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.

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).

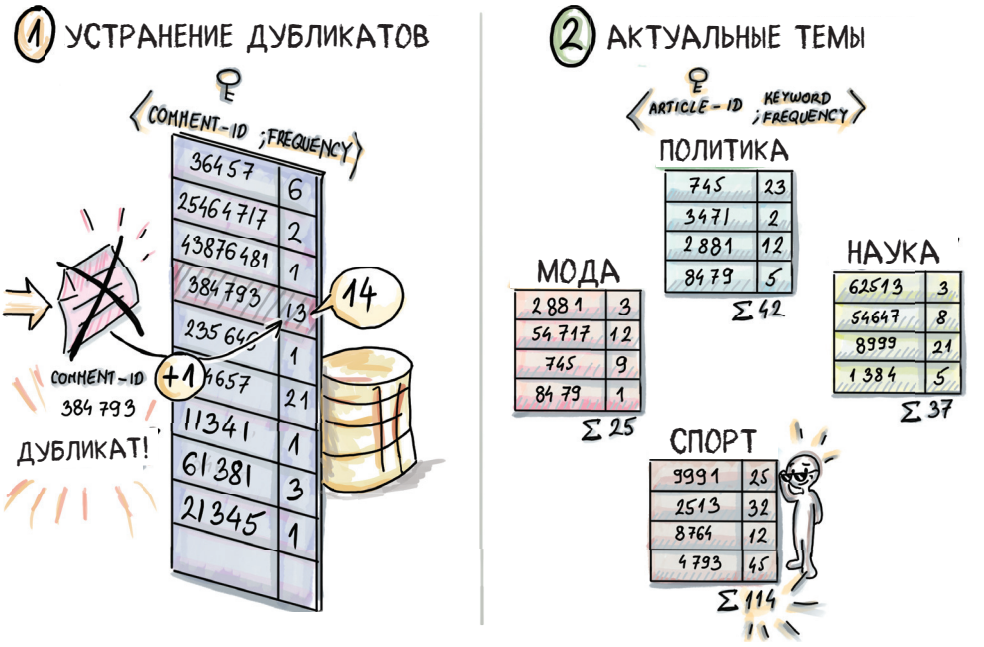

Рисунок 1.1 В данном примере создается хеш-таблица (`comment-id`, `frequency`), которая помогает хранить несовпадающие идентификаторы `comment-id` с их частотами. Входящий идентификатор `comment-id` 384793 в таблице уже содержится, и его частота возрастает. Помимо указанной хеш-таблицы, создаются хеш-таблицы, 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 «Руководство по конструированию алгоритмов» (3-е изд.) Скиены (издательство Springer, 2020); «Введение в алгоритмы» (3-е изд.) Кормена, Лейзерсона, Ривеста и Штейна (издательство MIT, 2022); «Алгоритмы» (4-е изд.) Седжвика и Уэйна (Addison-Wesley, 2011) и в качестве вводного и дружественного взгляда на предмет – «Грокаем ал-

[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.*

горитмы» Бхаргавы (издательство Manning, 2016)[6]. 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.

---

[6] В порядке появления: The Algorithm Design Manual (3rd ed., Skiena, Springer, 2020), Introduction to Algorithms (3rd ed., Cormen, Leiserson, Rivest, Stein, The MIT Press, 2022), Algorithms (4th ed., Sedgewick, Wayne, Addison-Wesley, 2011), Grokking Algorithms (Bhargava, Manning, 2016). – *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.

