6.5. ВЫБОРКА ПО ВТОРИЧНЫМ КЛЮЧАМ

Мы закончили изучение поиска по *первичным ключам*, т. е. по ключам, которые однозначно определяют запись в файле. Однако иногда необходимо выполнить поиск, основанный на значениях других полей записей, помимо первичного ключа. Эти другие поля часто именуются *вторичными ключами* или *атрибутами* записи. Например, имея файл регистрации с информацией о студентах университета, можно найти всех второкурсников из Огайо, не специализирующихся по математике или статистике, или всех незамужних франкоговорящих аспиранток...

В общем случае полагается, что в каждой записи содержится несколько атрибутов, и необходимо найти все записи с некоторыми значениями этих атрибутов. Определение требуемых записей называется *запросом* (*query*). Обычно запросы подразделяются на следующие три типа.

a) *Простой запрос*, определяющий конкретное значение некоторого атрибута, например "СПЕЦИАЛИЗАЦИЯ = МАТЕМАТИКА" или "МЕСТОЖИТЕЛЬСТВО.ШТАТ = ОГАЙО".

b) *Запрос диапазона*, запрашивающий определенный диапазон значений некоторого атрибута, например "ЦЕНА < $18.00" или "21 < ВОЗРАСТ ≤ 23".

c) *Логический запрос*, состоящий из запросов предыдущих типов, скомбинированных при помощи логических операций AND, OR, NOT, например

"(КУРС = ВТОРОКУРСНИК) AND (МЕСТОЖИТЕЛЬСТВО.ШТАТ = ОГАЙО)
AND NOT ((СПЕЦИАЛИЗАЦИЯ = МАТЕМАТИКА) OR (СПЕЦИАЛИЗАЦИЯ = СТАТИСТИКА))".

Проблема разработки эффективных технологий поиска достаточно сложна уже для этих трех простых типов запросов, а потому более сложные типы запросов обычно не рассматриваются. Например, железнодорожная компания может иметь файл с информацией о текущем состоянии всех товарных вагонов. Запрос наподобие "Найти все пустые вагоны-рефрижераторы в радиусе 500 миль от Сиэтла" при этом в явном виде невозможен, за исключением случая, когда "расстояние от Сиэтла" является атрибутом, хранящимся в каждой записи, а не сложной функцией вычисления этой величины по значениям других атрибутов. Использование же логических кванторов в дополнение к AND, OR, NOT приведет к дальнейшему усложнению запросов, ограниченных исключительно воображением запрашивающего. Имея файл со статистикой по бейсболу, например, можно запросить информацию о самой длинной серии удачных ударов в ночных играх. Такие запросы несмотря на их сложность могут выполняться в течение одного прохода по соответствующим образом упорядоченному файлу. Однако могут быть и существенно более сложные запросы, например "Найти все пары записей, имеющие одинаковые значения в пяти и более атрибутах (без указания атрибутов, которые должны совпадать)". Такие запросы могут рассматриваться как общие задачи программирования, выходящие за рамки нашего обсуждения, хотя зачастую они могут быть разбиты на подзадачи, подобные рассматриваемым здесь.

Прежде чем приступить к изучению различных технологий получения информации по вторичным ключам, стóит рассмотреть экономический аспект вопроса. Хотя огромное количество приложений помещается в жесткие рамки трех типов запросов, описанных выше, далеко не для всех из них подходят технологии, которые мы будем изучать; в некоторых из них лучше было бы использовать ручную, а не машинную работу! Люди взобрались на Эверест просто потому, что он существует.

И многое альпинистское снаряжение было разработано именно по этой причине. Так и в информационных технологиях: увидев перед собой Эверест данных, люди тут же начинают разрабатывать снаряжение для его покорения, позволяющее в оперативном режиме ответить на все вопросы, которые только могут присниться в кошмарном сне. Зачастую они просто не задумываются о вопросах стоимости. Такие вычисления возможны, но они не подходят для любого приложения.

Например, рассмотрим следующий простой подход к выборке по вторичным ключам: после сбора *пакета* из нескольких запросов (*batching*) выполним последовательный поиск по всему файлу, получая все нужные записи. (Сбор пакета означает, что мы накапливаем некоторое количество запросов перед их выполнением.) Такой метод вполне удовлетворителен при не слишком больших файлах и отсутствии требования немедленно обработать поступающие запросы. Этот подход может использоваться с файлами на лентах и требует работы компьютера над запросом через некоторые интервалы времени, что делает его вполне экономичным в плане стоимости оборудования. Более того, он способен обрабатывать вычислительные запросы наподобие рассмотренного нами ранее (с использованием понятия "расстояние от Сиэтла").

Другой простой путь облегчить получение информации по вторичному ключу состоит в делегировании части работы *человеку*, который снабжен подходящими печатными указателями. Этот метод зачастую является наиболее разумным и экономичным способом работы (если, конечно, старая бумага перерабатывается при издании нового указателя), в особенности потому, что люди обычно отмечают интересные запросы при обеспечении удобного доступа к данным большого объема*.

Приложения, для которых не подходят приведенные простые схемы, включают в себя очень большие файлы, критичные ко времени ответа на запрос. Такая ситуация, например, возникает при одновременной выдаче запросов от множества пользователей или при машинной генерации запросов. Назначение данного раздела — рассмотреть способы обеспечения выборки информации при различных предположениях о структуре файла. К счастью, методы, которые будут обсуждаться, становятся все более и более пригодными для практического применения, а цена вычислений продолжает резко снижаться.

Существует немало хороших способов решения поставленных задач, однако (как читатель должен был понять из всех предварительных примечаний) все эти алгоритмы не так хороши, как алгоритм поиска информации по первичному ключу. Из-за огромного разнообразия файлов и приложений мы не в состоянии сколь-нибудь по́лно обсудить все возможности или проанализировать поведение каждого алгоритма в типичных условиях. В оставшейся части этого раздела представлены основные предлагаемые подходы, и дело читателя — решать, какая комбинация описанных технологий лучше всего подходит для решения той или иной конкретной задачи.

---

* Пожалуй, автор несколько недооценивает тенденции развития вычислительной техники и технологий баз данных. Так, например, оператор службы "09", конечно же, держит в голове сотни телефонов и на самые часто задаваемые вопросы способен ответить мгновенно. Но кто мешает использовать, например, кэширование запросов, которое не подвержено той же усталости к концу смены? Вычислительная техника и информационные технологии — очень быстро развивающаяся область, и то, что еще вчера казалось недостижимым, сегодня становится обыденным. Общая тенденция такова, что думать и принимать решения — это работа человека, а механически искать информацию — удел компьютера. — *Прим. перев.*

Инвертированные файлы. Первый важный класс технологий поиска по вторичному ключу основан на идее *инвертированных файлов*. Данный термин не означает, что файл перевернут вверх ногами; это означает изменение ролей атрибутов записей (вместо списка атрибутов записи приводится список записей с данным атрибутом).

Мы часто сталкиваемся с инвертированными файлами (пусть и под другими названиями) в нашей повседневной жизни. Например, инверсией англо-русского словаря будет словарь русско-английский; инвертированным файлом, соответствующем какой-нибудь книге, является ее предметный или авторский указатель. Бухгалтеры традиционно используют в работе "двойную бухгалтерию" (double-entry bookkeeping), в которой все сделки фиксируются как в кассовых книгах, так и в счетах клиентов, чтобы иметь возможность быстро получить информацию как по кассе, так и по клиентам.

В общем случае инвертированный файл не самодостаточен и должен использоваться совместно с "прямым", неинвертированным файлом. В инвертированном файле содержится избыточная информация — цена, которую приходится платить за ускорение поиска по вторичному ключу. Компоненты инвертированного файла называются *инвертированными списками*, т. е. списками всех записей с данным значением некоторого атрибута.

Подобно прочим спискам, инвертированные списки могут быть представлены в компьютере самыми разными способами, причем для каждого приложения и данных может быть выбран свой, наиболее подходящий способ представления. Одни вторичные ключи могут иметь только два значения (например, графа анкеты ПОЛ (хотя, как говорят, находятся уникумы, пишущие в этой графе что-то вроде "паркетный", а в англоязычной анкете на вопрос SEX отвечающие "regular"; впрочем, эти замечания больше относятся к вопросу о проверке корректности вводимых данных. — *Прим. перев.*)), и соответствующие списки могут оказаться весьма длинными; другие же, наподобие НОМЕР ТЕЛЕФОНА, как правило, коротки и с малым количеством дубликатов.

Представим, что хранить информацию в телефонной книге нужно так, чтобы все записи могли быть получены на основе имени, телефонного номера или адреса абонента. Одно из решений — создать три различных файла, ориентированных на поиск по своему типу ключа. Вторая идея состоит в комбинировании всех трех файлов, например, в три хеш-таблицы, служащие в качестве заголовков списков для метода цепочек. В последней схеме каждая запись файла должна быть элементом трех списков и, таким образом, должна иметь три поля ссылок. Этот так называемый *многосписочный* (*multilist*) метод проиллюстрирован на рис. 13 раздела 2.2.6 и обсуждается ниже. Третья возможность состоит в комбинировании трех файлов в один суперфайл по аналогии с каталогом библиотеки, в котором карточки авторов, названий книг и их тем располагаются вместе в алфавитном порядке.

После рассмотрения формата, использованного в предметном указателе этой книги, возникают новые идеи по представлению инвертированных списков. Для полей вторичного ключа, когда имеется около пяти элементов на одно значение атрибута, можно просто создать короткий последовательный список позиций записей (по аналогии с номерами страниц в указателе книги), следующих за значением ключа. В случае кластеризации записей может оказаться удобным указание диапазонов позиций (как, например, указание диапазона страниц 559–582). Если

записи в файле часто перемещаются, то лучше вместо позиций записей использовать первичные ключи — при этом не требуется обновление при перемещении записей. Например, ссылки в Библии всегда даются на главу и стих; во многих книгах ссылки основываются на номерах параграфов, а не страниц.

Ни одна из рассмотренных идей не подходит для атрибута с двумя значениями, например ПОЛ. В таком случае, конечно, достаточно только одного инвертированного списка, так как все не мужчины являются женщинами и наоборот. Если каждое значение соответствует примерно половине элементов файла, инвертированный список может быть ужасно длинным, но существует возможность получить очень изящное решение на бинарном компьютере с помощью представления в виде битовой строки, в которой каждый бит определяет значение конкретной записи. Так, битовая строка 01001011101... может означать, что первая запись в файле относится к мужчине, вторая — к женщине, следующие две — к мужчинам и т. д.

Таких методов достаточно для обработки простых запросов по определенным значениям атрибутов. Немного расширив эти методы, можно работать с запросами диапазонов, но вместо хеширования следует использовать схемы поиска, основанные на сравнениях (см. раздел 6.2).

Для логических запросов наподобие "(СПЕЦИАЛИЗАЦИЯ = МАТЕМАТИКА) AND (МЕСТОЖИТЕЛЬСТВО.ШТАТ = ОГАЙО)" необходимо найти пересечение двух инвертированных списков. Это можно сделать несколькими путями; например, если оба списка упорядочены, два прохода (по каждому из них) дадут все общие элементы. С другой стороны, можно выбрать *более короткий* список и просмотреть каждую из его записей, проверяя, соответствуют ли запросу прочие атрибуты. Однако этот метод работает только с запросами типа AND, но не OR и не очень хорош для внешних файлов, поскольку требует множества обращений к записям, не удовлетворяющим критериям запроса.

