирование приложений с интенсивным использованием данных, M. Kleppmann, Designing Data-Intensive Applications, O'Reilly, 2017; A. Petrov, Database Internals, O'Reilly, 2019.
7. Архитектура компьютера: количественный подход, J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach (5th ed.), Morgan Kaufmann, 2011.
8. Материалы курсов MIT в свободном доступе, C. Terman, «MIT OpenCourseWare», Massachusetts Institute of Technol-ogy, Spring 2017, **https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-004-computation-structures-spring-2017/index.htm.**
9. Отставание задержки от пропускной способности, D. A. Patterson, «Latency Lags Bandwidth», Communications of the ACM, vol. 47, no. 10, pp. 71–75, 2004.
10. Архитектура компьютера, J. L. Hennessy and D. A. Patterson, Computer Architecture.
11. Отставание задержки от пропускной способности, D. A. Patterson, «Latency Lags Bandwidth».

Глава 2

1. ChunkStash: ускорение поточной дедупликации хранилища с использованием флеш-памяти, B. Debnath, S. Sengupta, and J. Li, «ChunkStash: Speeding up Inline Storage Deduplication Using Flash Memory», in Proceedings of the 2010 USENIX Confer-ence on USENIX Annual Technical Conference, p. 16, 2010.
2. Winnowing: локальные алгоритмы взятия цифровых отпечатков документов, S. Schleimer, D. S. Wilkerson, and A. Aiken, «Winnowing: Local Algorithms for Document Fingerprinting», in Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 76–85, 2003.
3. Введение в алгоритмы, T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms (3rd ed.), The MIT Press, 2009.
4. Линейное опробывание с постоянной независимостью, Pagh, R. Pagh, and M. Ruzic, «Linear Probing with Constant Independence», in Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing, San Diego, California, pp. 318–327, 2007.
5. Реализация словаря в Python на основе хеш-таблицы, python/cpython, «Python Hash Table Implementation of a Dictionary», Feb-ruary 20, 2020, **https://github.com/python/cpython/blob/master/Objects/ dictobject.c**.
6. Согласованное хеширование и случайные деревья: протоколы распределенного кеширования для разгрузки «горячих точек» во Всемирной паутине, D. K. Targer, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, «Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web», in Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, El Paso, Texas, 1997.
7. G. Valiant and T. Roughgarden, «CS168 The Modern Algorithmic Toolbox», April 1, 2019, **https://web.stanford.edu/class/cs168/l/l1.pdf**.
8. Chord: масштабируемый одноранговый протокол поиска для интернет-приложений, I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, «Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications», IEEE/ACM Transactions on Networking, vol. 11, no. 1, pp. 17–32, 2003.
9. Dynamo: высокодоступное хранилище в формате ключ-значение от Amazon G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, «Dynamo: Amazon's highly available key-value store», SIGOPS Review, vol. 41, no. 6, pp. 205–220, 2007.

Глава 3

1. Компромиссы между пространством и временем при хеш-кодировании с допустимыми ошибками, B. H. Bloom, «Space/Time Trade-Offs in Hash Coding with Allowable Errors», Communications of the ACM, vol. 13, no. 7, pp. 422–426, 1970; A. Broder and M. Mitzenmacher, «Network Applications of Bloom Filters: A Survey», Internet Mathematics, pp. 636–646, 2002.
2. Bigtable: система распределенного хранения структурированных данных, F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, «Bigtable: A Distributed Storage System for Structured Data», ACM Transactions on Computer Systems, vol. 26, no. 2, pp. 4:1–4:26, 2008.
3. Механизм хранилища Apache Cassandra, S. Lebresne, «The Apache Cassandra Storage Engine», 2012, **https://av.tib.eu/ media/39995**.
4. Не мусорить: как кешировать хеш во флеш-памяти, M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok, «Don't Thrash: How to Cache Your Hash on Flash», in Proceedings of the VLDB Endowment (PVLDB), vol. 5, no. 11, pp. 1627–1637, 2012.
5. Summary Cache: масштабируемый протокол обмена кеш-памятью в глобальной сети, L. Fan, P. Cao, J. Almeida, and A. Z. Broder, «Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol», IEEE/ACM Transactions on Networking, vol. 8, no. 3, pp. 281–293, 2000.
6. О мерах по обеспечению конфиденциальности фильтров Блума в легковесном биткоине, A. Gervais, S. Capkun, G. O. Karame, and D. Gruber, «On the Privacy Provisions of Bloom Filters in Lightweight Bitcoin», Proceedings of the 30th Annual Computer Security Applications Conference, pp. 326–335, 2014.
7. Summary Cache: масштабируемый протокол обмена кеш-памятью в глобальной сети, L. Fan, P. Cao, J. Almeida, and A. Z. Broder, «Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol», IEEE/ACM Transactions on Networking, vol. 8, no. 3, pp. 281–293, 2000.
8. Взвешенный фильтр Блума, J. Bruck, J. Gao, and A. Jiang, «Weighted Bloom Filter», Proceedings of IEEE International Symposium on Information Theory, pp. 2304–2308, 2006.
9. Фильтры Блума, адаптивность и задача о словаре, M. A. Bender, M. Farach-Colton, M. Goswami, R. Johnson, S. McCauley, and S. Singh, «Bloom Filters, Adaptivity, and the Dictionary Problem», in IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS), pp. 128–193, 2018.
10. Не мусорить: как кешировать хеш во флеш-памяти, M. A. Bender et al., «Don't Thrash: How to Cache Your Hash on Flash».