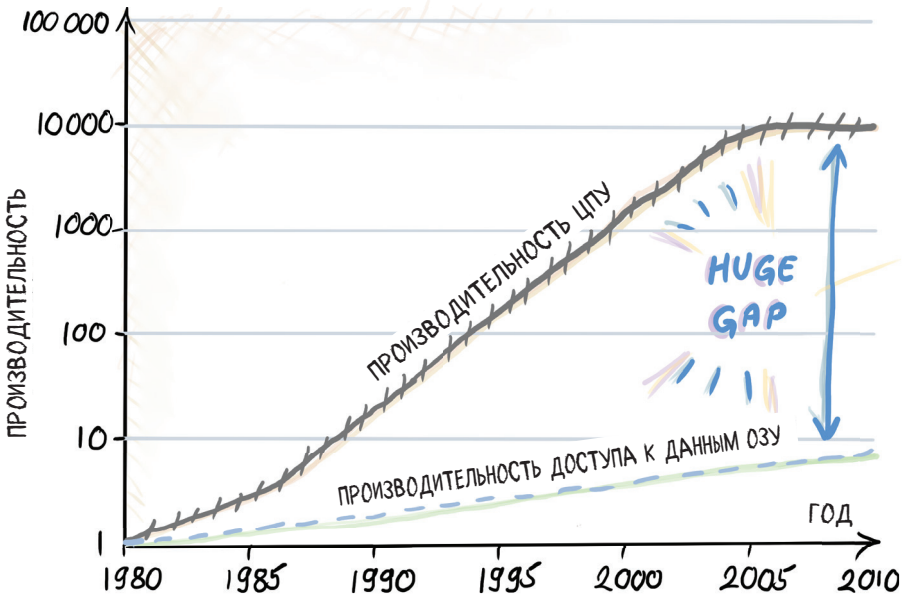

Рисунок 1.3 График разрыва в производительности центрального процессора и памяти, заимствованный из архитектуры вычислительной системы Hennessy & Patterson. На графике показан увеличивающийся разрыв между скоростями доступа к памяти центрального процессора и оперативной памяти (среднее число обращений 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.

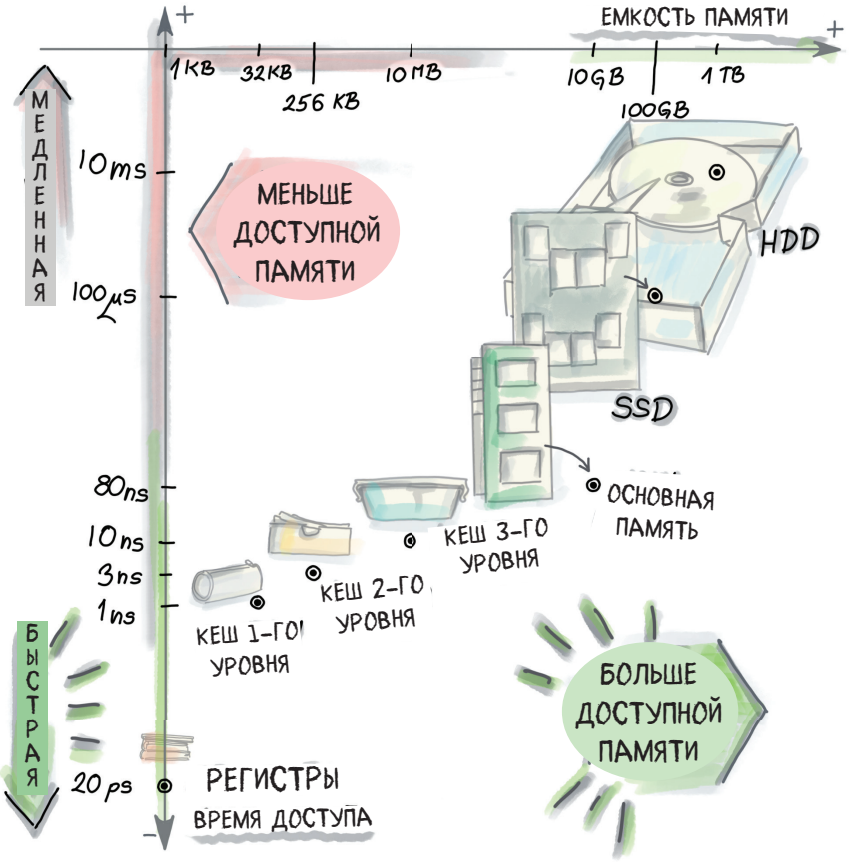

Рисунок 1.4 Разные типы памяти в компьютере. Начиная с регистров в левом нижнем углу, которые поразительно быстры, но в то же время очень малоемкостны, мы продвигаемся вверх (при этом скорость памяти становится медленнее) и вправо (емкость памяти увеличивается) с кешем 1-го уровня, кешем 2-го уровня, 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.

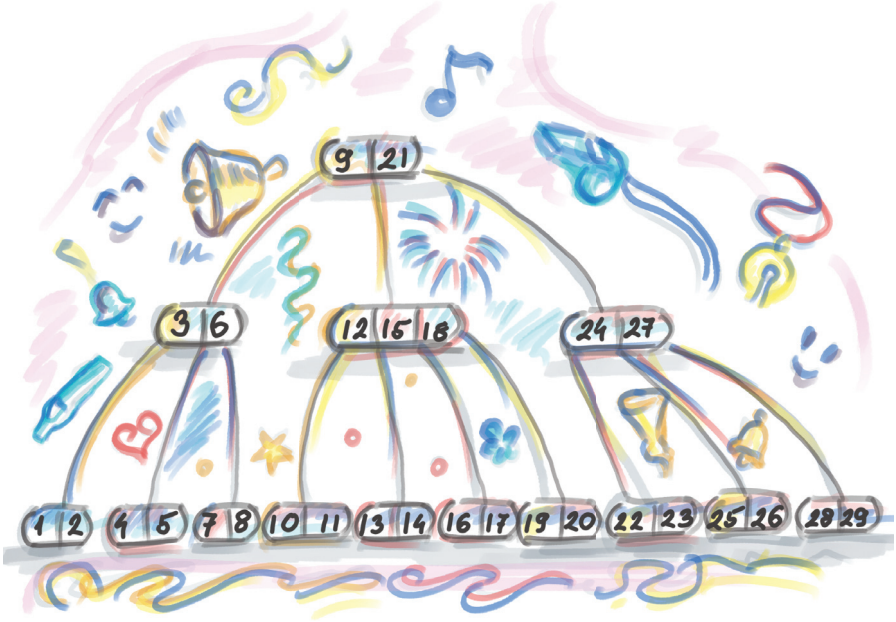

Рисунок 1.6 Эффективная структура данных с «танцами с бубнами»

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.

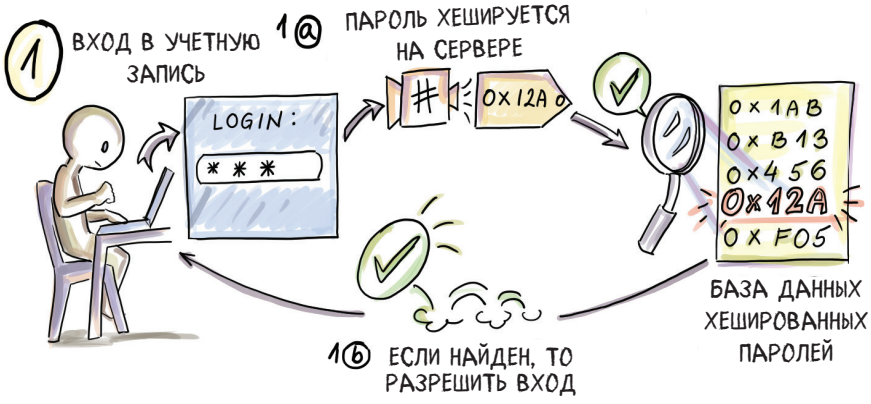

Рисунок 2.1 Вход в учетную запись электронной почты и хеширование

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[7] 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[8].

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

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

[8] 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[9], когда в крупный файл были внесены небольшие правки.

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

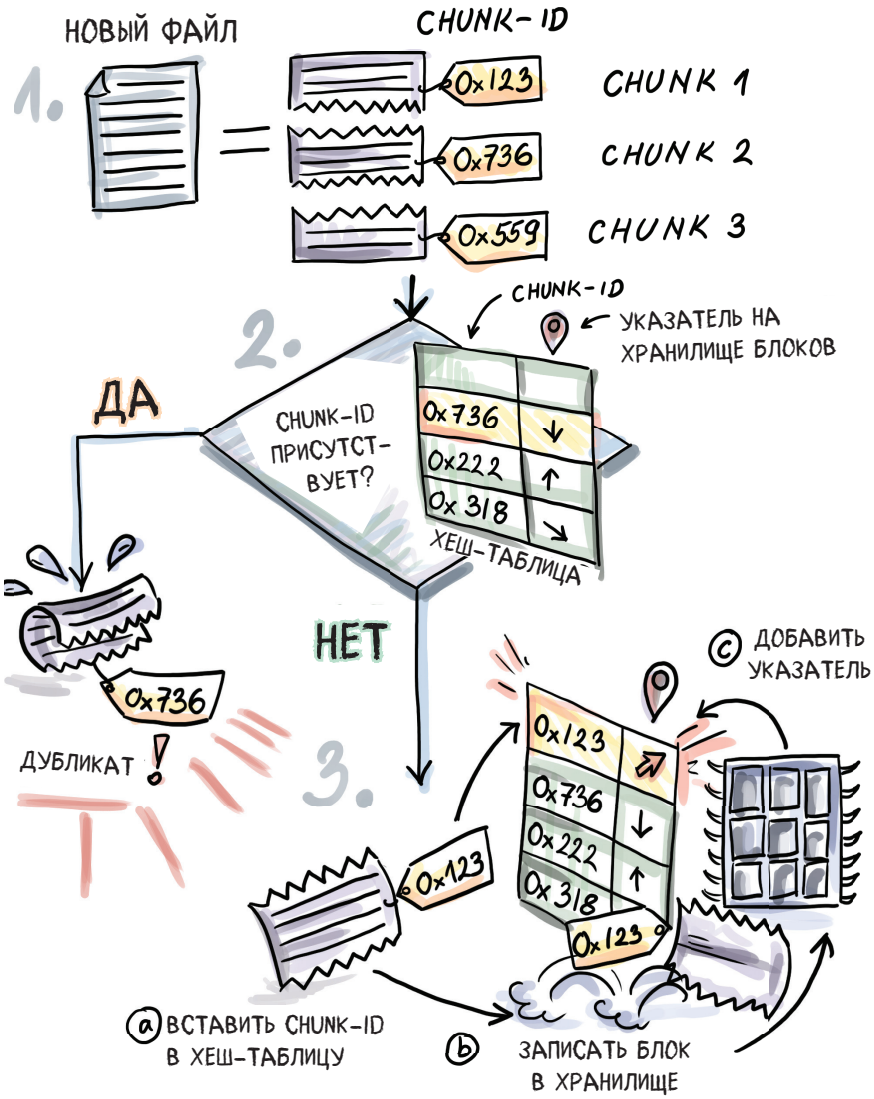

Рисунок 2.5 Процесс дедупликации в программных решениях по резервному копированию/хранению данных. По прибытии нового файла он разбивается на малые блоки. В нашем примере файл разбит на три блока, и каждый блок хешируется (например, блок 1 имеет `chunk-id 0x123`, а блок 2 имеет `chunk-id 0x736`). Идентификатор `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).

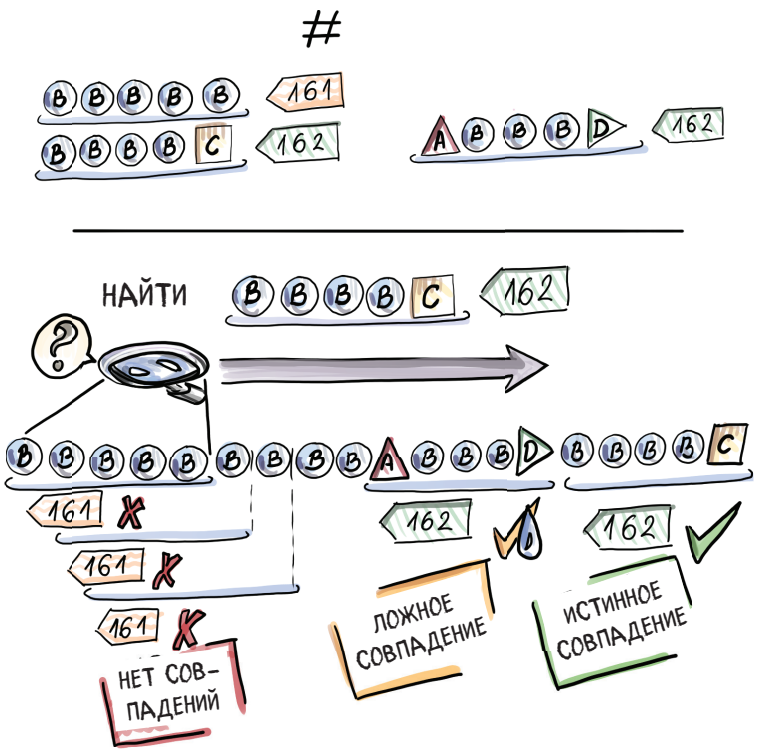

Рисунок 2.6 Пример алгоритма идентификации цифровых отпечатков Рабина–Карпа. Мы ищем шаблон `p=BBBBC` в большей строке `t=BBBBBBBBBBBBBBBBBC`. Хеш `BBBBC` равен 162 и не совпадает с хешем 161 для `BBBBB`, который встречается в начале длинной строки. По мере сдвига вправо мы неоднократно сталкиваемся 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.

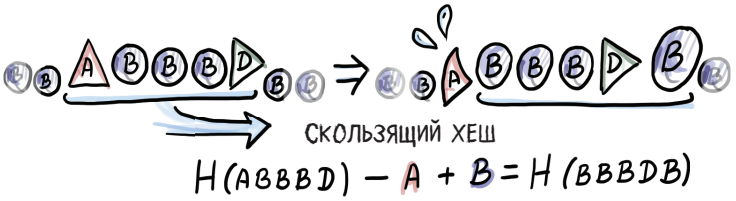

Рисунок 2.7 Скользящий хеш. Вычисление хеша для всех, кроме первой подстроки *t*, является постоянно-временной операцией. Например, для BBBDB нужно было «вычесть» A и «прибавить» B к ABBBD

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[10] 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.

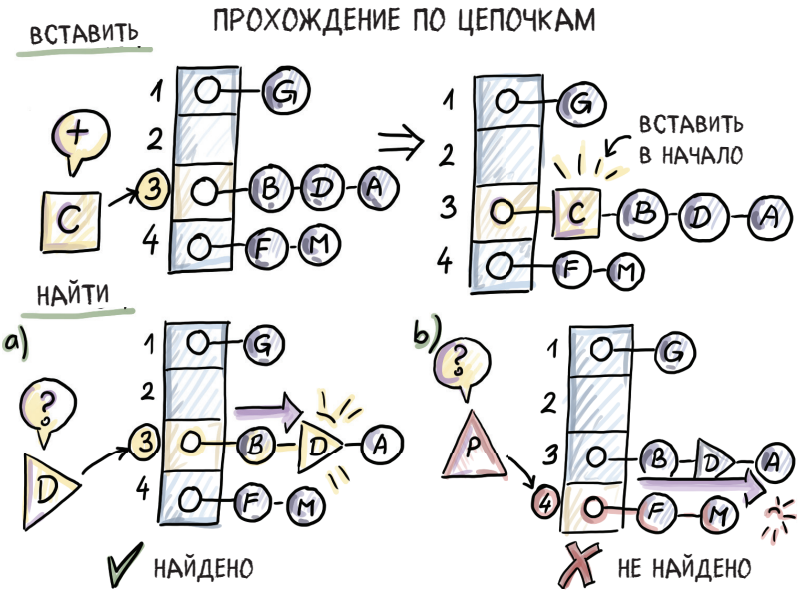

Рисунок 2.8 Пример вставки и поиска с прохождением по цепочкам

[10] Англ. chaining. – *Прим. перев.*

Линейное опробывание[11] – 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[12] на место удаляемого элемента. Пример линейного опробывания приведен на рис. 2.9.

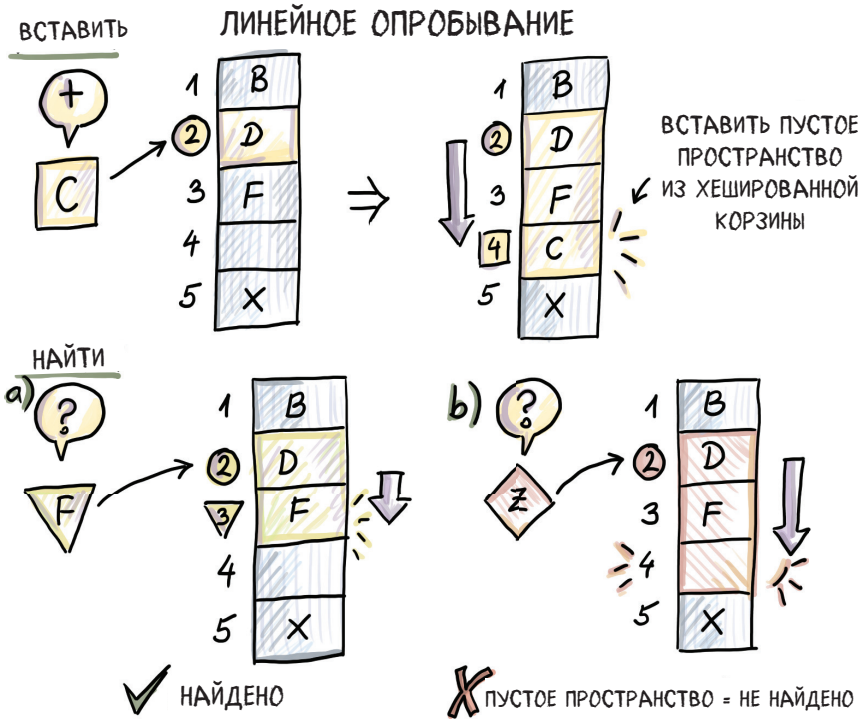

Рисунок 2.9 Пример вставки и поиска в линейном опробывании

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

[12] 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[13], 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())
```

[13] 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("Hello"))
print(mmh3.hash(key = "Hello", seed = 5, signed = True))
print(mmh3.hash(key = "Hello", seed = 20, signed = True))
print(mmh3.hash(key = "Hello", 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("Hello"))
print(mmh3.hash64(key = "Hello", seed = 0, x64arch= True, signed = True))
print(mmh3.hash64(key = "Hello", seed = 0, x64arch= False, signed = True))
print(mmh3.hash128("Hello"))
```

В результате чего генерируются следующие ниже (пары) хешей:

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