Такое же рассмотрение вопроса показывает, что организация в виде множества списков, описанная ранее, неэффективна для логических запросов к внешним файлам в связи с выполнением большого количества ненужных обращений. Например, представим, что указатель этой книги организован с помощью множества списков: каждый его элемент указывает только на последнюю страницу, на которой встречается соответствующий термин. Соответственно для каждого термина на этой странице имеется ссылка на предыдущую страницу с тем же термином и т. д. В таком случае для поиска всего материала, соответствующего запросу наподобие "[Анализ алгоритмов] AND ([Внешняя сортировка] OR [Внешний поиск])", придется перелистать очень много страниц. С другой стороны, ту же задачу можно решить, взглянув на две страницы имеющегося указателя и выполнив несложные операции над инвертированными списками для получения минимального подмножества страниц, которое удовлетворяет запросу.

Когда инвертированный список представлен в виде битовой строки, выполнение логических комбинаций простых запросов не вызывает трудностей, так как компьютеры работают с битовыми строками с относительно высокой скоростью. Для смешанных запросов, одни атрибуты которых представлены в виде последовательных списков записей, а другие — в виде битовых строк, несложно конвертировать последовательные списки в битовые строки, а затем выполнить над ними необходимые логические операции.

Здесь может оказаться небесполезным следующий пример. Предположим, что имеется 1 000 000 записей по 40 символов и файл хранится на дисках MIXTEC (см. раздел 5.4.9). Файл сам по себе заполняет два дисковых модуля, а инвертированные списки, вероятно, займут несколько больший объем. На каждой дорожке содержится 5 000 символов = 30 000 бит, так что инвертированный список для определенного атрибута займет не более 34 дорожек (это максимальное число дорожек соответствует самому короткому возможному представлению в виде битовой строки). Предположим, что имеется весьма сложный запрос, представляющий логическую комбинацию из десяти инвертированных списков. В худшем случае придется считать 340 дорожек информации из инвертированного файла с общим временем чтения $340 \times 25\,\text{ms} = 8.5\,\text{s}$. Среднее время задержки будет равно приблизительно половине времени чтения, но при аккуратном программировании его можно исключить. Сохраняя первые дорожки каждого битового списка на одном цилиндре, вторые — на следующем и т. д., можно практически исключить время поиска дорожки на диске, и в результате ожидать, что максимальное время поиска по диску составит около $34 \times 26\,\text{ms} \approx 0.9\,\text{s}$ (или в два раза больше при использовании двух независимых дисков). Наконец, если запросу удовлетворяют q записей, потребуется около $q \times (60\,\text{ms}$ (поиск по диску) $+ 12.5\,\text{ms}$ (скрытая задержка) $+ 0.2\,\text{ms}$ (чтение)) дополнительного времени, чтобы получить их для последующей обработки. Таким образом, грубая оптимистическая оценка общего ожидаемого времени для обработки этого довольно сложного запроса равна $(10 + .073q)\,\text{s}$. Полученное число может быть сопоставлено с примерно 210 s, требующимися для чтения всего файла с максимальной скоростью при тех же предположениях без использования инвертированных списков*. Этот пример показывает, что оптимизация по пространству тесно связана с оптимизацией по времени при работе с дисками; время обработки инвертированных списков примерно соответствует времени, необходимому для их поиска на диске и чтения.

В приведенном обсуждении в большей или меньшей степени полагалось, что файл не растет и не уменьшается во время запроса к нему. Однако что делать, если требуются частые обновления? Во многих приложениях для этого достаточно просто собрать в один пакет некоторое количество запросов на обновление, которые выполняются в момент, когда нет обрабатываемых запросов. Если же, напротив, обновление данных имеет наивысший приоритет, привлекательным представляется использование B-деревьев (см. раздел 6.2.4). Весь набор инвертированных списков может быть представлен в виде одного гигантского B-дерева со специальным соглашением о том, что узлы ветвей содержат значения ключей, а листья — как ключи, так и указатели на записи. Кроме того, обновленные версии файла могут обрабатываться и другими методами, которые будут рассмотрены ниже.

Геометрические данные. Огромное количество приложений работает с точками, линиями и прочими фигурами в двух или более измерениях. Одним из первых подходов к запросам, ориентированным на расстояния, было "почтовое дерево"

* Просьба смотреть на *относительные* значения приводимых автором величин, абстрагируясь от *абсолютных* значений, которые не соответствуют современной технике. Так, без специальной оптимизации в многозадачной OS/2 с жестким диском IDE при одновременном выполнении других операций для чтения 1 000 000 записей размером по 40 байт на компьютере переводчика этого раздела потребовалось 2.3 s. — *Прим. перев.*

(post-office tree), предложенное в 1972 году Брюсом Мак-Наттом (Bruce McNutt). Предположим, например, что необходимо ответить на запрос "Какой город является ближайшим к точке x?", имея заданную точку x. Каждый узел дерева Мак-Натта соответствует городу y и "тестовому радиусу" r; левое поддерево узла соответствует всем городам z, последовательно введенным в эту часть дерева, таким, что расстояние от y до $z \le r + \delta$, а правое поддерево точно такое же для расстояний $\ge r - \delta$. Здесь δ — данный допуск; города, находящиеся на расстоянии от $r - \delta$ до $r + \delta$ от y, должны входить в *оба* поддерева. Поиск в таком дереве позволяет определить все города на расстоянии не более δ от заданной точки (рис. 45).

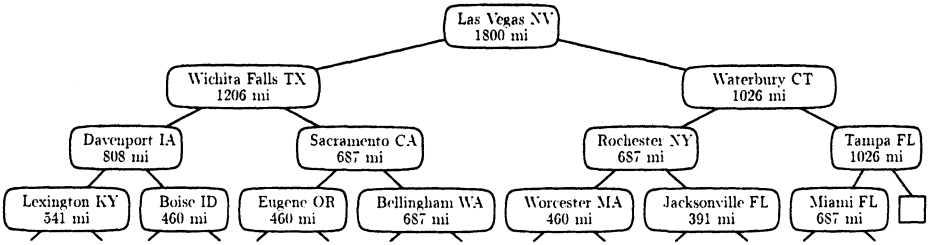


Рис. 45. Верхние уровни примера "почтового дерева". Для поиска всех городов вблизи данной точки x начинаем поиск от корня. Если x находится в пределах 1 800 миль от Лас-Вегаса, идем по дереву влево, в противном случае идем вправо. Повторяем этот процесс до тех пор, пока не будет достигнут конечный узел. Метод построения дерева гарантирует, что все деревья в пределах 20 миль от x встретятся в процессе поиска.

На основе этой идеи Мак-Натт и Эдвард Прингс (Edward Pring) провели несколько экспериментов с использованием в качестве примера базы данных, содержащей 231 наиболее населенный город континентальной части США в случайном порядке. Тестовый радиус уменьшался регулярно, а именно заменой r на $0.67r$ при перемещении влево, и на $0.57r$ — при перемещении вправо, за исключением случая, когда последовательно выбирались две правые ветви (при этом r оставался неизменным). В итоге было получено дерево из 610 узлов при $\delta = 20$ миль и из 1 600 узлов при $\delta = 35$ миль. Верхние уровни меньшего из получившихся деревьев показаны на рис. 45. (В оставшейся части дерева Орландо (штат Флорида) появляется ниже Джексонвилля и Майями. Некоторые города встречаются очень часто. Так, Брокстон (штат Массачусетс) встречается в 17 узлах!)

Быстрый рост файла при увеличении δ указывает на ограниченность применения почтовых деревьев. Пожалуй, лучше работать непосредственно с *координатами* каждой точки, рассматривая координаты как атрибуты или вторичные ключи; в таком случае можно создать логический запрос, основанный на диапазонах значений ключей. Например, предположим, что записи в файле относятся к городам Северной Америки и что запрос выполняет поиск всех городов, для которых справедливо

$$(21.49^\circ \le \texttt{ШИРОТА} \le 37.41^\circ) \quad \texttt{AND} \quad (70.34^\circ \le \texttt{ДОЛГОТА} \le 75.72^\circ).$$

Посмотрев на карту, можно увидеть, что множество городов удовлетворяет диапазону параметра ШИРОТА. Также найдется немало городов, имеющих соответствующий

параметр ДОЛГОТА, но сложно найти город, отвечающий обоим условиям одновременно. Один из подходов к подобным *ортогональным запросам диапазонов* состоит в разбиении множества всех возможных значений ШИРОТА и ДОЛГОТА таким образом, чтобы на каждый атрибут приходилось небольшое число классов (например, усечением значений до ближайшего меньшего значения, кратного 5°), и построении инвертированного списка по всем комбинированным классам (ШИРОТА, ДОЛГОТА). Это чем-то напоминает карты, состоящие из страниц для каждого локального региона. Используя 5-градусные интервалы, рассматриваемый здесь запрос будет обращаться к восьми страницам, а именно — $(20°, 70°)$, $(25°, 70°)$, ..., $(35°, 75°)$. Запрос диапазона в данном случае должен обработать каждую страницу, либо переходя к более мелким частям страницы, либо обращаясь непосредственно к записям — в зависимости от количества записей, соответствующих этой странице. По сути, получается структура дерева с двумерным ветвлением в каждом внутреннем узле.

Существенное усовершенствование этого подхода, называемое *сеточным файлом*, было разработано Ю. Нивергельтом (J. Nievergelt), Г. Гинтербергером (H. Hinterberger) и К. К. Севчик (K. C. Sevcik) [*ACM Trans. Database Systems* **9** (1984), 38–71]. Если каждая точка x имеет k координат $(x_1, \ldots, x_k)$, они разделяют значения i-й координаты на диапазоны

$$-\infty = g_{i0} < g_{i1} < \cdots < g_{ir_i} = +\infty \tag{1}$$

и положение x определяется индексами $(j_1, \ldots, j_k)$, такими, что

$$0 \le j_i < r_i, \qquad g_{ij_i} \le x_i < g_{i(j_i+1)} \qquad \text{для } 1 \le i \le k. \tag{2}$$

Совокупности точек, имеющих данное значение $(j_1, \ldots, j_k)$, называются *ячейками*. Записи для точек в одной и той же ячейке хранятся в одном блоке внешней памяти. Блоки могут также содержать точки из нескольких смежных ячеек, обеспечивая тем самым соответствие каждого блока k-мерному прямоугольному региону, или "суперячейке". Возможно использование различных стратегий для обновления значений границ сетки g_{ij} и для разделения и комбинирования блоков (см., например, K. Hinrichs, *BIT* **25** (1985), 569–592). Характеристики сеточных файлов со случайными данными проанализированы в работах M. Regnier, *BIT* **25** (1985), 335–357; P. Flajolet and C. Puech, *JACM* **33** (1986), 371–407, §4.2.

При простом способе работы с ортогональными запросами диапазонов, предложенном Дж. Л. Бентли (J. L. Bentley) и Р. А. Финкелем (R. A. Finkel), используются структуры, которые называются *четревьями** (*quadtrees*) [*Acta Informatica* **4** (1974), 1–9]. В двумерном случае каждый узел такого дерева представляет прямоугольник и содержит одну из точек в этом прямоугольнике. Имеется четыре поддерева, соответствующих четырем квадрантам исходного прямоугольника относительно координат данной точки. Аналогично для трех измерений существует восьмипутевое ветвление, и деревья соответственно называются *восьмиревьями* (*octrees*). k-мерные четревья, естественно, порождаются ветвлением по 2^k путям.

Математический анализ случайных четревьев является весьма сложным, однако в 1988 году независимо двумя группами исследователей (L. Devroye and L. Laforest,

* Переводя такие термины, поневоле чувствуешь себя Алисой в Зазеркалье, слушающей объяснение Шалтая-Болтая о словах, похожих на бумажник: открываешь его, а там два отделения. Надеюсь, читатель простит не слишком научно звучащие, зато оживляющие сухой текст термины. — *Прим. перев.*

SICOMP **19** (1990), 821–832; P. Flajolet, G. Gonnet, C. Puech, and J. M. Robson, *Algorithmica* **10** (1993), 473–500) была определена асимптотическая форма ожидаемого времени вставки N-го узла в случайное k-мерное четрево:

$$\frac{2}{k}\ln N + O(1). \tag{3}$$

Обратите внимание, что при $k = 1$ этот результат согласуется с хорошо известной формулой для вставки в бинарное дерево поиска 6.2.2–(5). Дальнейшие разработки Ф. Флажоле (P. Flajolet), Ж. Лабелля (G. Labelle), Л. Лафоре (L. Laforest) и Б. Салви (B. Salvy) показали, что в действительности средняя внутренняя длина пути может быть выражена в удивительно элегантной форме:

$$\sum_{l\ge 2}\binom{N}{l}(-1)^l\prod_{j=3}^{l}\Bigl(1-\frac{2^k}{j^k}\Bigr). \tag{4}$$

Таким образом, дальнейший анализ случайных четревьев возможен при помощи гипергеометрических функций [см. *Random Structures and Algorithms* **7** (1995), 117–144].

Бентли (Bentley) еще больше упростил представления четревьев, введя "k-d-деревья" (k-d trees), которые имеют только двойное ветвление в каждом узле [*CACM* **18** (1975), 509–517; *IEEE Transactions* **SE-5** (1979), 333–340]. 1-d-дерево представляет собой обычное бинарное дерево поиска, рассмотренное в разделе 6.2.2. 2-d-дерево подобно ему, но при ветвлении на четных уровнях сравниваются координаты x, а на нечетных — y. В общем случае k-d-дерево имеет узлы с k координатами и ветвление на каждом уровне базируется на сравнении одной из координат. Например, на уровне l происходит ветвление по координате $(k \bmod l) + 1$. Для гарантии того, что никакие две записи не будут иметь никаких совпадающих координат, можно использовать правило разрыва узлов (tie-breaking), основываясь на номерах записей или их положении в памяти компьютера. Случайно растущее k-d-дерево будет иметь то же значение средней длины пути и то же распределение ветвей, что и обычное бинарное дерево поиска, потому что предположения, лежащие в основе их роста, те же, что и в одномерном случае (см. упр. 6.2.2–6).

Если файл не изменяется динамически, можно сбалансировать любое k-d-дерево с N узлами так, чтобы его высота составляла $\approx \lg N$, выбрав среднее значение для ветвления в каждом узле. После этого можно быть уверенным в эффективности обработки запросов различных фундаментальных типов. Например, Бентли доказал, что можно найти все записи, имеющие t определенных координат, за $O(N^{1-t/k})$ шагов. Кроме того, можно найти все записи, лежащие в заданной прямоугольной области, не более чем за $O(tN^{1-1/k} + q)$ шагов, если всего имеется q таких записей, а t координат лежат в некоторых подобластях [D. T. Lee and C. K. Wong, *Acta Informatica* **23** (1977), 23–29]. В действительности, если данная область близка к кубической и q мало́ и если выбранные для ветвления координаты в каждом узле имеют наибольший разброс значений атрибутов, то, как показано в работе Friedman, Bentley, and Finkel, *ACM Trans. Math. Software* **3** (1977), 209–226, среднее время обработки запроса к такой области будет составлять всего лишь $O(\log N + q)$. Эта же формула применима при поиске в таком k-d-дереве ближайших соседей данной точки в k-мерном пространстве.

При использовании случайных, а не идеально сбалансированных k-d-деревьев среднее время работы для частичных совпадений по t определенным координатам несколько увеличивается до $\Theta(N^{1-t/k+f(t/k)})$. Функция f неявно определяется уравнением

$$\big(f(x)+3-x\big)^x\,\big(f(x)+2-x\big)^{1-x}=2 \tag{5}$$

и весьма мала: имеем

$$0\le f(x)<0.06329\,33881\,23738\,85718\,14011\,27797\,33590\,58170-. \tag{6}$$

При этом максимум достигается при x, близком к 0.585371 [см. P. Flajolet and C. Puech, *JACM* **33** (1986), 371–407, §3].

Рост популярности геометрических алгоритмов (и их эстетическая привлекательность) вызвали ускоренное развитие технологий решения многомерных задач и смежных вопросов разных видов. Фактически в 70-х годах появилось новое направление в математике и информатике, именуемое *вычислительной геометрией*. Отличным справочным пособием, в котором подробно излагается текущее состояние дел в этой отрасли знаний, является *Handbook of Discrete and Computational Geometry* под редакцией J. E. Goodman and J. O'Rourke (Boca Raton, Florida: CRC Press, 1997).

Всесторонний обзор структур данных и алгоритмов для важных случаев двух- и трехмерных объектов можно найти в двух взаимодополняющих книгах Ханана Самета (Hanan Samet) *The Design and Analysis of Spatial Data Structures* и *Applications of Spatial Data Structures* (Addison-Wesley, 1990). Самет обратил внимание на то, что исходные четревья Бентли и Финкеля более корректно было бы именовать "точечными четревьями" (point quadtrees). Название *четревья* теперь стало общим термином для любой иерархической декомпозиции геометрических данных.

Составные атрибуты. Два или более атрибутов могут быть скомбинированы в один суператрибут. Например, атрибут "(КУРС, СПЕЦИАЛИЗАЦИЯ)" может быть создан путем комбинирования полей КУРС и СПЕЦИАЛИЗАЦИЯ в университетском файле регистрации. Таким образом, запрос зачастую можно выполнить, объединяя короткие списки вместо пересечения длинных.

Идея комбинирования атрибутов была развита В. Ю. Лумом (V. Y. Lum) [*CACM* **13** (1970), 660–665], который предложил упорядочение инвертированных списков комбинированных атрибутов в лексикографическом порядке слева направо и создание нескольких копий с перестановкой индивидуальных атрибутов надлежащим способом. Например, предположим, что имеется три атрибута — A, B и C. Можно сформировать составные атрибуты

$$(\mathtt{A},\mathtt{B},\mathtt{C}),\qquad(\mathtt{B},\mathtt{C},\mathtt{A}),\qquad(\mathtt{C},\mathtt{A},\mathtt{B}) \tag{7}$$

и построить упорядоченные инвертированные списки для каждого из них. (Так, в первом списке записи упорядочены по их значениям A; записи с одинаковыми значениями A упорядочены по значениям B, а затем — по C.) Такая организация позволяет выполнять запросы, основанные на комбинации этих трех атрибутов; например, все записи, имеющие определенные значения A и C, будут располагаться в третьем списке последовательно.

Аналогично из атрибутов A, B, C и D можно сформировать шесть составных атрибутов:

$$(\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{D}),\ (\mathtt{B},\mathtt{C},\mathtt{D},\mathtt{A}),\ (\mathtt{B},\mathtt{D},\mathtt{A},\mathtt{C}),\ (\mathtt{C},\mathtt{A},\mathtt{D},\mathtt{B}),\ (\mathtt{C},\mathtt{D},\mathtt{A},\mathtt{B}),\ (\mathtt{D},\mathtt{A},\mathtt{B},\mathtt{C}). \tag{8}$$

Они позволяют выполнять все комбинации простых запросов с фиксированными значениями одного, двух, трех или четырех атрибутов. Существует общая процедура построения $\binom{n}{k}$ комбинированных атрибутов из n отдельных атрибутов при $k \le \frac{1}{2}n$, такая, что все записи, имеющие определенные комбинации не более чем из k или не менее $n-k$ значений атрибутов, будут последовательно расположены в одном из списков комбинированных атрибутов (см. упр. 1). Можно обойтись и меньшим количеством комбинаций, если некоторые атрибуты имеют ограниченное множество значений. Например, если D представляет собой атрибут с двумя возможными значениями, то комбинации

$$(\mathtt{D},\mathtt{A},\mathtt{B},\mathtt{C}),\quad (\mathtt{D},\mathtt{B},\mathtt{C},\mathtt{A}),\quad (\mathtt{D},\mathtt{C},\mathtt{A},\mathtt{B}), \tag{9}$$

полученные в результате помещения D в (7), будут так же хороши, как и (8), с половинной избыточностью, поскольку запросы, не зависящие от D, могут быть обработаны путем просмотра в двух местах одного из списков.

Бинарные атрибуты. Поучительно рассмотреть частный случай, когда все атрибуты могут иметь только два значения. По сути, это *противоположность* комбинированных атрибутов, поскольку любое значение можно представить как двоичное число и рассматривать индивидуальные биты этого числа как отдельные атрибуты. В табл. 1 показан типичный файл с атрибутами "Да"–"Нет". В этом примере записи содержат рецепты домашнего печенья, а атрибуты определяют используемые ингредиенты. Например, миндальные вафли с ромом сделаны из масла, муки, молока, орехов и сахарного песка. Если рассматривать табл. 1 как матрицу из нулей и единиц, то транспонированная матрица будет представлять собой инвертированный файл в виде битовых строк.

Правый столбец табл. 1 используется для указания специальных, редко используемых продуктов. Они могут быть закодированы более эффективно, без отдельного столбца для каждого из них. То же самое справедливо для столбца "Кукурузный крахмал". Кроме того, можно найти более эффективный путь кодирования столбца "Мука", поскольку мука встречается во всех рецептах, кроме рецепта приготовления меренги. Однако пока оставим этот вопрос и просто проигнорируем столбец "Специальные ингредиенты".

Будем определять *базовый запрос* к файлу бинарных атрибутов как запрос на все записи, в одних столбцах которых содержатся нули, в других — единицы и в остальных столбцах — произвольные величины. Используя символ "*" для обозначения любого значения, базовый запрос можно представить в виде последовательностей символов 0, 1 и "*". Например, представим человека, который захотел печенья с кокосом и у которого аллергия на шоколад, который ненавидит анис и у которого дома закончился ванилин. Тогда его запрос может быть сформулирован так:

$$*\,0\,*\,*\,*\,*\,0\,*\,*\,1\,*\,*\,*\,*\,*\,*\,*\,*\,*\,*\,*\,*\,*\,*\,*\,*\,*\,0. \tag{10}$$

Из табл. 1 становится понятно, что его желания совпадают с его возможностями в случае ароматных палочек с черносливом.

Таблица 1

ФАЙЛ С БИНАРНЫМИ АТРИБУТАМИ

	Душистый перец	Зерна аниса	Пекарный порошок	Пищевая сода	Масло	Кардамон	Шоколад	Корица	Гвоздика	Кокосовый орех	Кофе	Кукурузный крахмал	Финики	Яичный белок	Яичный желток	Мука	Имбирь	Лимонный сок	Лимонная цедра	Молоко	Патока	Мускатный орех	Орехи	Толокно	Изюм	Соль	Сахар, жженый	Сахар, песок	Сахар, пудра	Ванилин	Специальные ингредиенты
Миндальные вафли с ромом	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	1	0	0	—
Печенье с яблочным соусом	0	0	0	1	1	0	0	1	1	0	0	0	0	1	1	1	0	0	0	0	0	1	1	0	1	0	0	1	1	1	Яблочный соус
Банановo-овсяное печенье	0	0	0	1	1	0	0	1	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	0	1	0	1	0	1	Бананы
Шоколадный хворост	0	0	0	1	1	0	1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	0	1	—
Миндальное печенье с кокосами	0	0	1	0	1	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	0	1	—
Печенье со сливочным сыром	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	Сливочный сыр
Ароматные палочки с черносливом	0	0	0	0	1	0	0	0	0	1	0	1	0	1	1	1	0	0	0	0	0	0	1	0	0	1	1	0	0	0	Апельсины, чернослив
Драже в шоколаде	0	0	0	1	1	0	1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	0	0	—
Райские палочки	0	0	1	0	1	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	1	1	0	0	1	—
Пирог с начинкой	0	0	1	0	1	0	0	0	0	0	0	0	1	1	1	1	0	1	1	0	0	0	1	0	0	1	0	1	0	1	—
Финский какор	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	0	0	Экстракт миндаля
Глазированные имбирные пряники	0	0	1	1	1	0	0	1	1	0	0	0	0	1	1	1	1	0	0	1	1	0	0	0	0	0	1	0	1	0	Уксус
Печенье с орехами	0	0	0	1	1	0	0	1	0	0	1	0	0	1	1	1	0	0	0	0	0	1	0	0	1	1	1	0	0	0	Абрикос
Драгоценное печенье	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	0	1	0	0	1	Смородиновое желе
Перепутаница	0	0	0	0	1	0	0	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	1	0	1	Растительное масло
Малайский крендель	0	0	0	0	1	1	0	0	0	0	0	0	0	1	1	1	0	0	1	0	0	0	1	0	0	0	0	1	0	0	—
Медовые пряники	1	0	0	1	0	0	0	1	1	0	0	0	0	1	1	1	0	1	1	1	1	0	1	1	0	0	1	1	0	1	0	Мед
Меренги	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	Засахаренная вишня
Моравское печенье со специями	1	0	0	1	1	0	0	1	1	0	0	0	0	1	0	1	1	0	0	0	1	1	0	0	0	1	1	0	1	0	—
Овсяные палочки с финиками	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	1	1	0	1	1	1	0	0	—
Старинное сахарное печенье	0	0	1	1	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	1	0	0	0	1	0	1	0	1	Сметана
Печенье с арахисовым маслом	0	0	0	1	1	0	1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	1	Арахисовое масло
Юбочки	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	—
Печенье с перцем	1	1	1	0	0	1	0	1	1	0	1	0	0	1	1	1	0	0	1	0	0	1	1	0	0	1	1	0	1	0	Цукаты, перец, мускат
Шотландские овсяные коржики	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1	0	1	0	1	—
Песочные звездочки	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	—
Анисовое печенье	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	1	0	1	1	0	—
Воздушное печенье	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	1	0	1	—
Шведский крендель	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	—
Швейцарское рассыпчатое печенье с корицей	0	0	0	0	1	0	0	1	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	1	1	1	0	0	—
Ириски	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0	1	0	0	1	—
Ванильно-ореховое мороженое	0	0	1	0	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	0	1	—

McCall's Cook Book (New York: Random House, 1963), Chapter 9.

Прежде чем приступить к организации файла для базовых запросов, рассмотрим один важный специальный момент, когда в запросе отсутствуют нули, а есть только 1 и "*". Такой запрос можно назвать *включающим* (*inclusive query*), поскольку он запрашивает записи, которые *включают* некоторое множество атрибутов (если предположить, что 1 означает наличие определенного атрибута, а 0 — его отсутствие). Например, в табл. 1 два рецепта одновременно содержат и пекарный порошок, и пищевую соду — глазированные имбирные пряники и старинное сахарное печенье.

В некоторых приложениях достаточно использовать специальные включающие запросы, например в ручных карточных системах наподобие карт с перфорацией по краям или карт свойств. Система карт с перфорацией по краям для табл. 1 будет иметь по одной карточке на каждый рецепт с вырезами, соответствующими каждому ингредиенту (рис. 46). Обработка включающего запроса сводится к складыванию карточек аккуратной стопкой и введению спиц в положениях, соответствующих конкретным ингредиентам. После ввода спиц все карточки с определенными атрибутами выпадают из стопки.

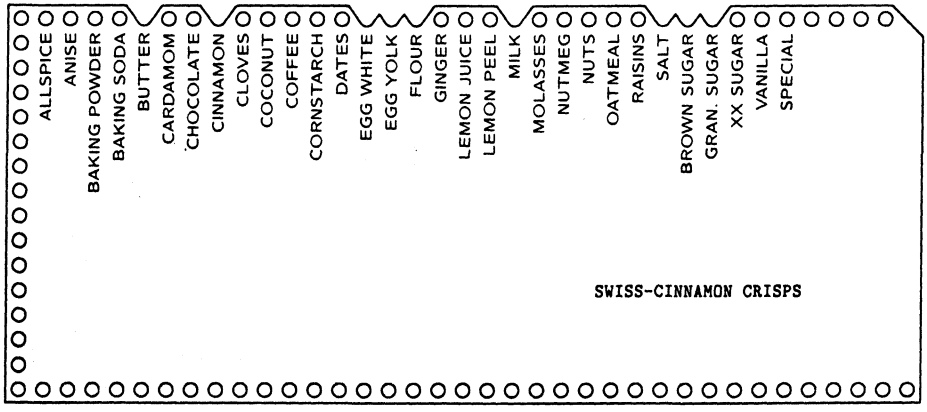


Рис. 46. Карта с перфорацией по краям.

Система, основанная на картах свойств, работает с инвертированным файлом аналогичным образом: имеется по одной карте для каждого атрибута и отверстия в карте пробиваются в позициях, которые соответствуют записям, содержащим этот атрибут. Таким образом, одна стандартная перфокарта шириной 80 позиций может использоваться для хранения информации о том, какие из $12 \times 80 = 960$ записей имеют данный атрибут. Для обработки включающего запроса отбираются карты свойств для определенных атрибутов и накладываются одна на другую. Луч света, проходя через позиции, в которых на всех картах имеются отверстия, покажет искомый результат. Это напоминает обработку логического запроса путем пересечения инвертированных битовых строк, как объяснялось выше.

Кодирование методом наложения. Причина, по которой нас (вооруженных современными компьютерами) интересуют методы поиска по карточкам вручную, заключается в том, что в свое время было разработано много хитроумных способов

хранения места на картах с перфорацией по краям, принципы которых применимы и для представления компьютерных файлов. Кодирование методом наложения представляет собой технологию, схожую с хешированием, хотя оно было разработано за несколько лет до изобретения хеширования. Идея заключается в отображении атрибутов в случайные k-битовые коды в n-битовых полях и наложении кодов каждого атрибута, имеющегося в записи. Включающий запрос для некоторого множества атрибутов может быть конвертирован во включающий запрос для соответствующих наложенных битовых кодов. Такому запросу могут удовлетворять несколько дополнительных записей, но количество подобных "ложных выпадений" может быть статистически учтено [см. Calvin N. Mooers, *Amer. Chem. Soc. Meeting* **112** (September, 1947), 14E–15E; *American Documentation* **2** (1951), 20–32].

В качестве примера кодирования методом наложения вновь обратимся к табл. 1, но только к той ее части, которая относится к специям, не рассматривая основные компоненты наподобие пекарного порошка, яиц и муки. В табл. 2 показано, что произойдет, если назначить случайные двухбитовые коды в десятибитовых полях каждой из специй и использовать наложение. Например, "Шоколадный хворост" будет получен в результате наложения кодов шоколада, орехов и ванилина:

$$0010001000 \vee 0000100100 \vee 0000001001 = 0010101101.$$

Наложение данных кодов даст также несколько ложных атрибутов; в нашем конкретном случае — душистый перец, засахаренную вишню, смородиновое желе, арахисовое масло и перец. Это вызовет ложные выпадения при некоторых запросах (тем самым будет предложено разработать новый рецепт — "Ложновыпадающее печенье"! :-)).

Для табл. 2 кодирование методом наложения работает не так уж хорошо, поскольку эта таблица представляет всего лишь маленький пример с большим количеством атрибутов. В самом деле, "Печенье с яблочным соусом" будет выпадать при *каждом* запросе, поскольку оно получено наложением семи кодов, покрывающих все десять позиций. Еще хуже обстоят дела в случае рецепта печенья с перцем (впрочем, с точки зрения кодов и ложных выпадений совершенно все равно, как получена цепочка из десяти единиц: наложением семи или двенадцати кодов. — *Прим. перев.*), полученного в результате наложения двенадцати кодов. С другой стороны, иногда табл. 2 работает на удивление неплохо; например, дав запрос "Ванилин", мы получим только одно ложное выпадение — "Печенье с перцем".

Более подходящий пример кодирования методом наложения получается при наличии, скажем, 32-битового поля и набора из $\binom{32}{3} = 4\,960$ различных атрибутов, где каждая запись может иметь до шести атрибутов и каждый атрибут кодируется тремя из 32 бит. В этой ситуации в предположении, что каждая запись имеет шесть случайно выбранных атрибутов, вероятности ложных выпадений в случае включающего запроса составляют:

по одному атрибуту	0.07948358
по двум атрибутам	0.00708659
по трем атрибутам	0.00067094
по четырем атрибутам	0.00006786
по пяти атрибутам	0.00000728
по шести атрибутам	0.00000082

(11)

Таблица 2

ПРИМЕР КОДИРОВАНИЯ МЕТОДОМ НАЛОЖЕНИЯ

Коды отдельных приправ

Экстракт миндаля	0100000001	Финики	1000000100
Душистый перец	0000100001	Имбирь	0000110000
Зерна аниса	0000011000	Мед	0000000011
Яблочный соус	0010010000	Лимонный сок	1000100000
Абрикос	1000010000	Лимонная цедра	0011000000
Бананы	0000100010	Мускатный цвет	0000010100
Засахаренная вишня	0000101000	Патока	1001000000
Кардамон	1000000001	Мускатный орех	0000010010
Шоколад	0010001000	Орехи	0000100100
Корица	1000000010	Апельсины	0100000100
Цукаты	0100000010	Арахисовое масло	0000000101
Гвоздика	0001100000	Перец	0010000100
Кокосовый орех	0001010000	Чернослив	0010000010
Кофе	0001000100	Изюм	0101000000
Смородиновое желе	0010000001	Ванилин	0000001001

Наложенные коды

Миндальные вафли с ромом	0000100100	Медовые пряники	1011110111
Печенье с яблочным соусом	1111111111	Меренги	1000101100
Банановo-овсяное печенье	1000111111	Моравское печенье со специями	1001110011
Шоколадный хворост	0010101101	Овсяные палочки с финиками	1000100100
Миндальное печенье с кокосами	0001111101	Старинное сахарное печенье	0000011011
Печенье со сливочным сыром	0010001001	Печенье с арахисовым маслом	0010001101
Ароматные палочки с черносливом	0111110110	Юбочки	0000001001
Драже в шоколаде	0010101100	Печенье с перцем	1111111111
Райские палочки	0001111101	Шотландские овсяные коржики	0000001001
Пирог с начинкой	1011101101	Песочные звездочки	0000000000
Финский какор	0100100101	Анисовое печенье	0011011000
Глазированные имбирные пряники	1001110010	Воздушное печенье	0000001001
Печенье с орехами	1101010110	Шведский крендель	0000000000
Драгоценное печенье	0010101101	Швейцарское рассыпчатое печенье с корицей	1000000010
Перепутаница	1000001011	Ириски	0010101101
Малайский крендель	1011100101	Ванильно-ореховое мороженое	0000101101

Таким образом, если имеется M записей, которые не удовлетворяют двухатрибутному запросу, то около $0.007M$ записей будут иметь наложенные коды, соответствующие всем битам этих двух атрибутов (все вероятности вычисляются в упр. 4). Общее количество битов, необходимых для представления инвертированного файла, составляет 32, умноженное на количество записей, что меньше половины количества битов, требуемых для описания всех атрибутов в исходном файле.

При аккуратном использовании неслучайных кодов можно избежать ложных выпадений, как показано в работе W. H. Kautz and R. C. Singleton, *IEEE Trans.* **IT-10** (1964), 363–377; одна из таких конструкций рассматривается в упр. 16.

Малкольм Ч. Харрисон (Malcolm C. Harrison) [*CACM* **14** (1971), 777–779] обнаружил, что кодирование методом наложения может использоваться для ускорения *поиска текста*. Пусть нужно найти все вхождения некоторой строки символов в тексте большого объема без построения обширных таблиц алгоритма 6.3Р. Предположим также, что текст разделен на строки, например, по 50 символов: $c_1c_2\ldots c_{50}$. Харрисон предложил кодировать каждую из 49 пар c_1c_2, c_2c_3, ..., $c_{49}c_{50}$ методом хеширования каждой из них в число, скажем, от 0 до 127, а затем подсчитать "ключ" строки $c_1c_2\ldots c_{50}$, который представляет собой строку из 128 бит $b_0b_1\ldots b_{127}$, где $b_i = 1$ тогда и только тогда, когда $h(c_jc_{j+1}) = i$ для некоторого j.

Пусть необходимо найти все вхождения слова NEEDLE (ИГОЛКА) в большом файле с именем HAYSTACK (СТОГ СЕНА). При этом мы просто просмотрим все строки, ключи которых содержат 1 в позициях h(NE), h(EE), h(ED), h(DL) и h(LE). Считая хеш-функцию случайной, вероятность того, что случайная строка содержит все эти биты в своем ключе, можно оценить как 0.00341 (см. упр. 4). Следовательно, пересечение пяти инвертированных списков битовых строк позволит быстро обнаружить все строки, содержащие NEEDLE (естественно, с некоторым количеством ложных выпадений).

Предположение случайности на самом деле мало применимо для текста, поскольку обычный текст весьма избыточен и пары символов в нем распределены весьма неравномерно. Например, может оказаться полезным опустить все пары c_jc_{j+1}, содержащие символ пробела, в связи с тем, что пробел встречается в тексте гораздо чаще прочих символов.

Другое интересное приложение кодирования методом наложения к задачам поиска было предложено Бартоном Блюмом (Burton Bloom) [*CACM* **13** (1970), 422–426]. Его метод на самом деле применим для выборки по *первичным* ключам, хотя более подходящее место для его обсуждения — этот раздел. Представьте себе приложение для поиска в большой базе данных, которое не требует каких-либо дальнейших действий, если поиск оказался неудачным. Например, нужно просто проверить чью-то кредитную карточку или номер паспорта и, если в файле нет записей о данной персоне, никаких дальнейших действий предпринимать не требуется. Аналогичный метод применим и при компьютерной верстке для правильных переносов слов. Например, у нас есть метод расстановки переносов, корректно работающий с большинством слов, но имеющий некоторое количество исключений на 50 000 слов; если слово не найдено в файле исключений, можно использовать простой алгоритм.

В такой ситуации можно хранить таблицу битов во внутренней памяти, чтобы отсутствие большинства ключей распознавалось *без* обращений к внешней памяти. Вот как это можно сделать: обозначим внутреннюю битовую таблицу через $b_0b_1\ldots b_{M-1}$, где M очень велико. Для каждого ключа K_j в файле вычислим k независимых хеш-функций $h_1(K_j),\ldots,h_k(K_j)$ и установим соответствующие k бит равными 1 (эти k значений не обязаны быть различными). Таким образом, $b_i = 1$ тогда и только тогда, когда $h_l(K_j) = i$ для некоторых j и l. Теперь, чтобы определить наличие аргумента поиска K во внешнем файле, сначала выясним, справедливо ли соотношение $b_{h_l(K)} = 1$ при $1 \le l \le k$. Если нет, то нет и необходимости обращаться к

внешней памяти, но если соотношение справедливо, то при корректном выборе k и M последовательный поиск, вероятно, найдет K. Вероятность ложного выпадения при N записях в файле равна примерно $(1-e^{-kN/M})^k$. По сути, метод Блума рассматривает весь файл как одну запись, в которой первичные ключи представлены как атрибуты, а кодирование методом наложения производится в огромном M-битовом поле.

Еще один вариант кодирования методом наложения был разработан Ричардом А. Густафсоном (Richard A. Gustafson) [Ph. D. thesis (Univ. South Carolina, 1969)]. Предположим, что есть N записей и что каждая из них имеет шесть атрибутов, выбранных из 10 000 возможных. Запись может представлять, например, техническую статью, а атрибуты — ключевые слова, описывающие эту статью. Пусть h — это хеш-функция, отображающая каждый атрибут в число между 0 и 15. Запись с атрибутами $a_1, a_2, \ldots, a_6$ Густафсон предлагает отображать в 16-битовое число $b_0 b_1 \ldots b_{15}$, где $b_i = 1$ тогда и только тогда, когда $h(a_j) = i$ для некоторого j. И далее, если только $k < 6$ бит b равны 1, то другие $6 - k$ единиц добавляются некоторым случайным методом (необязательно зависящим от самой записи). Дано $\binom{16}{6} = 8\,008$ 16-битовых кодов, в которых имеется ровно шесть единичных битов, и при определенной доле везения примерно $N/8008$ записей будут отображены на каждое значение. Можно хранить 8 008 списков записей, непосредственно вычисляя адрес, соответствующий $b_0 b_1 \ldots b_{15}$, с использованием подходящей формулы. В самом деле, если единицы встречаются в позициях $0 \le p_1 < p_2 < \cdots < p_6$, то функция

$$\binom{p_1}{1} + \binom{p_2}{2} + \cdots + \binom{p_6}{6}$$

конвертирует каждую строку $b_0 b_1 \ldots b_{15}$ в единственное число между 0 и 8087, как будет показано в упр. 1.2.6–56 и 2.2.6–7.

Теперь, чтобы найти все записи, имеющие три определенных атрибута A_1, A_2 и A_3, вычислим $h(A_1)$, $h(A_2)$, $h(A_3)$; предполагая, что все эти значения различны, нужно будет просмотреть только записи, хранящиеся в $\binom{13}{3} = 286$ списках, битовые коды $b_0 b_1 \ldots b_{15}$ которых содержат единицы в соответствующих трех позициях. Другими словами, только $286/8008 \approx 3.5\%$ записей должны будут проверяться при поиске.

Превосходное описание кодирования методом наложения вместе с приложением для работы с большой базой данных телефонных номеров можно найти в статье C. S. Roberts, *Proc. IEEE/* **67** (1979), 1624–1642. Приложение метода к программному обеспечению проверки орфографии обсуждается в работе J. K. Mullin and D. J. Margoliash, *Software Practice & Exper.* **20** (1990), 625–630.

Комбинаторное хеширование. Идея, лежащая в основе только что описанного метода Густафсона, заключается в поиске некоторого пути отображения записей на адреса в памяти, такого, что к определенному запросу имеет отношение лишь сравнительно небольшое количество адресов памяти. Однако этот метод применим только к включающим запросам в том случае, когда отдельные записи обладают небольшим количеством атрибутов. Другой тип отображения, предназначенный для обработки произвольных базовых запросов типа (10), в которых содержатся нули, единицы и звездочки, был разработан в 1971 году Рональдом Л. Ривестом [см. *SICOMP* **5** (1976), 19–50].

Предположим сначала, что необходимо создать словарь для составления кроссвордов, содержащий все шестибуквенные слова английского языка; типичный запрос ищет все слова вида, например, N**D*E и получает ответ {NEEDLE, NIDDLE, NODDLE, NOODLE, NUDDLE}. Можно решить эту задачу при помощи 2^{12} списков, поместив слово NEEDLE в список номер

$$h(\mathtt{N})\, h(\mathtt{E})\, h(\mathtt{E})\, h(\mathtt{D})\, h(\mathtt{L})\, h(\mathtt{E}).$$

Здесь h — хеш-функция, переводящая каждую букву в двухбитовое значение, и мы получаем 12-битовый адрес, записывая рядом шесть пар битов. Затем запрос N**D*E может быть обработан в результате просмотра только 64 списков из 4 096.

Аналогично предположим, что есть 1 000 000 записей, в каждой из которых содержится 10 вторичных ключей, а каждый вторичный ключ имеет большое количество возможных значений. Можно отобразить записи со вторичными ключами $(K_1, K_2, \ldots, K_{10})$ в 20-битовое число

$$h(K_1)\, h(K_2)\, \ldots\, h(K_{10}), \tag{12}$$

где h — хеш-функция, преобразующая каждый вторичный ключ в двухбитовое число, и (12) образуется после размещения этих десяти пар битов в одном 20-битовом числе. Данная схема отображает 1 000 000 записей в $2^{20} = 1\,048\,576$ возможных значений, и отображение, в целом, может рассматриваться как хеш-функция с $M = 2^{20}$. Для разрешения коллизий может использоваться метод цепочек. Чтобы получить все записи с определенными значениями пяти вторичных ключей, потребуется просмотреть только 2^{10} списков, соответствующих пяти неопределенным парам битов в (12). Таким образом, в среднем придется проверить около $1000 = \sqrt{N}$ записей. (Подобный подход был предложен М. Арисавой (M. Arisawa) [*J. Inf. Proc. Soc. Japan* **12** (1971), 163–167] и Б. Двайером (B. Dwyer) [не опубликовано]. Двайер предложил использовать более гибкое по сравнению с (12) отображение, а именно

$$\big(h_1(K_1) + h_2(K_2) + \cdots + h_{10}(K_{10})\big) \bmod M,$$

где M — любое подходящее число, а h_i — произвольные хеш-функции, возможно, вида $w_i K_i$ для "случайного" w_i.)

Ривест продолжил разработку этой идеи, так что во многих случаях возникает следующая ситуация. Предположим, что имеется $N \approx 2^n$ записей, в каждой из которых содержится m вторичных ключей. Каждая запись отображается в n-битовый хеш-адрес таким образом, что запрос, оставляющий неопределенными значения k ключей, соответствует примерно $N^{k/m}$ хеш-адресам. Все другие методы, обсуждавшиеся ранее в этом разделе (за исключением метода Густафсона), требуют порядка N шагов для получения информации, хотя и с малым коэффициентом пропорциональности; для больших значений N метод Ривеста более быстр и не требует инвертированных файлов.