11. Искусство программирования. Т. 3: Сортировка и поиск, D. E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.), Addison Wesley Longman, 1998.
12. Не мусорить: как кешировать хеш во флеш-памяти, M. A. Bender et al., «Don't Thrash: How to Cache Your Hash on Flash».
13. Кукушечный фильтр: практически лучше, чем фильтр Блума, B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, «Cuckoo Filter: Practically Better Than Bloom», in Proceedings of the 10th ACM International Confer-ence on Emerging Networking Experiments and Technologies, Sydney, Australia, 2014.
14. Не мусорить: как кешировать хеш во флеш-памяти, M. A. Bender et al., «Don't Thrash: How to Cache Your Hash on Flash».

Глава 4

1. Современный инструментарий алгоритмов. Лекция №2: Приближенные «тяжеловесы» и набросок count-min, T. Roughgarden and G. Valiant, «The Modern Algorithmic Toolbox Lecture #2: Approximate Heavy Hitters and Count-Min Sketch», Stanford University, 2020, **https://web.stanford.edu/class/cs168/l/l2.pdf**.
2. Алгоритмические методы работы с большими данными. Лекция 7: Тяжеловесы, набросок count-min, M. Charikar and N. Wein, «CS369G: Algorithmic Techniques for Big Data, Lecture 7: Heavy Hitters, Count-Min Sketch», Stanford University, 2015–2016, **https://learn.fmi.uni-sofia.bg/pluginfile.php/200059/mod_resource/content/2/Heavy_hitters_-_count-min_sketch.pdf**.
3. Улучшенная сводка потока данных: набросок count-min и его применение, G. Cormode and S. Muthukrishnan, «An Improved Data Stream Summary: The Count-Min Sketch and Its Applications», Journal of Algorithms, vol. 55, no. 1, pp. 58–75, 2005.
4. Обработка речи и языка, D. Jurafsky and J. H. Martin, Speech and Language Processing (2nd ed.), Pearson, 2009.
5. Алгоритмы формирования набросков для оценивания точечных запросов в обработке естественного языка A. Goyal, H. Daume, III, and G. Cormode, «Sketch Algorithms for Estimating Point Queries in NLP», in Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, pp. 1093–1103, 2012.
6. Улучшенная сводка потока данных: набросок count-min и его применение, G. Cormode and S. Muthukrishnan, «An Improved Data Stream Summary: The Count-Min Sketch and Its Applications», Journal of Algorithms, vol. 55, no. 1, pp. 58–75, 2005.
7. Charikar and Wein, «CS369G: Algorithmic Techniques for Big Data, Lecture 7».

Глава 5

1. Руководство по конструированию алгоритмов, S. Skiena, The Algorithm Design Manual (2nd ed.), Springer, 2008.
2. HyperLogLog: анализ близкого к оптимальному алгоритма оценивания кардинального числа, P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier, «HyperLogLog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm», AOFA: Proceedings of the 2007 International Conference on Analysis of Algorithms, pp. 137–156, 2007.
3. HyperLogLog на практике: алгоритмическая инженерия современного алгоритма оценивания кардинального числа, S. Heule, M. Nunkesser, and A. Hall, «HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm», Proceedings of the 16th International Conference on Extending Database Technology, Genoa, Italy, pp. 683–692, 2013.
4. Алгоритмы вероятностного подсчета для приложений баз данных, P. Flajolet and G. N. Martin, «Probabilistic Counting Algorithms for Data Base Applications», Journal of Computer and System Sciences, vol. 31, no. 2, pp. 182–209, 1985.
5. Подсчет больших кардинальных чисел на основе алгоритма Loglog, M. Durand and P. Flajolet, «Loglog Counting of Large Cardinalities», European Symposium on Algorithms (ESA), pp. 605–617, 2003.
6. HyperLogLog: анализ близкого к оптимальному алгоритма оценивания кардинального числа, P. Flajolet et al., «HyperLogLog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm».
7. Линейно-временной алгоритм вероятностного подсчета для приложений баз данных, K. Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, «A Linear-Time Probabilistic Counting Algorithm for Database Applications», ACM Transactions on Database Systems, vol. 15, no. 2, pp. 208–229, 1990.
8. Алгоритмы подсчета активных потоков на основе битовых карт на высокоскоростных линиях связи, C. Estan, G. Varghese, and M. Fisk, «Bitmap Algorithms for Counting Active Flows on High-Speed Links», ACM Transactions on Networking, vol. 14, no. 5, pp. 925–937, 2006.

Глава 6

1. Извлечение знаний из массивных наборов данных, Partly adopted from A. Rajaraman and J. D. Ullman, Mining of Massive Datasets, Cambridge University Press, 2011.
2. Исследование методов обнаружения изменений, R. Sebastiao and J. Gama, «A Study on Change Detection Methods», Proceedings of the 14th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelli-gence, pp. 353–364, 2009.

Глава 7

1. Взятие простых случайных выборок из реляционных баз данных, F. Olken and D. Rotem, «Simple Random Sampling from Relational Databases», Proceedings of 12th VLDB Endowment, 1986.
2. Взятие случайных выборок с резервуаром, J. S. Vitter, «Random Sampling with a Reservoir», ACM Transactions on Mathemat-ical Software, vol. 11, no. 1, 37–57, 1985.
3. Взятие выборок из потока данных: базовые методы и результаты, P. J. Haas, «Data-Stream Sampling: Basic Techniques and Results», in M. Garofalakis, J. Gehrke, and R. Rastogi R. (Eds.), Data Stream Management: Processing High-Speed Data Streams, pp. 24–27, Springer, 2016.
4. Различные методы, используемые при работе со случайными цифрами: методы Монте-Карло, J. Von Neumann, «Various Techniques Used in Connection with Random Digits: Monte Carlo Methods», in A. S. Householder, G. E. Forsythe, and H. H. Germond (Eds.), Monte Carlo Method, vol. 12, pp. 36–38, US Government Printing Office, 1951.
5. О взятии смещенных резервуарных выборок в условиях эволюции потока, C. C. Aggarwal, «On Biased Reservoir Sampling in the Presence of Stream Evolution», Proceedings of the 32nd International Conference on Very Large Data Bases, pp. 607–618, 2006.
6. Взятие выборок из окна, двигающегося над потоковыми данными, B. Babcock, M. Datar, and M. Rajeev, «Sampling from a Moving Window Over Streaming Data», Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 633–634, 2002.
7. Взятие выборок из потока данных: базовые методы и результаты, P. J. Haas, «Data-Stream Sampling: Basic Techniques and Results», in M. Garofalakis, J. Gehrke, and R. Rastogi R. (Eds.), Data Stream Management: Processing High-Speed Data Streams, pp. 30–31, Springer, 2016.
8. Введение в stream: расширяемый фреймворк для исследования потоков данных с помощью R, M. Hahsler, M. Bonalos, and J. Forrest, «Introduction to stream: An Extensible Framework for Data Stream Clustering Research with R», Journal of Statistical Software, vol. 76, no. 14, pp. 1–50, 2017.

Глава 8

1. Временные границы отбора, M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, «Time Bounds for Selection», Journal of Computer and System Sciences, vol. 7, pp. 448–461, 1973.
2. Отбор и сортировка при ограниченном объеме хранения, J. I. Munro and M. S. Paterson, «Selection and Sorting with Limited Storage», Theoretical Computer Science, vol. 12, no. 3, pp. 315–323, 1980.

3. Медианы, и не только: новые методы агрегирования для сенсорных сетей, N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri, «Medians and Beyond: New Aggregation Techniques for Sensor Networks», Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, pp. 239–249, 2004.

Глава 9

1. Входная/выходная сложность сортировки и смежные проблемы, S. Aggarwal and J. S. Vitter, «The Input/Output Complexity of Sorting and Related Problems», Communications of the ACM, vol. 31, no. 9, pp. 1116–1127, 1988.

Глава 10

1. Введение в B-деревья и оптимизацию под операции записи, M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and Y. Zhan, «An Introduction to B-trees and Write-Optimization», vol. 40, no. 5, 2015.
2. Вездесущее B-дерево, D. Comer, «The Ubiquitous B-Tree», ACM Computing Surveys, vol. 11, no. 2, pp. 121–137, 1979.
3. R. Bayer and E. M. McCreight, «Organization and Mainte-nance of Large Ordered Indices», in Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control, pp. 107–141, 1970.
4. Нижние границы для словарей внешней памяти, G. S. Brodal and R. Fagerberg, «Lower Bounds for External Memory Dictionaries», in Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algo-rithms, pp. 546–554, 2003.
5. Журнально-структурированное дерево слияния (LSM-дерево), P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, «The Log-Structured Merge-Tree (LSM-tree)», Acta Informatica, vol. 33, no. 4, pp. 351–385, 1996.
6. Технологии хранения данных на основе LSM: обзор, C. Luo and M. J. Carey, «LSM-Based Storage Techniques: A Survey», VLDB Journal, vol. 29, pp. 393–418, 2020;
7. MyRocks: механизм хранения баз данных на основе LSM-дерева, обслуживающий социальный граф Facebook, Y. Matsunobu, S. Dong, and H. Lee, «MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph», Proceedings of the VLDB Endowment, vol. 13, no. 12, pp. 3217–3230, 2020.
8. Нижние границы для словарей внешней памяти, G. S. Brodal and R. Fagerberg, «Lower Bounds for External Memory Dictionaries»; M.A. Bender et al., «An Introduction to B-trees and Write-Optimization».