Но прежде чем применять эту технологию, следует определить подходящее отображение. Ниже рассматривается пример с небольшими параметрами, с $m = 4$ и $n = 3$ и вторичными ключами, принимающими два значения. Можно отобразить 4-битовые записи в восемь адресов следующим образом.

$$
\begin{array}{llll}
*\,0\,0\,1 \to 0 & & *\,1\,1\,0 \to 4 & \\
0\,*\,0\,0 \to 1 & & 1\,*\,1\,1 \to 5 & \\
1\,0\,*\,0 \to 2 & & 0\,1\,*\,1 \to 6 & \\
1\,1\,0\,* \to 3 & & 0\,0\,1\,* \to 7 &
\end{array}
\qquad (13)
$$

Исследование этой таблицы показывает, что все записи, соответствующие запросу $0\ *\ *\ *$, отображаются в позиции 0, 1, 4, 6 и 7. Точно так же *любой* базовый запрос с тремя символами "*" соответствует в точности пяти позициям, базовый запрос с двумя "*" — трем позициям и с одной "*" — одной или двум позициям, $(8 \times 1 + 24 \times 2)/32 = 1.75$ в среднем. Таким образом, имеем следующее.

Количество неопределенных битов в запросе	Количество позиций поиска	(14)
4	$8 = 8^{4/4}$	
3	$5 \approx 8^{3/4}$	
2	$3 \approx 8^{2/4}$	
1	$1.75 \approx 8^{1/4}$	
0	$1 = 8^{0/4}$	

Конечно, это всего лишь маленький пример, и в таких случаях гораздо проще осуществлять поиск "в лоб". Этот метод приводит к появлению нетривиальных приложений, поскольку его можно использовать и тогда, когда $m = 4r$ и $n = 3r$, отображая $4r$-битовые записи на $2^{3r} \approx N$ позиций путем разделения вторичных ключей на r групп по 4 бит и применяя (13) к каждой группе. Получающееся отображение имеет требуемое свойство: *запрос, оставляющий k из m бит неопределенными, соответствует примерно $N^{k/m}$ позициям* (см. упр. 6).

В 1997 году А. Э. Брувер (A. E. Brouwer) нашел привлекательный способ сжатия 8 бит до 5 бит при помощи отображения, аналогичного (13). Каждый 8-битовый байт принадлежит в точности одному из следующих 32 классов.

$$
\begin{array}{llll}
0{*}000{*}0{*} & 01{*}0{**}11 & 00{*}11{**}1 & {*}11{**}101 \\
1{*}000{*}0{*} & 11{*}0{**}11 & 10{*}11{**}1 & {*}11{**}010 \\
0{*}010{*}0{*} & 01{*}1{**}11 & 00{*}0{*}01{*} & {*}10{*}0{*}10 \\
1{*}010{*}0{*} & 11{*}1{**}11 & 10{*}0{*}01{*} & {*}10{*}1{*}01 \\
0{*}10{*}1{*}0 & 0{*}1{*}000{*} & {*}01{*}01{*}1 & {*}0{*}1001{*} \\
1{*}10{*}1{*}0 & 1{*}1{*}000{*} & {*}10{*}10{*}0 & {*}0{*}0100{*} \\
0{*}11{*}1{*}0 & 0{*}0{*}11{*}0 & {*}00{*}011{*} & {*}0{*}011{*}1 \\
1{*}11{*}1{*}0 & 1{*}0{*}11{*}0 & {*}11{*}100{*} & {*}0{*}110{*}0
\end{array}
\qquad (15)
$$

Звездочки в этой конструкции расположены таким образом, что в каждой строке их по 3, а в каждом столбце — по 12. В упр. 18 поясняется, каким образом строятся подобные схемы, сжимающие записи с $m = 4^r$ бит в $n = 3^r$-битовые адреса. На практике используются блоки размером b и $N \approx 2^n b$; случай, когда $b = 1$, использовался выше для упрощения изложения материала.

Ривест предложил также другой простой путь обработки базовых запросов. Предположим, имеется $N \approx 2^{10}$ записей по 30 бит и необходимо ответить на произвольный 30-битовый базовый запрос, подобный (10). В этом случае мы можем просто поделить 30 бит на три 10-битовых поля и поддерживать три отдельные хеш-таблицы размером $M = 2^{10}$. Каждая запись хранится в трех местах, в списках, соответствующих битовым конфигурациям трех полей. При соответствующих усло-

виях в каждом списке будет содержаться около одного элемента. В данном базовом запросе с k неопределенными битами как минимум одно из полей будет иметь $\lfloor k/3 \rfloor$ или меньше неопределенных битов. Следовательно, потребуется просмотреть не более $2^{\lfloor k/3 \rfloor} \approx N^{k/30}$ списков для поиска всех ответов на запрос. Впрочем, можно использовать и любую другую технологию для обработки базовых запросов в выбранных полях.

Обобщенные лучи (tries). Ривест предложил еще один подход, который основан на структурах данных, подобных использованным в разделе 6.3 лучам. Можно предположить, что каждый внутренний узел обобщенного бинарного луча определяет представленные в записи биты. Например, используя приведенные в табл. 1 данные, можно положить корнем луча *Ванилин*. Тогда левый подлуч будет соответствовать тем 16 рецептам, в которых ванилин не нужен, в то время как в правом подлуче собираются 15 рецептов с ванилином. Такое разбиение 16-15 удачно делит файл практически пополам; каждый из подфайлов обрабатывается аналогично, и когда на некоторой стадии подфайл становится достаточно мал, его можно представить в качестве конечного узла.

Для поиска по обобщенному лучу с корнем, определяющим атрибут, которому в запросе соответствует 0 или 1, переходим к поиску в левом или правом подлуче соответственно; если этому атрибуту в запросе соответствует "$*$", просматриваем оба подлуча.

Предположим, что атрибуты не бинарны, но представлены в бинарной записи. Можно построить луч, рассматривая сначала первый бит атрибута 1, затем — первый бит атрибута 2, ..., первый бит атрибута m, второй бит атрибута 1 и т. д. Такая структура называется m-d-лучом, по аналогии с m-d-деревьями (которые ветвятся в результате сравнения, а не проверки битов). Ф. Флажоле (P. Flajolet) и К. Пуч (C. Puech) показали, что среднее время ответа на частично определенный запрос по случайному m-d-лучу с N узлами составляет $\Theta(N^{k/m})$, если k/m атрибутов не определены [*JACM* **33** (1986), 371–407, §4.1]. Дисперсия этой величины была вычислена В. Шахингером (W. Schachinger), *Random Structures and Algorithms* **7** (1995), 81–95.

Подобный алгоритм может быть распространен на m-мерные версии цифровых деревьев поиска и деревья метода "Патриция" из раздела 6.3. Эти структуры, которые обычно несколько лучше сбалансированы, чем m-d-лучи, были проанализированы в работе P. Kirschenhofer and H. Prodinger, *Random Structures and Algorithms* **5** (1994), 123–134.

***Сбалансированные схемы.** Другой комбинаторный подход к получению информации, основанный на *системах сбалансированных неполных блоков*, был предметом множества исследований. Хотя предмет исследования весьма интересен с математической точки зрения, к сожалению, его преимущества перед другими описанными выше методами до сих пор не доказаны. Здесь будет представлено краткое введение в теорию, чтобы показать изящество результатов, в надежде, что читатели придумают, как внедрить теоретические идеи в практику.

Штейнеровская система троек представляет собой распределение v объектов в неупорядоченные тройки таким образом, что каждая пара объектов встречается ровно в одной тройке. Например, при $v = 7$ имеется ровно одна Штейнеровская система троек.

Тройки	Содержащиеся пары
$\{1,2,4\}$	$\{1,2\}$, $\{1,4\}$, $\{2,4\}$
$\{2,3,5\}$	$\{2,3\}$, $\{2,5\}$, $\{3,5\}$
$\{3,4,6\}$	$\{3,4\}$, $\{3,6\}$, $\{4,6\}$
$\{4,5,0\}$	$\{0,4\}$, $\{0,5\}$, $\{4,5\}$
$\{5,6,1\}$	$\{1,5\}$, $\{1,6\}$, $\{5,6\}$
$\{6,0,2\}$	$\{0,2\}$, $\{0,6\}$, $\{2,6\}$
$\{0,1,3\}$	$\{0,1\}$, $\{0,3\}$, $\{1,3\}$

(16)

Поскольку существует $\frac{1}{2}v(v-1)$ пар объектов и три пары в одной тройке, всего должно быть $\frac{1}{6}v(v-1)$ троек; а так как каждый объект может быть в паре с $v-1$ объектами, каждый объект должен содержаться ровно в $\frac{1}{2}(v-1)$ тройках. Из этих условий следует, что Штейнеровская система троек может существовать тогда и только тогда, когда $\frac{1}{6}v(v-1)$ и $\frac{1}{2}(v-1)$ представляют собой целые числа. Это эквивалентно тому, что v должно быть нечетно и не быть равным 2 по модулю 3; таким образом,

$$v \bmod 6 = 1 \text{ или } 3. \tag{17}$$

И обратно, в 1847 году Т. П. Киркман (T. P. Kirkman) доказал, что Штейнеровская система троек существует для всех $v \geq 1$, удовлетворяющих (17). Его интересное построение приведено в упр. 10.

Штейнеровские системы троек могут использоваться для снижения избыточности индексов комбинированных инвертированных файлов. Вернемся, например, к рассматривавшимся ранее рецептам печенья из табл. 1 и конвертируем крайний справа столбец в тридцать первый атрибут, который равен 1, если требуется какой-либо специальный ингредиент, и равен 0 в противном случае. Предположим, что необходимо ответить на все включающие запросы о парах атрибутов наподобие "В каких рецептах используются и кокосовый орех, и изюм?". Можно было бы построить инвертированные файлы для всех $\binom{31}{2} = 465$ возможных запросов. Однако они займут очень много места, так как, например, "Печенье с перцем" будет содержаться в $\binom{17}{2} = 136$ списках, а запись, включающая 31 атрибут, будет содержаться в каждом списке! Штейнеровская система троек позволяет немного улучшить ситуацию. В случае 31 объекта существует Штейнеровская система из 155 троек, в которой каждая пара объектов встречается ровно в одной тройке. Каждой тройке $\{a,b,c\}$ можно назначить четыре списка: первый — для всех записей с атрибутами $\{a,b,\bar{c}\}$ (т. е. a, b, но не c), второй — для $\{a,\bar{b},c\}$, третий — для $\{\bar{a},b,c\}$ и четвертый — для записей со всеми тремя атрибутами $\{a,b,c\}$. Таким образом гарантируется, что никакая запись не будет включена более чем в 155 инвертированных списков, и сохраняется пространство, когда запись имеет три атрибута, соответствующих тройке системы.

Системы троек представляют собой частный случай систем блоков из трех или более объектов. Например, тот же 31 объект может быть распределен по шестеркам так, чтобы каждая пара объектов оказывалась ровно в одной шестерке:

$$\{0,4,16,21,22,24\},\ \{1,5,17,22,23,25\},\ \ldots,\ \{30,3,15,20,21,23\}. \tag{18}$$

(Эта конструкция создана из первого блока путем сложения по модулю 31. Чтобы проверить, что она обладает указанными свойствами, обратим внимание на сле-

дующий факт: 30 значений $(a_i - a_j) \bmod 31, i \neq j$ различны при $(a_1, a_2, \ldots, a_6) = (0, 4, 16, 21, 22, 24)$. Для поиска шестерки, содержащей пару (x, y), выберем i и j такими, что $a_i - a_j \equiv x - y$ (по модулю 31). Теперь, если $k = (x - a_i) \bmod 31$, имеем $(a_i + k) \bmod 31 = x$ и $(a_j + k) \bmod 31 = y$.)

Можно использовать конструкцию, приведенную выше, для хранения инвертированных списков таким образом, чтобы ни одна запись не появилась более 31 раза. Каждая шестерка $\{a, b, c, d, e, f\}$ ассоциирована с 57 списками всех возможных для записей, имеющих два или более атрибутов a, b, c, d, e, f, а именно — $\{a, b, \bar{c}, \bar{d}, \bar{e}, \bar{f}\}$, $\{a, \bar{b}, c, \bar{d}, \bar{e}, \bar{f}\}$, $\ldots$, $\{a, b, c, d, e, f\}$. Ответом на любой включающий запрос по двум атрибутам является объединение без пересечений 16 подходящих списков соответствующей шестерки. Так, "Печенье с перцем" войдет в 29 блоков из 31, поскольку эта запись имеет по два из шести атрибутов во всех шестерках, кроме $\{19, 23, 4, 9, 10, 12\}$ и $\{13, 17, 29, 3, 4, 6\}$, если перенумеровать столбцы от 0 до 30.

Теория построения блоков и связанные с ней вопросы детально рассмотрены в книге Маршалла Холла (мл.) (Marshall Hall, Jr.) *Combinatorial Theory* (Waltham, Mass.: Blaisdell, 1967). Хотя такие комбинаторные построения очень красивы, их основное приложение к задачам получения информации сводится к уменьшению избыточности при построении инвертированных списков. Однако в работе David K. Chow, *Information and Control* **15** (1969), 377–396, замечено, что такое уменьшение может быть достигнуто и без использования комбинаторных конструкций.

Краткая история и библиография. Первой опубликованной статьей, посвященной вопросам технологии получения информации по вторичным ключам, явилась статья L. R. Johnson, *CACM* **4** (1961), 218–222. Многосписочная система была независимо разработана в то же время Ноа Ш. Прайзом (Noah S. Prywes), Г. Дж. Греем (H. J. Gray), В. И. Ландауэром (W. I. Landauer), Д. Лефковицем (D. Lefkowitz) и С. Литвином (S. Litwin) (см. *IEEE Trans. on Communication and Electronics* **82** (1963), 488–492). Еще одна ранняя публикация, оказавшая влияние на более поздние работы, принадлежит Д. Р. Дэвису (D. R. Davis) и Э. Д. Лину (A. D. Lin), *CACM* **8** (1965), 243–246.

С тех пор количество публикаций на данную тему быстро увеличивается, однако почти все они посвящены таким не имеющим отношения к нашей книге вопросам, как пользовательский интерфейс и использование тех или иных языков программирования. Кроме уже указанных статей, автор считает, что наиболее полезными при написании этого раздела (начиная с 1972 года — времени создания настоящего раздела) были следующие работы: Jack Minker and Jerome Sable, *Ann. Rev. of Information Science and Technology* **2** (1967), 123–160; Robert E. Bleier, *Proc. ACM Nat. Conf.* **22** (1967), 41–49; Jerome A. Feldman and Paul D. Rovner, *CACM* **12** (1969), 439–449; Burton H. Bloom, *Proc. ACM Nat. Conf.* **24** (1969), 83–95; H. S. Heaps and L. H. Thiel, *Information Storage and Retrieval* **6** (1970), 137–153; Vincent Y. Lum and Huei Ling, *Proc. ACM Nat. Conf.* **26** (1971), 349–356. Хороший обзор ручных картотек содержится в работе C. P. Bourne, *Methods of Information Handling* (New York: Wiley, 1963), Chapter 5. Сбалансированные схемы были первоначально разработаны в 1965 году Ч. Т. Абрахамом (C. T. Abraham), С. П. Гошем (S. P. Ghosh) и Д. К. Рэй-Чаудури (D. K. Ray-Chaudhuri); см. статью R. C. Bose and Gary G. Koch, *SIAM J. Appl. Math.* **17** (1969), 1203–1214.

Большинство классических алгоритмов для многоатрибутных данных, практическая важность которых общеизвестна, обсуждались в этом разделе; однако в следующее издание настоящей книги планируется добавить еще несколько тем, включая следующие.

- Э. М. Мак-Крейт (E. M. McCreight) ввел понятие *приоритетные деревья поиска* (*priority search trees*) [*SICOMP* **14** (1985), 257–276], которые разработаны специально для представления пересечений динамически изменяющихся семейств интервалов и обработки запросов диапазонов в форме "Найти все записи с $x_0 \le x \le x_1$ и $y \le y_1$" (заметьте, что нижняя грань y должна быть равной $-\infty$, но x может быть ограничен с обеих сторон).
- М. Л. Фредман (M. L. Fredman) нашел ряд фундаментальных нижних граней, показывающих, что последовательность N смешанных вставок, удалений и k-мерных запросов диапазонов требуют в худшем случае выполнения $\Omega\bigl(N(\log N)^k\bigr)$ операций, независимо от используемых структур данных. См. *JACM* **28** (1981), 696–705; *SICOMP* **10** (1981), 1–10; *J. Algorithms* **1** (1981), 77–87.

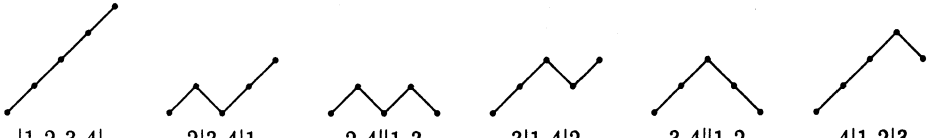
Базовые алгоритмы поиска соответствий шаблону (pattern matching) и приближенных соответствий шаблону в текстовых строках будут обсуждаться в главе 9.

Интересно отметить, что мозг человека гораздо лучше компьютера справляется с поиском информации по вторичным ключам. Люди достаточно легко распознают лица, мелодии и т. п. по фрагментам информации, в то время как для компьютера это практически непосильная задача. Таким образом, вполне вероятно, что в один прекрасный день будет найден совершенно новый подход к решению задач, связанных с поиском информации по вторичным ключам, который сделает это примечание, да и весь текущий раздел, безнадежно устаревшим.

УПРАЖНЕНИЯ

▸ **1.** [*M27*] Пусть $0 \le k \le n/2$. Докажите, что следующая конструкция дает $\binom{n}{k}$ перестановок множества $\{1, 2, \ldots, n\}$, таких, что каждое t-элементное подмножество множества $\{1, 2, \ldots, n\}$ встречается в качестве первых t элементов как минимум одной перестановки при $t \le k$ или $t \ge n - k$. Рассмотрим путь на плоскости из точки $(0, 0)$ в точку (n, r), где $r \ge n - 2k$, в котором i-й шаг выполняется из точки $(i-1, j)$ в точку $(i, j+1)$ или $(i, j-1)$; последняя возможность разрешена только при $j \ge 1$, так что путь никогда не проходит ниже оси x. Существует ровно $\binom{n}{k}$ таких путей. Для каждого из них соответствующая перестановка строится с использованием трех первоначально пустых списков следующим образом: для $i = 1, 2, \ldots, n$, если i-й шаг идет вверх, поместим число i в список B; если шаг идет вниз, поместим i в список A и переместим максимальный на текущий момент список B в список C. Результирующая перестановка представляет собой содержимое списка A, затем списков B и C, причем содержимое каждого списка приводится в порядке возрастания.

Например, при $n = 4$ и $k = 2$ такая процедура определяет шесть путей и перестановок.



|1 2 3 4| 2|3 4|1 2 4||1 3 3|1 4|2 3 4||1 2 4|1 2|3

(Вертикальные линии отделяют списки A, B и C. Эти шесть перестановок соответствуют составным атрибутам в (8).)

Указание. Представьте каждое t-элементное подмножество S в виде пути, который идет из $(0,0)$ в $(n, n-2t)$, i-ый шаг которого идет из $(i-1, j)$ в $(i, j+1)$ при $i \notin S$ и в $(i, j-1)$ при $i \in S$. Преобразуйте каждый такой путь в путь описанного выше вида.

2. [*M25*] (Сакти П. Гош (Sakti P. Ghosh).) Найдите минимально возможную длину l списка $r_1 r_2 \ldots r_l$ ссылок на записи, такого, что множество всех ответов на любой из включающих запросов **1, *1*, 1**, *11, 1*1, 11*, 111 по трем вторичным ключам с двумя возможными значениями оказывается расположенным в последовательных позициях $r_i \ldots r_j$.

3. [*19*] Какие включающие запросы к табл. 2 вызовут ложное выпадение (a) старинного сахарного печенья, (b) овсяных палочек с финиками?

4. [*M30*] Найдите точные формулы для вероятностей в (11), предполагая, что каждая запись имеет r различных атрибутов, случайным образом выбираемых из $\binom{n}{k}$ k-битовых кодов в n-битовом поле, и что запрос включает q различных (но в остальном случайных) атрибутов (не волнуйтесь, если формулы не будут упрощаться).

5. [*40*] Проэкспериментируйте с различными путями снижения избыточности текста при использовании метода Харрисона для поиска подстрок.

▶ **6.** [*M20*] Общее количество m-битовых базовых запросов с t определенными битами составляет $s = \binom{m}{t} 2^t$. Если комбинаторная хеш-функция наподобие (13) конвертирует такие запросы в позиции $l_1, l_2, \ldots, l_s$ соответственно, то среднее количество позиций на запрос составляет $L(t) = (l_1 + l_2 + \cdots + l_s)/s$ (например, в (13) получим $L(3) = 1.75$).

Рассмотрим теперь составную хеш-функцию над $(m_1 + m_2)$-битовым полем, построенную путем отображения первых m_1 бит с помощью одной хеш-функции и оставшихся m_2 с помощью другой, а $L_1(t)$ и $L_2(t)$ — соответствующие средние количества позиций на запрос. Найдите формулу, выражающую значение $L(t)$ в случае составной функции через L_1 и L_2.

7. [*M24*] (Р. Л. Ривест (R. L. Rivest).) Найдите функции $L(t)$, определенные в предыдущем упражнении, для следующих комбинаторных хеш-функций.

(a) $m = 3$, $n = 2$

$0\,0\,* \to 0$
$1\,*\,0 \to 1$
$*\,1\,1 \to 2$
$1\,0\,1 \to 3$
$0\,1\,0 \to 3$

(b) $m = 4$, $n = 2$

$0\,0\,*\,* \to 0$
$*\,1\,*\,0 \to 1$
$*\,1\,1\,1 \to 2$
$1\,0\,1\,* \to 2$
$*\,1\,0\,1 \to 3$
$1\,0\,0\,* \to 3$

8. [*M32*] (Р. Л. Ривест.) Рассмотрим множество $Q_{t,m}$ всех $2^t\binom{m}{t}$ базовых m-битовых запросов типа (10), в которых определено ровно t бит. Дано множество m-битовых записей S. Пусть $f_t(S)$ описывает количество запросов в $Q_{t,m}$, в ответах на которые содержатся члены множества S, а $f_t(s,m)$ представляет собой минимум $f_t(S)$ по всем таким множествам S с s элементами при $0 \le s \le 2^m$. По определению $f_t(0,0) = 0$ и $f_t(1,0) = \delta_{t0}$.

a) Докажите, что для всех $t \ge 1$ и $m \ge 1$ при $0 \le s \le 2^m$

$$f_t(s,m) = f_t(\lceil s/2 \rceil, m-1) + f_{t-1}(\lceil s/2 \rceil, m-1) + f_{t-1}(\lfloor s/2 \rfloor, m-1).$$

b) Рассмотрим некоторую комбинаторную хеш-функцию h, отображающую 2^m возможных записей на 2^n списков, причем каждый список соответствует 2^{m-n} записям. Если все запросы $Q_{t,m}$ равновероятны, то среднее количество проверяемых списков на запрос составляет $1/2^t\binom{m}{t}$, умноженное на

$$\sum_{Q\in Q_{t,m}} (\text{списки, проверяемые для } Q) =$$

$$= \sum_{\text{списки } S} (\text{запросы } Q_{t,m}, \text{ относящиеся к } S) \geq 2^n f_t(2^{m-n}, m).$$

Покажите, что h оптимальна в том смысле, что нижняя грань достигается, когда каждый из списков представляет собой "субкуб"; другими словами, покажите, что в случае, когда каждый список соответствует множеству записей, удовлетворяющих базовому запросу с ровно n определенными битами, неравенство превращается в равенство.

9. [*M20*] Докажите, что если $v = 3^n$, то множество всех троек вида

$$\{(a_1 \ldots a_{k-1}\, 0\, b_1 \ldots b_{n-k})_3, (a_1 \ldots a_{k-1}\, 1\, c_1 \ldots c_{n-k})_3, (a_1 \ldots a_{k-1}\, 2\, d_1 \ldots d_{n-k})_3\},$$

$1 \leq k \leq n$, где a, b, c и d принимает значения 0, 1 или 2 и $b_j + c_j + d_j \equiv 0$ (по модулю 3) при $1 \leq j \leq n-k$ образует Штейнеровскую систему троек.

10. [*M32*] (Томас П. Киркман (Thomas P. Kirkman), *Cambridge and Dublin Math. Journal* **2** (1847), 191–204.) Назовем *системой троек Киркмана* порядка v такое упорядочение $v+1$ объектов $\{x_0, x_1, \ldots, x_v\}$ в тройки, при котором каждая пара $\{x_i, x_j\}, i \neq j$ встречается ровно в одной тройке, за исключением v пар $\{x_i, x_{(i+1) \bmod v}\}, 0 \leq i < v$, которые не встречаются в тройках. Например,

$$\{x_0, x_2, x_4\},\ \{x_1, x_3, x_4\}$$

представляет собой систему троек Киркмана порядка 4.

a) Докажите, что система троек Киркмана может существовать только для $v \bmod 6 = 0$ или 4.

b) Дана Штейнеровская система троек S над v объектами $\{x_1, \ldots, x_v\}$. Докажите, что следующее построение дает другую Штейнеровскую систему троек S' над $2v+1$ объектами и систему троек Киркмана K' порядка $2v-2$. Тройки S' представляют собой все тройки из S плюс

   i) $\{x_i, y_j, y_k\}$, где $j + k \equiv i$ (по модулю v) и $j < k$, $1 \leq i, j, k \leq v$;

   ii) $\{x_i, y_j, z\}$, где $2j \equiv i$ (по модулю v), $1 \leq i, j \leq v$.

   Тройки системы K' представляют собой множество троек S' минус все тройки, содержащие y_1 и/или y_v.

c) Дана система троек Киркмана K на множестве $\{x_0, x_1, \ldots, x_v\}$, где $v = 2u$. Докажите, что следующее построение дает Штейнеровскую систему троек S' над $2v+1$ объектами и систему троек Киркмана K' порядка $2v-2$. Тройки S' представляют собой тройки K плюс

   i) $\{x_i, x_{(i+1) \bmod v}, y_{i+1}\}$, $0 \leq i < v$;

   ii) $\{x_i, y_j, y_k\}$, $j + k \equiv 2i + 1$ (по модулю $v-1$), $1 \leq j < k - 1 \leq v - 2$, $1 \leq i \leq v - 2$;

   iii) $\{x_i, y_j, y_v\}$, $2j \equiv 2i + 1$ (по модулю $v-1$), $1 \leq j \leq v - 1$, $1 \leq i \leq v - 2$;

   iv) $\{x_0, y_{2j}, y_{2j+1}\}$, $\{x_{v-1}, y_{2j-1}, y_{2j}\}$, $\{x_v, y_j, y_{v-j}\}$, $1 \leq j < u$;

   v) $\{x_v, y_u, y_v\}$.

   Тройки K' представляют собой множество троек S' минус все тройки, содержащие y_1 и/или y_{v-1}.

d) Используйте предыдущие результаты для доказательства того, что система троек Киркмана порядка v существует для всех $v \geq 0$ вида $6k$ или $6k+4$ и Штейнеровская система троек над v объектами существует для всех $v \geq 1$ вида $6k+1$ или $6k+3$.

11. [*M25*] В тексте раздела описано использование Штейнеровских систем троек в связи с включающими запросами. Для расширения их применения на все базовые запросы естественно определить следующую концепцию. *Комплементарной системой троек* порядка v является такое размещение $2v$ объектов $\{x_1, \ldots, x_v, \bar{x}_1, \ldots, \bar{x}_v\}$ в тройках, при котором каждая пара объектов встречается ровно в одной тройке, за исключением комплементарных пар $\{x_i, \bar{x}_i\}$, никогда не встречающихся вместе. Например,

$$\{x_1, x_2, x_3\},\ \{x_1, \bar{x}_2, \bar{x}_3\},\ \{\bar{x}_1, x_2, \bar{x}_3\},\ \{\bar{x}_1, \bar{x}_2, x_3\}$$

представляет собой комплементарную систему троек третьего порядка.

Докажите, что комплементарные системы троек порядка v существуют для всех $v \geq 0$, не имеющих вид $3k + 2$.

12. [*M23*] Продолжая упр. 11, постройте комплементарную систему *четверок* порядка 7.

13. [*M25*] Постройте систему четверок с $v = 4^n$ элементами, аналогичную системе троек из упр. 9.

14. [*28*] Обсудите проблему удаления узлов из четревьев, k-d-деревьев и почтовых деревьев, подобных показанному на рис. 45.

15. [*HM30*] (П. Элиас (P. Elias).) Дана большая коллекция m-битовых записей. Предположим, что необходимо найти запись, ближайшую к данному аргументу поиска в том смысле, что согласовано наибольшее количество битов. Разработайте алгоритм эффективного решения этой задачи в предположении, что задан m-битовый код из 2^n элементов, исправляющий t ошибок, и что каждая запись хешируется в один из 2^n списков, соответствующих ближайшему кодовому слову.

▸ **16.** [*25*] (В. Х. Кац (W. H. Kautz) и Р. К. Синглтон (R. C. Singleton).) Покажите, что Штейнеровская система троек порядка v может использоваться для построения $v(v-1)/6$ v-битовых кодовых слов, таких, что никакое слово не содержится в суперпозиции двух других.

▸ **17.** [*M30*] Рассмотрите следующий путь сведения $(2n+1)$-битовых ключей $a_{-n} \ldots a_0 \ldots a_n$ к $(n+1)$-битовому блоку адресов $b_0 \ldots b_n$:

$b_0 \leftarrow a_0$;
если $b_{k-1} = 0$, то $b_k \leftarrow a_{-k}$, иначе $b_k \leftarrow a_k$ для $1 \leq k \leq n$.

a) Опишите ключи, появляющиеся в блоке $b_0 \ldots b_n$.

b) Чему равно наибольшее количество блоков, которые необходимо проверить при базовом запросе с t определенными битами?

▸ **18.** [*M35*] (*Конструирование ассоциативного блока.*) Множество m-элементных наборов типа (13) с ровно $m - n$ звездочками в каждой из 2^n строк называется ABD(m, n)*, если в каждом столбце содержится одно и то же количество звездочек и если каждая пара строк имеет "несоответствие" (0 против 1) в некотором столбце. Каждое m-битовое бинарное число будет в таком случае соответствовать в точности одной строке. Например, (13) является ABD$(4, 3)$.

a) Докажите, что ABD(m, n) невозможно, кроме случаев, когда m является делителем $2^{n-1}n$ и $n^2 \geq 2m(1 - 2^{-n})$.

b) Будем говорить, что строка ABD имеет *нечетный паритет*, если в ней содержится нечетное количество единиц. Покажите, что для любого выбора $m - n$ столбцов в ABD(m, n) количество строк с нечетным паритетом с "*" в этих столбцах равно количеству строк с четным паритетом. В частности, каждое возможное расположение символов "*" должно встречаться в четном количестве строк.

---

* От *Associative Block Designs*. — *Прим. перев.*

c) Найдите ABD(4, 3), которое не может быть получено из (13) путем перестановки и/или дополнения столбцов.

d) Постройте ABD(16, 9).

e) Постройте ABD(16, 10). Начните с ABD(16, 9) из п. (d) вместо ABD(8, 5) из (15).

19. [*M22*] Проанализируйте ABD(8, 5) из (15) так же, как (13) было проанализировано в (14). Сколько из 32 позиций должно быть проверено при запросе с k неопределенными битами в среднем и в наихудшем случаях?

20. [*M47*] Найдите все ABD(m, n) при $n = 5$ или $n = 6$.

Новый раздел 6.6 в следующем издании книги планируется посвятить постоянным структурам данных. Эти структуры позволяют представить измененную информацию таким образом, что история изменений может быть эффективно восстановлена. Иными словами, можно сделать множество вставок и удалений, но, тем не менее, оставаться способными выполнить поиск так, как будто обновлений после некоторого данного момента времени вовсе не было. Пока что в настоящее издание включены лишь ссылки на литературу по этой теме:

- J. K. Mullin, *Comp. J.* **24** (1981), 367–373;
- M. H. Overmars, *Lecture Notes in Comp. Sci.* **156** (1983), Chapter 9;
- E. W. Myers, *ACM Symp. Principles of Prog. Lang.* **11** (1984), 66–75;
- B. Chazelle, *Information and Control* **63** (1985), 77–99;
- D. Dobkin and J. I. Munro, *J. Algorithms* **6** (1985), 455–465;
- R. Cole, *J. Algorithms* **7** (1986), 202–220;
- D. Field, *Information Processing Letters* **24** (1987), 95–96;
- C. W. Fraser and E. W. Myers, *ACM Trans. Prog. Lang. and Systems* **9** (1987), 277–295;
- J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, *J. Comp. Syst. Sci.* **38** (1989), 86–124;
- R. B. Dannenberg, *Software Practice & Experience* **20** (1990), 109–132;
- J. R. Driscoll, D. D. K. Sleator, and R. E. Tarjan, *JACM* **41** (1994), 943–959.

Таблицы инструкций (программы) будут создаваться математиками с опытом вычислительной работы и, вероятно, с определенными способностями к решению головоломок. Перевод на некоторой стадии каждого известного процесса в форму таблиц инструкций, скорее всего, будет значительной частью такой работы. ... Процесс построения таблиц инструкций должен очаровывать. Реальной опасности стать слугой машины нет, ибо любой более или менее механический процесс можно передать самой машине.

— АЛАН М. ТЬЮРИНГ (ALAN M. TURING) (1945)