9. BetrFS: файловая система, оптимизированная под операции записи, W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. Kuszmaul, and D. E. Porter, «BetrFS: A Write-Optimized File System», in Pro-ceedings of the 13th USENIX Conference on File and Storage Technologies, vol. 11, no. 4, pp. 1–29, 2015.
10. Технологии хранения данных на основе LSM: обзор, C. Luo and M. J. Carey, «LSM-Based Storage Techniques: A Survey».

Глава 11

1. Реализация сортировки в системах баз данных, G. Graefe, «Implementing Sorting in Database Systems», ACM Computing Surveys, vol. 38, pp. 1–37, 2006.
2. Входная/выходная сложность сортировки и смежные проблемы, Aggarwal and S. J. Vitter, «The Input/Output Complexity of Sorting and Related Problems», Communications of the ACM, vol. 31, no. 9, pp. 1116–1127, 1988.
3. Временные границы отбора, M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, «Time Bounds for Selection», Journal of Computer and System Sciences, vol. 7, no. 4, pp. 448–461, 1973.

Об иллюстрации на обложке

Рисунок на обложке книги «Алгоритмы и структуры для массивных наборов данных» – Roussienne, или «русская женщина», взят из книги Жака Грассе де Сен-Совера, опубликованной в 1788 году. Каждая иллюстрация тонко прорисована и раскрашена вручную.

В те дни было легко определить место жительства людей и их ремесло или положение в жизни просто по их одежде. Издательство Manning прославляет изобретательность и инициативность вычислительного бизнеса обложками книг, основанными на богатом разнообразии региональной культуры многовековой давности, оживленными фотографиями из коллекций, подобных этой.

Предметный указатель

Ж

З

И

К

С

Книги издательства «ДМК ПРЕСС» можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;

Тел.: **+7(499) 782-38-89**. Электронная почта: **books@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине:

www.galaktika-dmk.com.

Джейла Меджедович, Эмин Тахирович

**Алгоритмы и структуры
для массивных наборов данных**

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод с английского *Логунов А. В.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Печать цифровая. Усл. печ. л. 24.86.
Тираж 100 экз.

Веб-сайт издательства: **www.dmkpress.com**

Стандартные алгоритмы и структуры при применении к крупным распределенным наборам данных могут становиться медленными — или вообще не работать. Правильный подбор алгоритмов, предназначенных для работы с большими данными, экономит время, повышает точность и снижает стоимость обработки.
Книга знакомит с методами обработки и анализа больших распределенных данных. Насыщенное отраслевыми историями и занимательными иллюстрациями, это удобное руководство позволяет легко понять даже сложные концепции. Вы научитесь применять на реальных примерах такие мощные алгоритмы, как фильтры Блума, набросок count-min, HyperLogLog и LSM-деревья, в своих собственных проектах.

Приведены примеры на Python, R и в псевдокоде.

«Доступное и прекрасно иллюстрированное введение в вероятностные дисковые структуры данных и алгоритмы».
Маркус Янг, Prosper Marketplace

«Повысит ваши знания об алгоритмах и структурах данных с уровня учебника до уровня реального мира».
Руи Лю, Oracle

«Прекрасно объясняет масштабируемые структуры данных и алгоритмы. Обязательное чтение для любого инженера данных».
Алекс Гоур, Shopify

«Подробный практический подход к работе с распределенными системами и архитектурами данных».
Сатедж Кумар Саху, Honeywell

Основные темы:

- вероятностные структуры данных в виде набросков;
- выбор правильного движка базы данных;
- конструирование эффективных дисковых структур данных и алгоритмов;
- понимание алгоритмических компромиссов в крупномасштабных системах;
- правильное формирование выборок из потоковых данных;
- вычисление процентилей при ограниченных пространственных ресурсах.

Джейла Меджедович получила докторскую степень в лаборатории прикладных алгоритмов Университета Стоуни Брук, Нью-Йорк.
Эмин Тахирович получил докторскую степень по биостатистике в Пенсильванском университете.

ISBN 978-5-93700-250-1

Интернет-магазин: www.dmkpress.com
Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф