Глава 11

Структуры хранения и основные алгоритмы СУБД

В этой главе обсуждаются вопросы, которые необходимо решать разработчику СУБД при выборе методов хранения данных: организация хранения таблиц, структуры индексов, а также алгоритмы, необходимые для реализации операций реляционной алгебры и алгебры SQL. Далеко не все из рассматриваемых альтернативных проектных решений применяются в системе PostgreSQL. Ни разработчикам приложений, ни администраторам базы данных обычно не нужно реализовывать что-либо из описанного в этой главе, однако знание этого материала помогает наилучшим образом использовать возможности СУБД.

11.1. Хранение объектов логического уровня

Объекты логического уровня — это объекты, в терминах которых происходит взаимодействие приложения с сервером базы данных. Поскольку PostgreSQL относится к числу SQL-ориентированных систем, основным видом логических объектов являются таблицы — коллекции объектов, каждый из которых содержит фиксированный набор атрибутов, один и тот же для всех объектов одной таблицы.

В то же время место для хранения любых видов данных выделяется в терминах блоков (или страниц) одинакового размера, как минимум в пределах одного табличного пространства. В PostgreSQL размер страницы фиксирован и обычно составляет 8 Кб. Этот размер можно изменить на 16 или 32 Кб при компиляции из исходного кода, но в любом случае размер будет одинаковым для всех табличных пространств.

11.1.1. Размещение коллекций объектов

Разработчик СУБД должен принимать решения, определяющие, каким образом объекты коллекции (например, строки таблиц) будут размещаться в блоках. Поскольку не может существовать единственного метода, оптимального для всех применений и для всех видов нагрузки, высокопроизводительные СУБД обычно предоставляют несколько альтернативных методов размещения логических объектов и коллекций. Метод размещения (из предоставляемого СУБД ассортимента) выбирается разработчиком приложения совместно с администратором базы данных для каждой таблицы на этапе проектирования схемы хранения базы данных с учетом требований приложений, использующих эту базу.

Варианты отображений логических объектов на структуры хранения

В процессе проектирования методов хранения логических объектов разработчик СУБД должен принимать решения по ряду вопросов, определяющих свойства и характеристики структуры хранения.

Фрагментация объектов определяет, каким образом и при выполнении каких условий данные одного объекта разбиваются на части, размещаемые на разных страницах хранилища. Например, в системе PostgreSQL значения атрибутов строки таблицы, длина которых превышает порог, могут подвергаться сжатию, и если после этого длина все еще превышает порог, то размещаются отдельно от значений остальных атрибутов строки таблицы. При этом стратегия сжатия и размещения зависит от типа значения и может быть переопределена для отдельных колонок.

Адресация и перемещаемость определяют, каким образом система представляет ссылки на объект, и может ли измениться положение объекта в хранилище после того, как он был размещен первоначально.

Во многих СУБД применяется обновление «на месте», т. е. старые значения заменяются на новые, возможно, другой длины. При этом фактически данные могут перемещаться, однако обычно в таких системах принимаются меры, для того чтобы адреса оставались неизменными. В системе PostgreSQL при любом изменении создается новая версия объекта, которая получает новый адрес, а предыдущая версия остается доступной по

старому адресу, до тех пор пока она не будет удалена в результате сборки мусора. При этом все объекты могут перемещаться в пределах одной страницы.

Размещение по значениям определяет, зависит ли размещение объектов коллекции от значений атрибутов объектов. В PostgreSQL размещение по значениям атрибутов не применяется.

Упорядочение определяет, учитывается ли отношение порядка для некоторого атрибута (или набора атрибутов) при размещении объектов. Такая организация хранения полезна, если достаточно часто выполняются запросы, в которых требуется выборка объектов по интервалу значений атрибута. Заметим, что размещение по значениям, намеченное в предыдущем пункте, не обязательно предполагает упорядоченность.

В системе PostgreSQL упорядочивание строк таблиц реализуется с помощью команды CLUSTER, которая реорганизует таблицу в соответствии с упорядочением в одном из индексов. Однако при обновлениях это упорядочение не поддерживается, поэтому необходима периодическая повторная реорганизация. Этот механизм применяется редко, поскольку доступ к таблице во время выполнения такой операции полностью блокируется.

Совместное размещение коллекций в общем наборе страниц хранилища позволяет собрать вместе (на одной или смежных страницах) строки разных таблиц, содержащие одинаковые значения каких-либо атрибутов. В системе PostgreSQL значения атрибутов не используются при выборе места для записи строки, и совместное размещение не применяется.

Решения по этим вопросам определяют, какие алгоритмы доступа к данным коллекции могут и будут использоваться при работе СУБД.

При проектировании структуры хранения необходимо также принимать решения, связанные с поведением структуры при внесении изменений. В частности, необходимо определить:

- стратегию поиска и выделения свободного места для новых объектов и стратегию добавления новых страниц для объектов коллекции;
- действия при переполнении страницы (при увеличении размеров объекта или при добавлении новых объектов на страницу);
- обработку удалений объектов и недостаточно заполненных страниц;

- возможность одновременного доступа нескольких транзакций, в том числе обновляющих данные на одной странице;
- реорганизацию структуры хранения с целью улучшения ее характеристик, если при обновлениях структура может деградировать (примером может служить сборка мусора, которая в PostgreSQL обычно выполняется фоновым процессом).

Адресация и перемещаемость

Доступ к объектам по адресу на уровне хранения необходим даже в тех случаях, когда модель данных не предусматривает явные ссылки на объекты. Например, в реализациях реляционной модели данных доступ к строкам таблиц может производиться с использованием индексов, и для перехода от индексной записи непосредственно к объекту используется прямая адресация. Наиболее эффективным методом адресации является использование низкоуровневого (абсолютного или относительного) адреса страницы и смещения на странице. Такая адресация, однако, несовместима с необходимостью перемещать объект при изменении его длины или длины других объектов, размещенных на той же странице.

Для того чтобы обеспечить неизменность адреса, используется ряд приемов.

- Замена смещения на порядковый номер объекта на странице, возможно, с поддержкой массива смещений объектов, размещенных на этой странице. Этот прием обеспечивает неизменность адреса объекта при его перемещении в пределах одной страницы.
- Использование «якорей» объектов фиксированной длины, содержащих ссылку на фактическое размещение объекта. Как правило, объект размещается на той же странице, что и якорь, что исключает потерю эффективности из-за косвенной адресации. Однако при увеличении длины объект может перемещаться в другое место с сохранением размещения якоря.

Как видно из структуры страницы, описываемой ниже в разделе 11.1.2, в системе PostgreSQL применяется первый из этих приемов, поэтому в качестве ссылки на объект можно использовать номер страницы, на которой размещается объект, в сочетании с номером позиции в массиве указателей на объекты этой страницы. Такая ссылка остается неизменной, если объект перемещается в пределах страницы.

Заметим, что фиксированная адресация несовместима с размещением объектов по значению. Это обстоятельство не имеет значения в системе PostgreSQL, поскольку размещение по значению для основных таблиц не используется. Конечно, размещение по значению необходимо в индексных структурах, однако нет необходимости ссылаться на объекты индекса извне самой структуры этого индекса.

Размещение по значениям и упорядоченность

Размещение объектов с учетом значений одного или нескольких атрибутов позволяет исключить использование индекса при фильтрации по такому атрибуту (или набору атрибутов). По существу, такое хранение является комбинацией индексной структуры и структуры для хранения коллекций.

Если индексная структура, на основе которой организовано такое хранение, поддерживает упорядоченность, то появляется возможность весьма эффективно выполнять запросы, требующие фильтрации по интервалу значений ключа, по которому упорядочены строки таблицы.

Подчеркнем, что такая организация хранения может применяться для размещения по ключу, который не является уникальным.

Хотя в PostgreSQL подобные структуры для хранения таблиц не используются, соответствующий алгоритм выборки данных применяется в том случае, если все данные, необходимые для выполнения запроса, содержатся в индексе и обращение к основной таблице не требуется.

Совместное размещение коллекций

Совместное хранение предполагает размещение объектов нескольких коллекций на одной странице. Такое размещение полезно, если объекты этих коллекций связаны общим значением некоторого атрибута и часто используются вместе (например, выполняется операция соединения по этому атрибуту).

В системе PostgreSQL такой способ организации хранения таблиц не поддерживается на уровне структур хранения, однако его можно моделировать на уровне логической структуры, используя атрибуты с множественными значениями (например, массивы).

11.1.2. Размещение данных на страницах

Напомним, что, поскольку PostgreSQL поддерживает объектные расширения, типы атрибутов могут быть достаточно сложными — в частности, могут содержать коллекции (представленные массивами), слабоструктурированные данные (например, в формате JSON), тексты и пр. Длины скалярных атрибутов (например, символьных строк text или varchar) могут изменяться. Размер памяти, необходимый для хранения объекта, может варьироваться в очень широких пределах как для разных коллекций, так и для разных объектов одной коллекции и даже значений одного атрибута в одной коллекции.

Для того чтобы достаточно эффективно обрабатывать такое большое разнообразие, наборы логических объектов PostgreSQL (отношения) представляются структурой хранимых таблиц. Принципиальных различий между логическими и хранимыми таблицами нет, и обычно их не различают. Если содержимое логической таблицы (отношения) удовлетворяет ограничениям, наложенным на хранимые таблицы, то для хранения этой логической таблицы используется одна хранимая и отображение становится тождественным.

Ограничение, накладываемое на хранимые таблицы, состоит в том, что длина записей в таких таблицах не может превышать размер страницы и фактически ограничивается некоторым порогом (обычно около 2 Кб), обеспечивающим возможность хранения нескольких записей на одной странице. Если размер объекта логического отношения превышает этот порог, то значения атрибутов этого объекта преобразуются и часть данных может быть вынесена в другую хранимую таблицу, которая связана с тем же логическим отношением.

Значения переменной длины

В системе PostgreSQL для представления значений переменной длины используется общий формат, не зависящий от типа значения: непосредственно перед значением записывается его длина в байтах, включая место, необходимое для хранения самой длины. В общем случае для записи длины выделяется 4 байта, из которых 2 бита используются для служебных целей; для записи длины остается 30 бит, что ограничивает максимальный размер значения одним гигабайтом.

Выделение 4 байтов для хранения длины коротких значений (скажем, строковых значений небольшой длины) привело бы к чрезмерному расходу памяти.

Поэтому для коротких значений длина записывается в один байт, старший (служебный) бит которого установлен в ноль. Оставшиеся 7 бит дают возможность хранения длины, не превышающей 127, включая один байт на саму длину, поэтому максимально возможный размер такого значения составляет 126 байт.

Похожие схемы организации хранения объектов переменной длины применяются и в других СУБД.

Значения большего размера могут быть представлены в нескольких различных форматах, и при необходимости система PostgreSQL выполняет преобразования. Формат, в котором значение без всяких изменений записывается непосредственно после длины, необходим для передачи значений между сервером базы данных и приложением (клиентом), а также применяется для временного хранения значений в оперативной памяти.

Если размер значения превышает определенный порог (обычно около 2 Кб), такое значение разрезается на фрагменты, каждый из которых оформляется как отдельная запись в служебной хранимой таблице типа TOAST (The Oversized-Attribute Storage Technique), которая представляет собой внутреннюю структуру, используемую для хранения объектов большого размера. Конечно, идентификация этих фрагментов определяет значения, из которых они получены, а также логический объект, содержащий эти значения. И, конечно, длина этих фрагментов не превышает порог.

Такая служебная таблица может быть автоматически создана для каждого логического отношения в дополнение к основной хранимой таблице, в которую записываются объекты этого логического отношения. Таким образом, при постоянном хранении в базе данных значения большого размера представлены набором фрагментов («тостов»); кроме этого, длинные значения могут храниться в сжатом виде.

Структура страниц

Организация данных на странице в системе PostgreSQL не зависит от типов хранимых данных и от типа объекта, к которому относятся эти данные: хранимые таблицы, индексы и др. Единицей хранения на странице является запись, которая может быть строкой таблицы, индексной записью, фрагментом значения большого размера (тостом). Для модулей, отвечающих за размещение данных на странице, это не имеет никакого значения.

Поскольку требования к структурам хранения довольно близки в самых различных системах, организация страниц оказывается очень похожей во многих СУБД. Например, в PostgreSQL она содержит следующие разделы:

Заголовок страницы фиксированного размера (24 байта).

Массив указателей на записи, содержащиеся на этой странице. Эти указатели содержат смещения записей относительно начала страницы, их длину и служебные биты. Массив имеет переменный размер, зависящий от числа записей (объектов) на этой странице.

Незанятый участок памяти, который может быть использован для добавления новых записей (или для расширения имеющихся в тех системах, в которых записи могут обновляться), а также для расширения массива указателей.

Область данных, в которой находятся размещенные на странице объекты или их заголовки (для объектов большого размера).

Дополнительная область, которая может по-разному использоваться для индексов разных типов и не применяется для таблиц.

Ссылки на объекты включают идентификацию страницы базы данных и номер позиции в массиве указателей, размещенном на этой странице. Такая косвенная адресация дает возможность перемещать объекты в пределах страницы без изменения ссылок.

Несмотря на то что в PostgreSQL при любых изменениях создается новая версия объекта, а ранее созданная версия не изменяется, перемещение объектов требуется для устранения фрагментации памяти на странице после удаления устаревших версий.

Вспомогательные структуры

Кроме основных файлов, используемых для хранения таблиц или индексов, в системе PostgreSQL создаются дополнительные структуры:

- файл, хранящий информацию о наличии свободного места на страницах таблицы или индекса (free space map, FSM);
- карта видимости (visibility map, VM), описывающая, какие страницы содержат записи, обрабатываемые активными транзакциями, и какие страницы могут содержать записи, подлежащие удалению.

Карта свободного пространства организована как дерево. На нижнем уровне (в листьях этого дерева) хранится один байт на каждую страницу таблицы или индекса, указывающий на возможность добавления записей на эту страницу. Для таблиц или индексов большого размера верхние уровни содержат обобщенную информацию о наличии свободного места на страницах, описываемых вершинами дерева следующего уровня. Карта свободной памяти не создается для индексов на основе хеширования.

Карта видимости содержит информацию о том, какие страницы могут содержать устаревшие версии строк. Значения битов признаков в карте видимости устанавливаются консервативно: 1 обязательно означает, что условие выполняется (устаревших версий нет), однако 0 означает только, что условие может нарушаться (устаревшие версии могут быть). Карта видимости создается для таблиц, но не для индексов.

Если страница таблицы не содержит устаревших версий строк, ее не нужно обрабатывать при сборке мусора (vacuum). Кроме того, карта видимости определяет необходимость дополнительной проверки после поиска по индексу. Дело в том, что индекс может содержать записи, указывающие на устаревшие версии строк таблиц, и поэтому для исключения таких строк из результатов поиска необходима дополнительная проверка содержимого табличной страницы. Если же известно, что страница не содержит устаревшие версии, такую проверку можно опустить.

Кроме этого, к служебным структурам можно отнести таблицы TOAST, применяемые для хранения больших объектов, потому что обычно эти таблицы не упоминаются явно в запросах пользователей. Организация хранения таблиц TOAST не отличается от хранения обычных таблиц: для них также создаются необходимые вспомогательные структуры.

11.1.3. Хранение больших объектов

В системе PostgreSQL имеется возможность создания больших объектов, не входящих в какую-либо таблицу или другую структуру (однако фактически такие объекты создаются в одной из системных таблиц). При создании большого объекта PostgreSQL возвращает значение типа oid, которое можно использовать в дальнейшем для доступа к этому объекту. Содержимое больших объектов никак не интерпретируется PostgreSQL; с точки зрения ядра СУБД это просто последовательность байтов.

Другими словами, логическая структура данных, хранимых в большом объекте, полностью определяется кодом приложения. Заметим, что такой код может находиться как в программе, выполняемой в роли клиента базы данных, так и в виде функций, размещенных на сервере базы данных и таким образом расширяющих функциональность PostgreSQL.

По историческим причинам размер типа oid составляет 4 байта, что ограничивает максимальное количество больших объектов, которые могут быть созданы в одном кластере баз данных.

Понятие большого объекта считается устаревшим и, скорее всего, будет вытеснено структурами TOAST, однако предельный размер объекта, хранимого в TOAST, составляет 1 Гб, а максимальный размер большого объекта — 4 Тб. Кроме этого, интерфейс доступа к большим объектам лучше приспособлен для обработки потоков байтов, таких, как звук или видео. Заметим, что нет никаких концептуальных препятствий, для того чтобы реализовать аналогичный интерфейс и для объектов, хранимых в структурах TOAST.

11.1.4. Строки или колонки?

В зависимости от класса задач, в которых используются данные, для каждой таблицы может быть выбран один из нескольких способов хранения, если СУБД такие альтернативы предоставляет. Так, для задач оперативной обработки (OLTP), характеризующихся тем, что каждое выполнение приложения использует очень небольшую часть объектов базы данных, а обновления этих объектов происходят относительно часто, обычно используется хранение таблицы *по строкам*. Это означает, что атрибуты одной строки таблицы хранятся в смежных областях памяти внутри страницы, и, поскольку объекты имеют небольшие размеры, на каждой странице размещается несколько объектов.

Другой способ хранения таблиц — *по колонкам* — состоит в том, что в смежных областях памяти хранятся значения одного атрибута из разных объектов (строк) отношения. Такое хранение целесообразно использовать, если запросы, как правило, выбирают значительную часть хранимых в таблице объектов, но используют небольшое количество атрибутов, и при этом обновления выполняются относительно редко. Все эти предположения, как правило, выполняются для задач аналитической обработки (OLAP).

В начале 80-х гг. было экспериментально установлено, что хранение данных по строкам дает преимущества в производительности для класса задач, который

доминировал в то время (OLTP), и на характеристиках оборудования, которое было доступно в то время. В дальнейшем, в связи с расширением области применения баз данных и изменением характеристик оборудования, оказалось, что для решения аналитических задач хранение по колонкам более предпочтительно.

Во многих случаях целесообразно применять гибридные структуры: хранить в столбцах значения, составленные из нескольких атрибутов, часто используемых вместе в одном запросе. Своего рода экстремальным решением является хранение каждого атрибута как отдельной коллекции.

В системе PostgreSQL хранение по колонкам не реализовано, хотя возможно использование внешних хранилищ через механизм оберток сторонних данных (foreign data wrappers). Можно моделировать хранение по колонкам, заменяя таблицу на совокупность нескольких таблиц с малым числом колонок в каждой, однако такое решение не будет эффективным, поскольку большой объем памяти будут занимать заголовки строк. В версиях системы PostgreSQL, начиная с 12, возможно подключение альтернативных методов хранения таблиц. Среди таких методов, несомненно, будет доступна и организация хранения по колонкам.

11.2. Индексы

В контексте структур хранения индексом называется вспомогательная (избыточная) структура данных, предназначенная для ускорения некоторых классов запросов. Более конкретно — индексы позволяют применять эффективные алгоритмы обработки запросов, содержащих условия на значения атрибутов, для которых построен индекс.

Любой индекс можно рассматривать как коллекцию, состоящую из объектов специального вида (индексных записей), включающих два атрибута:

индексный ключ, представляющий поисковый образ (как правило, значения атрибута или атрибутов, для которых построен индекс);

информацию об объектах, которую в той или иной форме задает указатель (или указатели) на объекты, содержащие значения атрибутов, совпадающие со значениями в индексном ключе.

В случае если значение индексного ключа содержится в нескольких объектах, в системе PostgreSQL индексные записи, как правило, дублируются; в других системах вместо этого могут храниться списки указателей. Очевидно, в первом случае возникает избыточность из-за дублирования ключей, во втором — необходимо специальным образом обрабатывать списки, которые могут быть очень длинными. Конечно, для атрибутов, на значения которых наложено ограничение целостности UNIQUE, дублирование ключей не требуется. Известны разновидности индексов, в которых вместо списков указателей используются битовые карты.

Особенность индексов по сравнению с другими коллекциями состоит в том, что для индексов имеет смысл только выполнение запросов весьма ограниченного вида, а именно выполняющих фильтрацию по значениям ключа (не обязательно по совпадению значений, но в любом случае проверяется истинность некоторого предиката, связывающего поисковый критерий со значениями индексного ключа). Благодаря этому для индексов оказывается целесообразным применять значительно более развитые конструкции, чем для коллекций общего вида.

Изменение значения атрибута в таблице приводит к необходимости удаления из индекса старого значения и вставки нового. Ассортимент операций обновления индексов обычно включает только эти две операции: вставка новой записи (или ссылки) и удаление существующей. Как известно, в PostgreSQL операции UPDATE также выполняются как удаление старой и вставка новой версии, но в других СУБД обновление таблиц может выполняться иначе.

Перечислим основные требования, предъявляемые к индексным структурам, и критерии их оценки.

Поскольку основное назначение индексов — ускорение поиска данных, наиболее важным критерием их полезности является время поиска в индексе. Традиционно для оценки времени поиска применяется количество страниц, которые необходимо обработать, для того чтобы выполнить запрос на поиск в индексе. Применительно к ранним системам важность этого критерия связана с тем, что при размещении страниц на диске время ожидания обмена существенно превосходит суммарное время выполнения всех остальных операций, необходимых для выполнения запроса. В связи с ростом объемов оперативной памяти и распространением устройств хранения SSD значение критериев, основанных на времени доступа к вращающимся дискам, снизилось, однако оценки, основанные на количестве обрабатываемых страниц, по-прежнему дают некоторое представление о сложности индексных структур.

Можно предполагать, что количество операций, выполняемых процессором при обработке одной страницы индекса, примерно одинаково для всех страниц (другими словами, время процессора не будет существенно различаться при обработке разных индексных страниц). Количество обработанных страниц можно поэтому применять для оценки времени поиска в индексе, даже если все требуемые страницы уже находятся в буферах в оперативной памяти.

Другим важным критерием является сложность модификации. Индекс называется *динамическим*, если при добавлении или удалении записей не требуется его глобальная перестройка и производительность индекса не деградирует при многократных изменениях.

Еще один критерий — доля памяти, занятой данными, по отношению к общему размеру памяти, выделенной для индекса. Само по себе заполнение памяти не так важно, как время доступа, однако при низкой заполненности непомерно увеличивается количество страниц, занимаемых индексом, что косвенно влияет и на время доступа.

11.2.1. Одномерные индексы

Индекс называется *одномерным*, если значения индексного ключа рассматриваются как скалярные. Особенность одномерных ключей состоит в том, что для них обычно существует естественное упорядочение. Например, такое упорядочение присуще числовым типам, текстовым строкам и моментам времени. Составные ключи, построенные из значений нескольких атрибутов с лексикографическим упорядочиванием, также являются одномерными.

B-деревья

Упорядоченные одномерные индексы, организованные в некоторое дерево, являются наиболее часто используемым видом индексов. Такие индексы обеспечивают поиск индексных записей по значению атрибута на совпадение и на попадание в заданный интервал значений. Можно также рассматривать поиск по префиксу, однако его реализация по существу не отличается от поиска по интервалу значений.

Рассмотрим структуру B^+-дерева. Напомним, что эта структура отличается от структуры B-дерева тем, что индексные записи размещаются только в листовых вершинах.

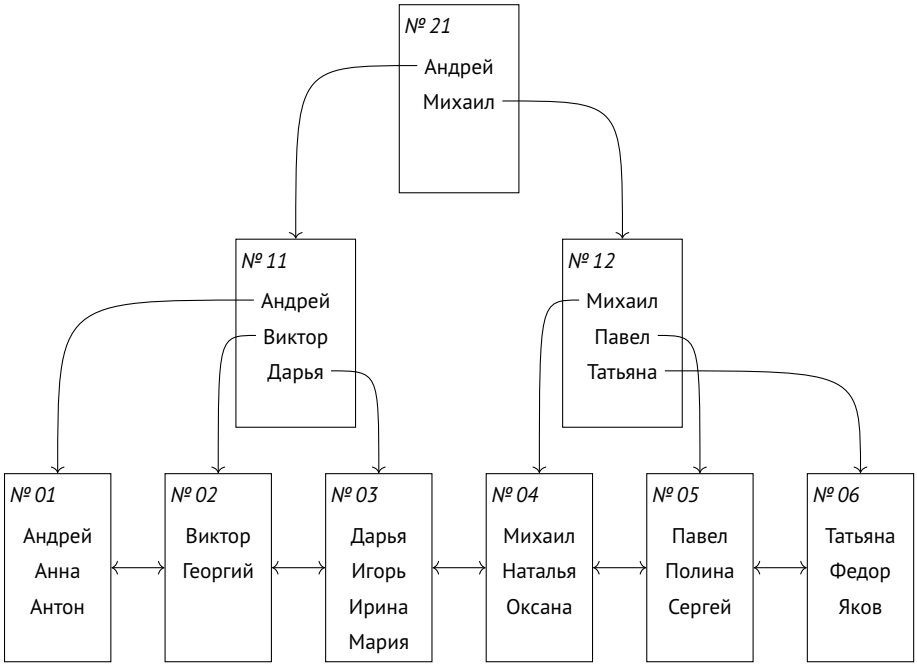

Рис. 11.2.1. Структура B^+-дерева

Пример B^+-дерева показан на рис. 11.2.1. Каждая вершина дерева соответствует странице в пространстве, выделенном для хранения индекса. Индексные записи, включенные в дерево, размещаются в соответствии с отношением порядка для индексных ключей. Если все индексные записи не помещаются в одну страницу, используется необходимое количество логически упорядоченных страниц, и при этом поддерживается отношение порядка для всех индексных ключей. Таким образом, каждая страница содержит некоторый интервал значений ключей; все ключи, попадающие в этот интервал, размещаются на этой странице. Если ключи индексных записей уникальны, то интервалы, соответствующие разным страницам, не пересекаются, иначе граничные значения интервалов в соседних страницах могут совпадать. Совокупность этих страниц составляет нижний (нулевой) уровень дерева.

Если количество страниц на уровне i превышает 1, то для каждой такой страницы создается вспомогательная индексная запись, размещаемая на уровне $i + 1$. Она включает границу интервала ключей, содержащихся на странице уровня i, и указатель на эту страницу. Различные реализации B^+-дерева могут исполь-

зовать минимальный или максимальный ключ в качестве граничного; в системе PostgreSQL используется минимальный. Полученные таким образом записи образуют уровень $i + 1$ и если этот уровень также содержит более одной страницы, то строится следующий уровень.

При добавлении новая запись вставляется в ту страницу нулевого уровня, которой соответствует значение индексного ключа. Если на странице нет места, то выполняется процедура расщепления, которая создает новую страницу нулевого уровня и переносит на нее примерно половину записей с переполненной страницы, а затем добавляет новую запись на следующий уровень (на котором также при необходимости может произойти расщепление). На рис. 11.2.2 показано состояние дерева после вставки записи и расщепления блока № 03.

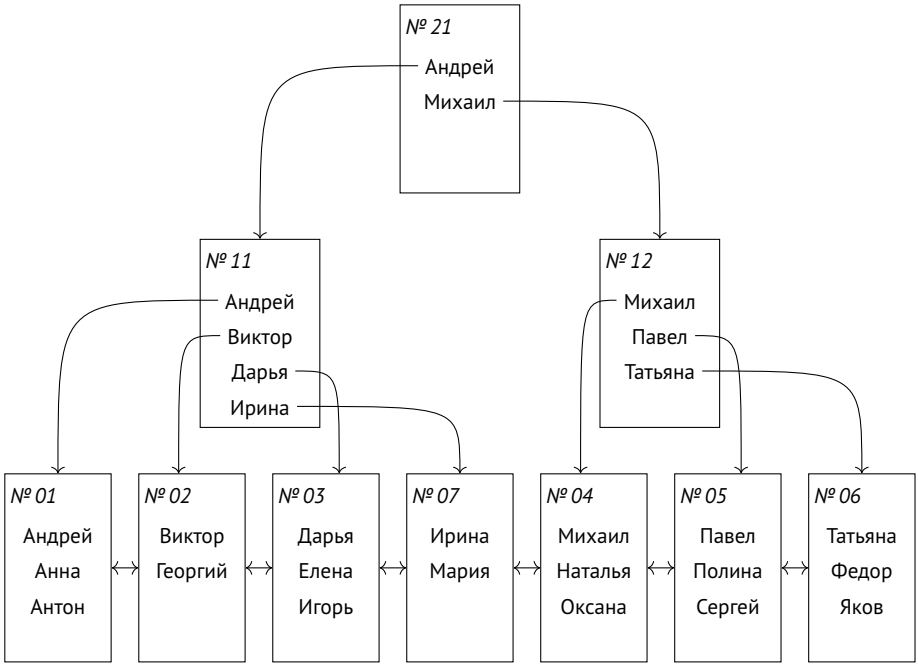

Рис. 11.2.2. Расщепление вершины B^+-дерева при добавлении записи

Легко видеть, что дерево оказывается сбалансированным, т. е. длина всех путей от корня дерева до листьев одинакова. Такая процедура гарантирует, что все страницы индекса будут заполнены не менее чем на 50 %. Можно доказать, что при поступлении новых записей в случайном порядке ожидаемая средняя заполненность блоков составит $\ln 2 \approx 0{,}69$.

Существуют варианты алгоритма, обеспечивающие более высокую степень заполнения за счет более сложных трансформаций дерева при его модификациях. Например, если вместо расщепления одной страницы на две распределять содержимое двух соседних переполнившихся страниц на три (примерно поровну), то можно получить заполнение страницы не ниже чем $^2/_3$. При этом, однако, реструктуризации становятся более частыми, и, кроме того, усложняются механизмы, обеспечивающие корректность сканирования дерева параллельно выполняемыми запросами на поиск в индексе.

Условие заполненности страниц на 50 % может нарушаться при удалениях записей. В этом случае необходимо слияние страниц, содержащих недостаточное количество индексных записей, с соседними страницами. Если процедура слияния страниц предусмотрена, то структура B^+-дерева становится динамической (в определенном выше смысле) и не деградирует при любых изменениях в данных.

Многие практические реализации, в том числе PostgreSQL, однако, слияние не делают. В результате структура индекса может деградировать и потребуется ее реорганизация, при которой все дерево строится заново. По-видимому, отказ от динамического сжатия связан с тем, что необходимость в нем возникает относительно редко: базы данных обычно имеют тенденцию расти, а не сокращаться. Тем не менее деградация дерева большого размера может существенно ухудшить производительность системы, поэтому администратору базы данных необходимо следить за состоянием индексов.

Алгоритм поиска в B^+-дереве начинает работу с единственной страницы верхнего уровня (корня дерева), выбирая на ней ключ, указывающий на страницу следующего уровня, которая может содержать искомый ключ. Если страницы нижележащего уровня представляются минимальными ключами (как в PostgreSQL), то это — наибольший ключ, не превосходящий искомый, а если максимальным — то наименьший, превосходящий искомый. Структура страниц, применяемая в PostgreSQL, дает возможность внутри страницы использовать алгоритм бинарного поиска. Если запрос предполагает поиск по интервалу, то выполняется поиск левого (или правого, в зависимости от вида условия) конца интервала.

Далее выбирается страница следующего уровня, на которую ссылается указатель, связанный с найденным ключом. Спуск повторяется до тех пор, пока не будет достигнут нулевой уровень, на котором поиск по точному значению заканчивается (положительным или отрицательным результатом), а поиск по интервалу продолжается последовательным просмотром нулевого уровня вплоть

до ключа, который окажется больше правого (меньше левого) конца интервала. Для того чтобы упростить последовательный просмотр, обычно страницы одного уровня связываются двухсторонними указателями.

Например, при поиске значения «Дмитрий» в корневом блоке № 21 будет выбран ключ «Андрей», указывающий на блок № 11, затем ключ «Дарья», указывающий на блок № 03, в котором искомый ключ отсутствует, и будет возвращен результат, указывающий, что искомого ключа в индексе нет.

Количество страниц, которое будет просмотрено при спуске, равно количеству уровней и оценивается логарифмом от числа индексных записей N. Если среднее количество записей на одной странице равно m, то количество уровней в дереве будет примерно равно $\log_m N$.

Для доказательства следует заметить, что при каждом подъеме на один уровень количество страниц сокращается в m раз. Поэтому количество уровней равно наименьшему числу k, такому, что $N \leqslant m^k$.

При поиске по интервалу к этому значению прибавляется количество страниц нулевого уровня, занятых индексными записями, входящими в интервал.

Теоретически структура B-дерева является динамической, т. к. при обновлениях могут перестраиваться только страницы, находящиеся на пути к корню, что составляет небольшую часть дерева. Для реализаций, которые не делают слияние недостаточно заполненных страниц, это не совсем так, но даже в этом случае структура B-дерева обладает очень хорошими эксплуатационными свойствами.

Важно заметить, что простые варианты алгоритмов, описанные выше для индексов на основе B-деревьев и ниже для других типов индексов, недостаточны для применения в промышленных системах, в которых требуется учитывать много дополнительных требований, в первую очередь возможность конкурентной (параллельной или псевдопараллельной) обработки. В частности, в системе PostgreSQL реализация B-деревьев использует алгоритмы конкурентного доступа, описанные в [42].

Индексы на основе хеширования

Идея любого варианта метода хеширования состоит в том, что адрес, по которому размещена запись, вычисляется некоторым преобразованием ключа. Функция, выполняющая такое преобразование, называется *функцией хеширования*. Можно сказать, что функция хеширования отображает пространство

ключей (обычно потенциально очень большое, но редко заполненное) в пространство адресов значительно меньшего размера. При таком преобразовании неизбежно возникновение коллизий — разные ключи могут отображаться в один и тот же адрес.

Поскольку функция хеширования почти никогда не сохраняет упорядочение, индексы на основе хеширования можно использовать только для проверки на равенство значений атрибута. Исключением является метод многомерного хеширования LSH (locality sensitive hashing), однако этот метод крайне редко встречается в системах управления базами данных, и мы не будем его рассматривать.

Чтобы определить индексную структуру на основе хеширования, необходимо:

- задать функцию хеширования;
- определить адресное пространство (обычно в СУБД в качестве адресов используются номера страниц в файле, в котором хранится индекс, при этом каждая страница может хранить несколько записей);
- выбрать метод размещения записей переполнения (тех записей, которые не могут быть размещены по адресу, вычисленному функцией хеширования вследствие коллизий).

Если количество записей переполнения невелико, то индексы на основе хеширования обеспечивают очень быстрый доступ по точному значению ключа: функция хеширования сразу указывает страницу, на которой находится запись, поэтому сложность (и время) поиска в таком индексе не зависит от количества записей в нем. Однако сложность поиска зависит от количества записей переполнения: чем их больше, тем больше времени понадобится для поиска.

Статические варианты индексов на основе хеширования предполагают, что адресное пространство (и, следовательно, память, выделяемая для индекса) должны быть определены при создании индекса. При малой заполненности количество записей переполнения оказывается небольшим, но плохо используется выделенная память. С ростом количества записей увеличивается и количество записей переполнения, поэтому возрастает время поиска и требуется реорганизация индекса (при которой содержимое полностью переносится в новую область памяти большего размера).

Известны методы доступа на основе хеширования, допускающие динамическое расширение адресного пространства: расширяемое хеширование [24] и

линейное хеширование [43]. В системе PostgreSQL применяется вариант расширяемого хеширования.

Метод расширяемого хеширования не требует предварительного выделения памяти, блоки могут добавляться по мере роста объема данных. Кроме блоков памяти для хранения данных, для работы метода необходима небольшая таблица переадресации, которая хранится в оперативной памяти. Пусть в некотором состоянии размер файла составляет N блоков, тогда размер этой таблицы составляет 2^k позиций, где k такое, что $2^{k-1} < N \leqslant 2^k$. Другими словами, k равно количеству битов, необходимых для записи текущего размера файла, выраженного количеством блоков данных. Каждая позиция таблицы, определяемая k младшими битами функции хеширования, содержит адрес некоторого блока данных.

Если при добавлении новой записи оказывается, что для нее нет места в блоке, на который указывает таблица переадресации, то выполняется расщепление этого блока. Процедура расщепления использует дополнительный бит функции хеширования, для того чтобы распределить записи, находившиеся в расщепляемом блоке, между двумя блоками, и вносит соответствующие изменения в таблицу переадресации. При этом может потребоваться увеличение размеров таблицы переадресации в два раза.

В начальном состоянии $k = 0$, файл содержит один блок, а таблица переадресации — одну позицию, содержащую адрес этого блока. Все объекты данных, независимо от значения функции хеширования, размещаются в этом блоке. После первого расщепления $k = 1$, таблица переадресации содержит $2^k = 2$ позиции, в которых хранятся адреса двух блоков данных. Следующее расщепление также приводит к удвоению размеров таблицы, после этого $k = 2$. На рис. 11.2.3 показано, как при этом меняется таблица переадресации.

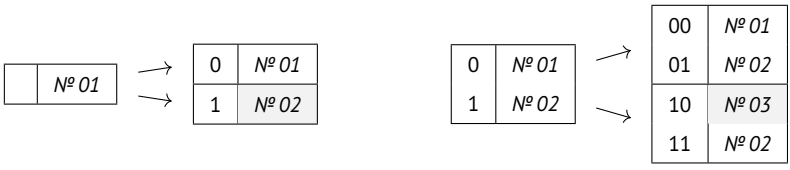

Рис. 11.2.3. Расширяемое хеширование, первые два расщепления

Конечно, в реализации хранить значения функции хеширования, показанные в левой колонке, не нужно: на рисунке они приводятся только для пояснения. Цветом на этом рисунке помечены адреса блоков, состояние которых изменено процедурой расщепления.

Если при дальнейшем добавлении записей переполнится блок № 01 (или блок № 03), то произойдет новое удвоение размеров таблицы. Если же переполнится блок, адрес которого содержится в нескольких позициях таблицы переадресации, то удвоение таблицы не понадобится: достаточно только изменить адрес в одной из позиций, содержащих адрес этого блока.

На рис. 11.2.4 показаны эти дальнейшие изменения таблицы переадресации: сначала расщепляется блок № 03 (с удвоением таблицы), затем — блок № 01 (удвоения таблицы не происходит).

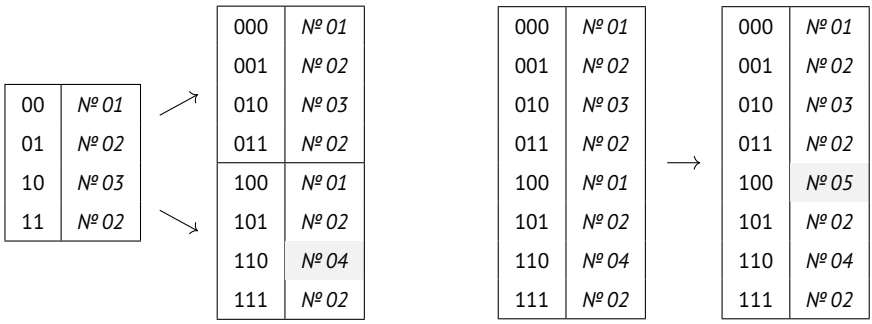

Рис. 11.2.4. Расширяемое хеширование, следующие два расщепления

Ожидаемый коэффициент заполнения блоков данных этого файла, как и для B-деревьев (и по тем же вероятностным причинам) составляет $\ln 2 \approx 0{,}69$. Существуют многочисленные разновидности метода расширяемого хеширования, позволяющие ограничить рост таблицы переадресации или увеличить заполненность файла.

Метод расширяемого хеширования гарантирует доступ к любой записи за одно обращение к блокам данных, записи переполнения не возникают. Однако этот метод чувствителен к качеству функции хеширования: при неравномерном распределении данных по блокам таблица переадресации растет слишком быстро.

В системе PostgreSQL хеширование используется как для построения индексов, так и для выполнения алгебраических операций соединения, группировки и устранения дубликатов.

11.2.2. Пространственные индексы

В отличие от одномерных многомерные индексные структуры характеризуются тем, что ключ индексной записи состоит из нескольких полей, не связанных отношением порядка. Критерии поиска могут задаваться для любого непустого подмножества ключей. Одним из распространенных классов объектов, для которых целесообразно применение многомерных индексов, являются пространственные объекты. Например, для множества точек на плоскости или в трехмерном пространстве компонентами ключа являются координаты точек.

Конечно, совсем не обязательно, чтобы поля многомерного ключа были вещественными числами (как в случае координат), — можно использовать любые скалярные типы данных, имеющие естественное упорядочение. Необходимо также, чтобы соответствующий домен содержал достаточно много различных значений, и требуется, чтобы в этом домене существовала метрика, т. е. способ вычисления расстояния между значениями.

Все многомерные индексные структуры используют разбиение множества ключей на подмножества, содержащие ключи, расположенные в определенном смысле близко друг от друга. Обычно такие подмножества организуются в некоторое дерево.

Наиболее известной, реализованной во многих СУБД и широко применяемой многомерной индексной структурой является R-дерево (R — rectangle). Эта структура предназначена для индексирования точек или прямоугольников со сторонами, параллельными осям координат (в пространствах размерности больше 2 вместо прямоугольников используются параллелепипеды). Если необходимо индексировать объекты другой формы, то вокруг них описывается прямоугольник нужной ориентации.

Прямоугольники (параллелепипеды) такого вида полностью определяются координатами концов самой большой диагонали (скажем, такой, у которой по каждой размерности значение координаты в начальной точке меньше, чем в конечной). Координаты этой пары точек являются полями индексного ключа. Таким образом, для пространства размерности k будет использовано $2k$ полей ключа.

Каждая вершина дерева представляется блоком, содержащим индексные записи, характеризующие прямоугольники, и сама описывается прямоугольником, содержащим все прямоугольники, лежащие внутри блока. Структура в целом

представляет собой сбалансированное дерево, в котором блоки, расположенные в листьях, содержат прямоугольники индексируемых объектов данных. Нелистовые вершины содержат прямоугольники, описывающие вершины следующего уровня вместе со ссылками на эти вершины.

Структура примерного R-дерева проиллюстрирована на рис. 11.2.5.

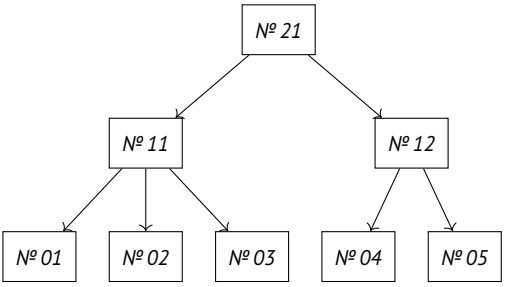

Рис. 11.2.5. Структура R-дерева

На рис. 11.2.6 для того же R-дерева показано расположение объектов (представленных геометрическими фигурами) и охватывающих прямоугольников на плоскости. Так, листовая страница № 01 содержит записи, относящиеся к объектам 1, 2 и 3, страница № 12 (на уровне 1) содержит записи, описывающие прямоугольники страниц № 04 и № 05, а корневая вершина дерева содержит описания прямоугольников страниц № 11 и № 12.

Запрос на поиск в R-дереве представляется прямоугольником такого же вида. Результатом выполнения запроса является множество объектов, прямоугольники которых имеют непустое пересечение с прямоугольником запроса. На рис. 11.2.6 запрос изображен прямоугольником, стороны которого показаны пунктиром.

Работа алгоритма поиска начинается с корневой вершины, из которой выбираются все прямоугольники, пересекающиеся с прямоугольником запроса. Далее для каждого из выбранных прямоугольников выполняется аналогичный поиск на следующих уровнях вплоть до листьев дерева. В отличие от одномерного случая (B-деревьев) спуск выполняется не по одному пути, а по нескольким.

Подчеркнем, что индексируемые объекты могут иметь любую форму или могут быть точечными (в этом случае их охватывающие прямоугольники также вырождаются в точку), однако ключ в индексной структуре R-дерева всегда будет прямоугольником. Это приводит к тому, что поиск в R-дереве может возвращать объекты, которые не должны входить в результат запроса, в тех случаях,

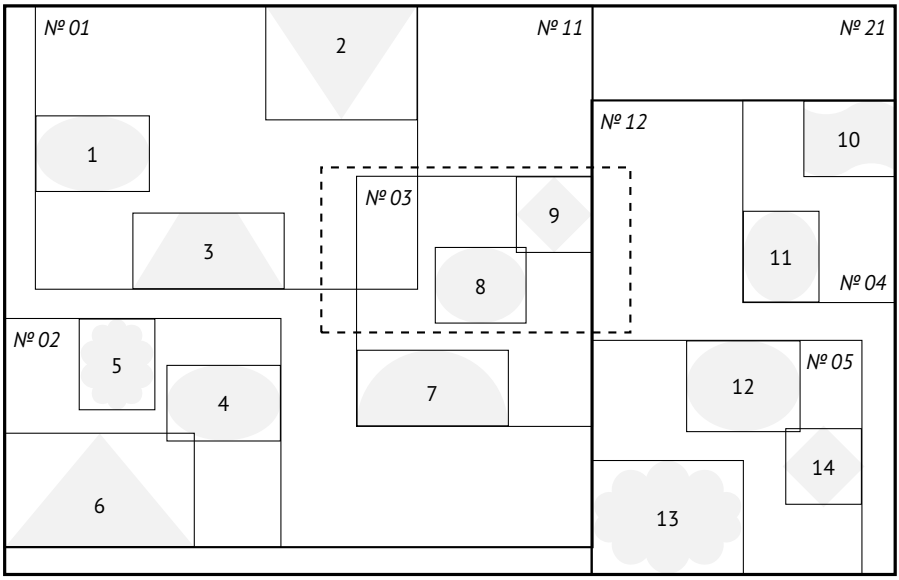

Рис. 11.2.6. Расположение объектов R-дерева

когда сам объект не пересекается с запросом, но охватывающий его прямоугольник — пересекается. Аналогично, запрос на поиск объектов в некоторой непрямоугольной области заменяется на прямоугольный запрос, охватывающий эту область, что также может вызвать включение лишних объектов в результат. Важный пример таких запросов — поиск объектов, находящихся не дальше, чем на заданном расстоянии от указанной точки. Этот запрос представляется кругом, но для поиска в R-дереве круг заменяется на описанный вокруг него квадрат.

Для того чтобы добавить новый объект, выполняется процедура поиска с охватывающим прямоугольником этого объекта в качестве прямоугольника запроса, причем поиск останавливается на предпоследнем уровне (т. е. выполняется поиск блоков, а не объектов). Прямоугольники разных блоков могут пересекаться, поэтому в результате может быть найдено несколько блоков. Поскольку охватывающие прямоугольники блоков на всех уровнях, кроме корневого, обычно не покрывают все пространство. может оказаться, что новый объект не пересекается ни с одним из имеющихся блоков. В любом случае процедура вставки выбирает один из блоков и, если объект не помещается в охватывающий прямоугольник полностью, то охватывающий прямоугольник блока расширяется.

Для того чтобы R-дерево не вырождалось при модификациях, задаются минимальное и максимальное количество записей в каждой вершине (кроме корневой, в которой ограничение снизу не применяется). Если при добавлении нового объекта в вершине нет места, выполняется процедура расщепления, которая распределяет объекты из старой вершины между двумя новыми, затем удаляет старую запись и добавляет две новые в блок более высокого уровня. Этот блок в случае необходимости также расщепляется, а при расщеплении корня добавляется новый уровень дерева.

В отличие от одномерного случая (B-дерево), в котором процедура расщепления переносит в новый блок примерно половину записей с большими ключами, расщепление для R-деревьев весьма нетривиально. Известны различные эвристические критерии качества полученного разбиения, в том числе:

- минимальный суммарный периметр;
- минимальная суммарная площадь;
- минимальная площадь пересечения;
- минимальная площадь, не занятая объектами следующего уровня.

В течение более чем 30 лет развития было предложено большое количество алгоритмов расщепления для разнообразных модификаций R-дерева, различающихся по вычислительной сложности и по качеству получаемого результата.

Также как B-деревья, R-деревья являются динамической индексной структурой при условии, что применяется процедура слияния недостаточно заполненных блоков. Если же, как в большинстве реализаций, слияние не применяется, то R-дерево может деградировать. Кроме возникновения недостаточно заполненных блоков может возникать и другой вид деградации: охватывающие прямоугольники, занимающие бóльшую область пространства, чем необходимо (эта ситуация может возникать при удалении объекта, находящегося на краю охватывающего прямоугольника блока, содержавшего этот объект).

Упомянем один из дополнительных эвристических приемов, позволяющих улучшить структуру R-дерева. Идея повторной принудительной вставки (forced re-insert) состоит в том, что объект, отстоящий слишком далеко от других объектов, находящихся в том же блоке (и, следовательно, сильно увеличивающий размеры охватывающего прямоугольника) удаляется из этого блока и выполняется его повторная вставка, при которой он, возможно, попадет в другой блок. Неформальное объяснение этого приема состоит в том, что на начальных фазах заполнения дерева, когда объектов (и, следовательно, блоков) немного,

объекты могут находиться на относительно большом расстоянии друг от друга, и поэтому необходимы охватывающие прямоугольники большого размера. С ростом дерева количество блоков и плотность размещения объектов в пространстве увеличиваются, поэтому размеры прямоугольников могут уменьшаться.

Индексы на основе R-деревьев хорошо работают с объектами небольшого размера (или точечными) в пространствах небольшой размерности, однако могут оказаться малополезными по одной или нескольким из следующих причин:

- Охватывающие прямоугольники индексируемых объектов имеют большие пересечения. В этом случае никакая структура, использующая прямоугольники в качестве поисковых образов, не сможет различить такие объекты.
- Объекты большого размера, занимающие малую часть площади охватывающего прямоугольника (например, участки дорог). В этом случае индекс будет возвращать большое количество объектов, на самом деле не пересекающихся с поисковым прямоугольником.
- В задачах поиска объектов, находящихся на расстоянии, не превышающем заданное, естественным поисковым запросом является круг, а не прямоугольник, поэтому результат применения индекса требует дополнительной фильтрации.

Последняя из перечисленных причин становится решающей при увеличении размерности пространства. Дело в том, что отношение объема гиперсферы к объему охватывающего ее гиперкуба очень быстро падает с ростом размерности, соответственно, возрастает доля нерелевантных объектов, возвращенных индексом. Известно, что любая индексная структура на основе дерева или любого разбиения пространства деградирует с ростом размерности, так что поиск становится более медленным, чем полный просмотр всей коллекции объектов. Этот факт образно называют «проклятьем размерности». В литературе описаны структуры, которые могут работать в бóльших размерностях, чем R-деревья, однако ни одна из известных структур такого типа не может эффективно работать в пространствах, размерности которых измеряются сотнями.

Имеются также структуры, позволяющие выполнить запрос приближенно (т. е. некоторые объекты, удовлетворяющие условию поиска, могут быть не найдены с помощью такого индекса). Такие структуры, однако, не реализуются в составе систем управления базами данных.

11.2.3. Инвертированные индексные структуры

Во многих классах задач объекты, среди которых выполняется поиск, характеризуются набором поисковых признаков (ключей), причем размер этого набора не фиксирован, т. е. разные объекты могут иметь различное количество ключей. Для поиска таких объектов используются *инвертированные индексы*. Наиболее широко известные применения этой структуры связаны с задачами поиска текстов на естественных языках, но, конечно, возможные применения этим не ограничиваются. Можно, например, использовать эту структуру для ускорения поиска множеств по заданному подмножеству, а также для решения некоторых задач на графах. Первые структуры данных такого типа применялись еще в конце 50-х гг. и получили название *инвертированных файлов*.

Описывать структуру инвертированного индекса удобнее всего на упрощенном примере индексирования текстовых документов.

В задачах поиска текстов запросы строятся на основе отдельных слов, содержащихся в тексте, их комбинаций или наборов, но не текста целиком. Структура прямого и инвертированного файлов представлена на рис. 11.2.7.

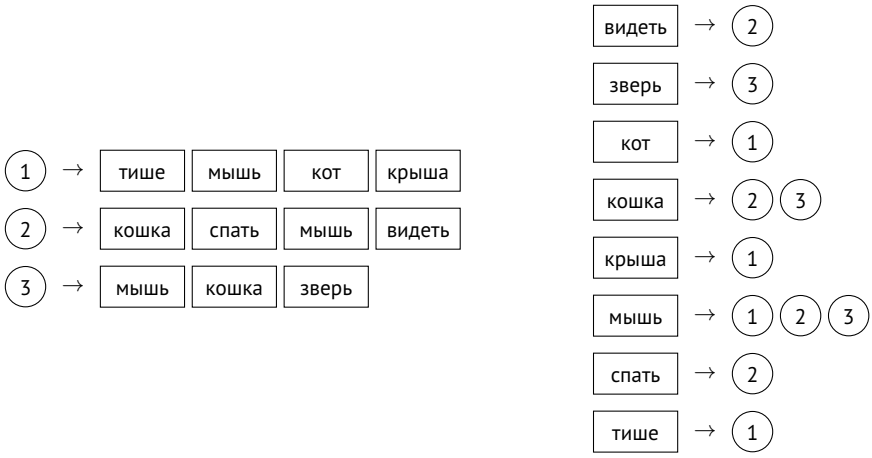

Рис. 11.2.7. Прямой и инвертированный файлы

На этом рисунке числа в кружках обозначают идентификаторы (номера) документов, а слова в прямоугольниках — идентификаторы поисковых термов (слов, характеризующих документ для поиска). В ранних системах в качестве поисковых термов могли использоваться только ключевые слова или слова,

входящие в аннотацию, а не все слова, составляющие полный текст документа. Прямой файл, таким образом, сопоставляет каждому документу последовательность слов, характеризующих этот документ, а инвертированный — наоборот, позволяет по слову найти список документов, в которых встречается это слово, т. е. является обратным (обращенным, инвертированным) по отношению к прямому файлу. Конечно, инвертированный файл не является обратной функцией в математическом смысле.

Списки документов в подобных структурах принято называть *пост-листами*. Заметим, что порядок слов в прямом файле может отражать порядок этих слов в документе, а относительное расположение номеров документов в пост-листах не может иметь никакой семантической нагрузки, поэтому пост-листы обычно упорядочиваются по возрастанию номеров документов.

Запрос, содержащий набор слов, обрабатывается следующим образом: из инвертированного файла считываются пост-листы, соответствующие этим словам, и затем вычисляется их пересечение. Поскольку все пост-листы упорядочены одинаково, для вычисления пересечения можно использовать алгоритм слияния, что требует однократного прохода по пост-листам. Для более сложных запросов может понадобиться вычисление объединения или разности, однако в любом случае алгоритм остается однопроходным.

Большое количество поисковых термов в запросе требует обработки значительного числа пост-листов. Поскольку теоретико-множественные операции обладают свойствами коммутативности и ассоциативности, обычно применяются эвристические алгоритмы для определения последовательности, в которой обрабатываются пост-листы. Например, для вычисления пересечения может быть целесообразно начинать с самых коротких пост-листов, чтобы как можно быстрее сократить размер промежуточного результата.

В литераторе описано большое количество вариантов и модификаций индексных структур на основе инвертированных файлов. Для текстовых документов на естественном языке построение инвертированных файлов из слов, непосредственно содержащихся в тексте, не дает удовлетворительные результаты по качеству поиска. Поэтому обычно слова, извлеченные из документов, подвергаются различным видам лексической обработки: приведению к канонической форме, выделению основы и т. п.

Реализация текстового поиска в PostgreSQL содержит для этого ряд библиотечных функций для лингвистической обработки. Более детально методы полнотекстового поиска и средства их реализации, включенные в состав PostgreSQL, обсуждаются в главе 18.

11.2.4. Разреженные индексы

Индекс называется *плотным*, если он содержит ссылки на отдельные объекты индексируемой коллекции. Все индексы, рассмотренные выше, являются плотными.

Хранение коллекции может быть организовано таким образом, что объекты, содержащие совпадающие или близкие значения некоторого атрибута (или группы атрибутов), размещаются рядом, на одной или нескольких соседних страницах, выделенных для хранения коллекции. В таких случаях в индексной записи вместо ссылок на отдельные объекты достаточно поместить только ссылку на область памяти, содержащую объекты с ключом, включенным в индекс. Такие индексы называются *неплотными* или *разреженными*. Как правило, разреженный индекс содержит одно значение ключа для каждой страницы данных.

11.2.5. Сигнатурные индексы

Рассматриваемая в этом подразделе индексная структура предназначена для ускорения *запросов включения* (containment queries). Предполагается, что значения некоторого атрибута таблицы представляют собой конечные множества. Например, если строка таблицы описывает текстовый документ, то значением этого атрибута может быть множество слов, встречающихся в документе.

Предполагается также, что все значения этого атрибута являются подмножествами некоторого универсального множества U. Для текстовых документов таким множеством будет множество всех слов, которые могут встречаться в документах.

Пусть задано некоторое множество $Q \subset U$. Любое такое множество определяет запрос включения, результат выполнения которого состоит из строк, в которых значение рассматриваемого атрибута содержит Q (как подмножество). Если запрос представляет собой множество слов, то ответом на этот запрос будет набор всех документов, каждый из которых содержит все слова, включенные в запрос.

Для того чтобы проверить, что множество A входит в результат выполнения запроса Q, нужно вычислить теоретико-множественную разность $Q \setminus A$. Множество A удовлетворяет запросу тогда и только тогда, когда эта разность является

пустым множеством. Чтобы не выполнять (ресурсоемкое) вычисление разности для каждого значения нашего атрибута, строится *сигнатурный индекс*.

Сигнатурой множества A называется битовая карта фиксированной длины $s(A)$, которая строится таким образом, что если два множества связаны отношением включения $A \subset B$, то $s(A) \leqslant s(B)$ в каждом бите сигнатуры. Другими словами, если сигнатура $s(B)$ в некоторой позиции содержит 0, то $s(A)$ тоже должна содержать 0 в этой позиции.

Сигнатура множества вычисляется побитовой логической операцией OR, применяемой к сигнатурам всех элементов множества (например, слов документа), а сигнатуры элементов множества вычисляются с помощью функции хеширования.

Для того чтобы метод хорошо работал, функция хеширования должна вырабатывать сигнатуру, содержащую малое количество ненулевых битов (иначе в результате применения операции OR сигнатуры множеств будут содержать слишком много единичных битов и поэтому множества будут плохо различимы). Обычно количество ненулевых битов задается заранее, а функция хеширования вырабатывает номера битов, которые устанавливаются в единицу. В реализации сигнатурных индексов в системе PostgreSQL сигнатуры элементов содержат ровно один ненулевой бит.

Сигнатурный индекс в системе PostgreSQL содержит все сигнатуры индексируемого атрибута, организованные в дерево с помощью обобщенной индексной структуры GiST. Результатом поиска в таком индексе является набор строк таблицы, для которых $s(Q) \leqslant s(A)$ (для каждого бита сигнатуры).

При фильтрации по сигнатурному индексу не гарантируется включение $Q \subset A$, потому что разные множества могут отображаться в одинаковые сигнатуры. Поэтому после применения индекса обычно необходима дополнительная проверка того, что включение множеств действительно имеет место. Проверка не требуется, если каждому биту сигнатуры соответствует не более одного слова; ее необходимость определяется в PostgreSQL автоматически.

Если функция хеширования элементов (слов) вырабатывает, как в PostgreSQL, сигнатуру с одним единичным битом, то разные элементы могут отображаться в одинаковые сигнатуры. Если количество различных элементов велико, то такие коллизии будут частыми, и поэтому индекс будет давать большое количество ложных кандидатов.

Если функция хеширования вырабатывает больше одного ненулевого бита, то ложные кандидаты могут появляться также при наличии общих единичных битов в сигнатурах разных слов. Например, допустим, что слово «кошка» имеет сигнатуру 1001, слово «мышь» — сигнатуру 0101, а слово «собака» — сигнатуру 1100. Тогда документ, описывающий отношения кошек и мышей, будет иметь сигнатуру 1101 и будет найден по запросу «собака».

Доля ложных срабатываний может оказаться очень большой, что ограничивает применимость сигнатурных индексов. Во многих ситуациях применение инвертированных индексов на основе GIN может оказаться предпочтительным.

11.2.6. Особенности реализации индексов в PostgreSQL

Реализация индексов в системе PostgreSQL решает две задачи: с одной стороны, эта реализация обеспечивает высокоэффективное выполнение запросов и учет особенностей отдельных типов индексов при выборе плана выполнения запроса и, с другой стороны, обеспечивает возможность расширения набора функций системы путем реализации новых индексных структур. Для достижения этих трудно совместимых целей потребовались инженерные решения, включающие многоуровневую систему моделей и понятий.

С точки зрения исполнителя запросов применение любого индекса сводится к получению адреса или набора адресов строк, удовлетворяющих критерию поиска, переданному в индексную структуру. Адреса строк могут использоваться для выборки данных непосредственно из таблиц или для построения битовых карт, но в любом случае внутренняя структура индекса не имеет никакого значения. Битовые карты полезны при выполнении запросов, в которых выбирается значительная доля кортежей, а также в тех случаях, когда целесообразно применение нескольких различных индексов.

Индексы могут также использоваться для выборки значений индексных ключей — в этом случае значения можно взять из самого индекса, не переходя к таблице. Однако в системе PostgreSQL применяется хранение множественных версий строк таблиц, поэтому при выборке значений из индексов, вообще говоря, требуется дополнительная проверка того, что эти значения принадлежат версиям строк таблицы, которые видны текущей транзакции. Проверка видимости всех выбираемых значений сделала бы бессмысленным такое

использование индексов. Для того чтобы сократить затраты на эти проверки, в PostgreSQL используется вспомогательная структура данных (карта видимости), которая позволяет обойтись без проверок для строк, размещенных на страницах, на которых ни одна из строк не обновлялась в течение достаточно длительного времени, и поэтому все строки на таких страницах видны всем транзакциям.

Внутренняя организация индекса определяется методом доступа, отвечающим за поддержку структуры хранения, а также выдающим информацию о свойствах и стоимости выполнения операций (в основном — поиска) в этом индексе. Эта информация необходима для выбора оптимального плана выполнения запроса. Кроме этого, метод доступа отвечает за реализацию транзакционных свойств индекса, обеспечивающих корректность работы при одновременном доступе к индексу нескольких транзакций.

Реализация новых методов доступа является, таким образом, достаточно сложной задачей. Для того чтобы упростить ее решение, в PostgreSQL реализованы обобщенные методы доступа.

Возможность добавления новых типов методов доступа на основе деревьев обеспечивается в PostgreSQL механизмами GiST и SP-GiST. По существу, эти средства обеспечивают поддержку структур типа дерева, согласованных с другими частями PostgreSQL: управлением памятью, транзакционностью и пр. Отличие GiST от SP-GiST в том, что GiST поддерживает сбалансированные деревья (такие, как R-деревья), а SP-GiST позволяет строить несбалансированные деревья (такие, как k-d-деревья или k-d-B-деревья [52]). GiST также обеспечивает выполнение запросов на поиск ближайших соседей (k-nn, k-nearest neighbours).

Обобщенными методами доступа являются также GIN, обеспечивающий построение инвертированных индексов (раздел 11.2.3). Как и в случае обобщенных индексов GiST и SP-GiST, создание нового типа индекса сводится к разработке ряда функций, реализующих операции, зависящие от индексируемого типа данных.

Создание типов неплотных индексов обеспечивается обобщенным методом доступа BRIN, входящей в состав PostgreSQL. Также, как для GiST, SP-GiST и GIN, реализация типа индекса на основе BRIN сводится к разработке нескольких функций, определяющих способы выполнения операций поиска и модификации индекса для конкретного типа данных.

Наконец, метод доступа может быть применен для различных типов данных, в том числе определенных пользователем. Для этого описываются функции,

представляющие семантику типа данных, подлежащего индексированию. Набор таких функций, связывающих метод доступа с типом данных, называется *классом операторов*. Так, для того чтобы обеспечить работу индексов на основе B-деревьев, необходимо определить операторы проверки на равенство и упорядочение значений типа данных, подлежащего индексированию.

Особенности организации и применения индексов в системе PostgreSQL обсуждаются в [69]. Возможности и применение обобщенных индексных структур в PostgreSQL рассматриваются в главе 17.

11.3. Выполнение алгебраических операций

11.3.1. Алгебраические операции и алгоритмы

Выполнение любого запроса начинается с операций выборки хранимых данных. Поскольку реляционная модель данных не занимается вопросами хранения, эти операции не могут считаться частью реляционной алгебры, однако обычно при их выполнении учитываются условия, которым должны удовлетворять данные, необходимые для выполнения запроса.

Реляционная алгебра и в особенности алгебра SQL предусматривает большой ассортимент бинарных операций (т. е. операций с двумя аргументами), однако для их выполнения используются модификации одних и тех же основных алгоритмов. В этом разделе рассматриваются три алгоритма просмотра отношений, которые применимы для вычисления:

- декартова произведения;
- внутреннего и внешних соединений;
- теоретико-множественной разности;
- пересечения.

Модификации тех же алгоритмов можно применять для операций группировки и устранения дубликатов. Эти операции не являются в точном смысле бинарными, но для их выполнения требуется сопоставлять разные кортежи из аргумента. Поэтому для них применимы те же алгоритмы, в которых в качестве второго аргумента используется частично вычисленный результат операции.

Отметим, что для всех алгоритмов бинарных операций, рассматриваемых в этом разделе, существуют эффективные параллельные версии, которые обсуждаются в главе 22.

11.3.2. Операции выборки данных

Основной операцией доступа к коллекциям является фильтрация по некоторому условию на атрибуты. При отсутствии такого условия результатом выполнения операции является множество всех элементов коллекции. Во многих руководствах эта операция называется *селекцией* (например, в литературе по теоретической реляционной модели эта операция обычно обозначается буквой σ). Однако термин «селекция» может вызвать некорректные ассоциации с ключевым словом SELECT языка SQL. Наиболее точно эта операция может быть охарактеризована выражением «выборка данных, возможно, по условию».

Здесь мы рассматриваем фильтрацию применительно к коллекциям, хранимым в базе данных. Для фильтрации промежуточных результатов выполнения запроса применяются другие алгоритмы. В зависимости от выбранного метода хранения и условий фильтрации могут быть применимы некоторые из перечисленных ниже алгоритмов. Не все алгоритмы применимы для любой ситуации, не все алгоритмы реализованы и используются во всех СУБД, и в некоторых СУБД могут использоваться и другие алгоритмы.

Полный просмотр. Последовательно считываются все страницы, выделенные для коллекции, с проверкой условия фильтрации для всех объектов на каждой странице. Пример плана запроса, использующего полный просмотр, в системе PostgreSQL:

```
demo=# EXPLAIN (costs off)
SELECT * FROM flights;
     QUERY PLAN
--------------------
 Seq Scan on flights
```

Доступ по адресу. Для каждого адреса объекта, полученного из индекса по критерию фильтрации, выбирается страница, содержащая объект, и, возможно, выполняется просмотр всех объектов на этой странице для выборки других объектов, удовлетворяющих условию фильтрации.

```
demo=# EXPLAIN (costs off)
SELECT * FROM flights
WHERE flight_id = 12345;
```

```
                QUERY PLAN
-----------------------------------------
 Index Scan using flights_pkey on flights
   Index Cond: (flight_id = 12345)
```

Просмотр страницы нужен, для того чтобы исключить повторный доступ к той же странице (например, если условие, для которого применяется индекс, задает диапазон значений). Просмотр страницы также необходим при использовании неплотных индексов, которые локализуют блоки, содержащие искомые записи, а не отдельные записи. В системе PostgreSQL эти задачи решаются другим способом с применением битовых карт, которые строятся при просмотре индексов:

```
demo=# EXPLAIN (costs off)
SELECT * FROM boarding_passes
WHERE ticket_no = '0005435212351';
                        QUERY PLAN
-----------------------------------------------------------
 Bitmap Heap Scan on boarding_passes
   Recheck Cond: (ticket_no = '0005435212351'::bpchar)
   ->  Bitmap Index Scan on boarding_passes_pkey
         Index Cond: (ticket_no = '0005435212351'::bpchar)
```

Поскольку в системе PostgreSQL на странице могут находиться несколько версий одного объекта, при доступе по адресу выполняются дополнительные проверки, чтобы выбирались только корректные версии.

Доступ по ключу. Если размещение учитывает значения атрибута или атрибутов, входящих в условия фильтрации, то доступ к нужной странице возможен без использования индекса, построенного для этой таблицы.

Просмотр интервала. Если объекты размещены в соответствии с упорядоченностью атрибута, входящего в условие фильтрации, для выборки требуемых объектов достаточно выполнить просмотр только последовательно расположенных страниц, содержащих требуемые значения атрибута.

Выбор данных (только) из индекса. Выбор данных из индекса вместо чтения таблицы возможен в том случае, если индекс содержит все значения, которые должны быть выбраны из таблицы для вычисления результата запроса. Этот метод может применяться в различных вариантах:

- выборка данных по значению ключа;
- выборка диапазона значений;
- полный просмотр индекса.

Наиболее широко известный случай применения этого метода состоит в использовании составного индекса, содержащего все необходимые колонки (такой индекс называется *покрывающим*). Этот же метод может применяться и в других случаях, например для вычисления агрегатных функций (count, min, max). Напомним, что в системе PostgreSQL индексы содержат информацию обо всех версиях объекта, поэтому для получения корректных результатов требуется использовать карту видимости и в некоторых случаях обращаться к страницам данных.

Далеко не каждый тип индекса из большого разнообразия типов, имеющихся в системе PostgreSQL, пригоден для использования в этой группе алгоритмов. Пример плана запроса для индекса на основе B-дерева:

```
demo=# EXPLAIN (costs off)
SELECT ticket_no, flight_id FROM boarding_passes
WHERE ticket_no = '0005435212351';
                          QUERY PLAN
---------------------------------------------------------------
 Index Only Scan using boarding_passes_pkey on boarding_passes
   Index Cond: (ticket_no = '0005435212351'::bpchar)
```

Если при размещении объектов использована горизонтальная фрагментация, т. е. таблица хранится в виде нескольких секций (partitions), то структура хранения и алгоритм выбираются отдельно для каждой секции.

Вертикальная фрагментация, т. е. разнесение групп атрибутов по нескольким таблицам, не поддерживается явным образом в языке SQL, поскольку для этого достаточно использовать несколько таблиц (скорее всего, с одинаковыми первичными ключами).

11.3.3. Сортировка

Несмотря на то что в теоретической реляционной модели данных упорядочение кортежей в отношениях не имеет никакого значения, операции сортировки являются неотъемлемой частью любой системы управления базами данных. Необходимость упорядочивания (сортировки) данных возникает по нескольким причинам.

- Последовательность, в которой выводятся строки результата, может быть важна для приложения (и для пользователя). В языке SQL упорядочение результата запроса задается предложением ORDER BY.

- Некоторые алгоритмы выполнения алгебраических операций основаны на предположении об упорядоченности аргументов, в частности это алгоритмы на основе слияния, обсуждаемые ниже.
- Сортировка может быть необходима для построения некоторых структур хранения.

Алгоритмы сортировки можно подразделить на две группы:

Алгоритмы внутренней сортировки упорядочивают коллекции, полностью размещенные в оперативной памяти. К этой группе относятся, в частности, пирамидальная сортировка (heap sort) и быстрая сортировка (quicksort).

```
demo=# EXPLAIN (analyze, costs off, timing off)
SELECT * FROM seats
ORDER BY seat_no;
                      QUERY PLAN
-----------------------------------------------------
 Sort (actual rows=1339 loops=1)
   Sort Key: seat_no
   Sort Method: quicksort  Memory: 111kB
   ->  Seq Scan on seats (actual rows=1339 loops=1)
```

Алгоритмы внешней сортировки упорядочивают данные, которые размещены во внешней памяти (на дисках). Наиболее часто применяемым алгоритмом этой группы является алгоритм на основе слияния.

```
demo=# EXPLAIN (analyze, costs off, timing off)
SELECT * FROM bookings
ORDER BY book_date;
                        QUERY PLAN
-----------------------------------------------------------
 Sort (actual rows=262788 loops=1)
   Sort Key: book_date
   Sort Method: external merge  Disk: 8480kB
   ->  Seq Scan on bookings (actual rows=262788 loops=1)
```

Можно доказать, что вычислительная сложность задачи сортировки не может быть ниже чем $O(N \log N)$, где N — количество сортируемых объектов. Упомянутые выше алгоритмы имеют именно такую сложность. Это означает, что создать алгоритм, который работал бы существенно быстрее известных, невозможно.

Алгоритмы сортировки детально изучены в 1960–1970-х гг. и описаны практически во всех учебниках по программированию, алгоритмам и структурам данных, поэтому мы воздержимся от их изложения.

11.3.4. Алгоритм вложенных циклов

Простейший вариант алгоритма вложенных циклов показан на рис. 11.3.1 и может быть описан следующим псевдокодом:

```
FOR r IN R LOOP
    FOR s IN S LOOP
        обработать_кортежи(r,s);
    END LOOP;
END LOOP;
```

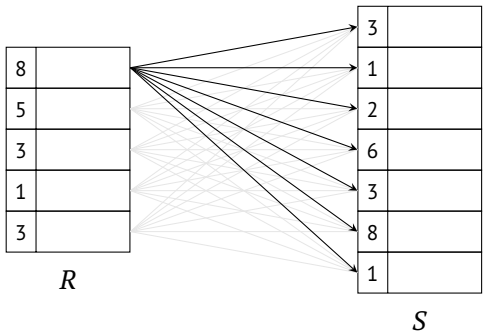

Рис. 11.3.1. Алгоритм вложенных циклов

Если функция обработать_кортежи записывает в результат кортеж, содержащий все атрибуты всех входных кортежей, то результатом этого алгоритма будет декартово произведение. Сложность этого алгоритма, очевидно, пропорциональна произведению кардинальностей множеств: $\mathrm{card}(R)\,\mathrm{card}(S)$. Важный факт состоит в том, что, поскольку размер результата декартова произведения равен произведению размеров аргументов, получить декартово произведение алгоритмом меньшей сложности невозможно.

Другие варианты функции обработать_кортежи реализуют другие алгебраические операции. Так, если эта функция выводит строку результата только в том случае, если выполняется некоторое условие, то будет получен результат операции соединения по этому условию. Если результат выводится только при полном совпадении входных строк, то будет вычислено теоретико-множественное пересечение, и т. п.

Если в качестве второго аргумента используется накопленный в ходе выполнения операции результат (начиная с пустого), то алгоритм вложенных циклов

можно использовать для удаления дубликатов из первого аргумента. Для этого необходимо, чтобы строка первого аргумента добавлялась в результат в том и только в том случае, если она не обнаружена при просмотре частично сформированного результата.

В любом случае, однако, сложность алгоритма остается пропорциональной произведению размеров аргументов, т. к. алгоритм будет просматривать все пары независимо от того, формируется из них строка результата или нет. Важно, однако, заметить, что, в отличие от вычисления декартова произведения, для других операций (соединения, теоретико-множественных, устранения дубликатов) существуют другие алгоритмы, обладающие лучшими теоретическими оценками сложности. Такие алгоритмы обсуждаются в следующих разделах этой главы. Другими словами, алгоритм вложенных циклов можно применять для реализации многих операций, но для операций, отличающихся от декартова произведения, этот алгоритм может быть (и часто оказывается) не оптимальным.

Можно, однако, сократить количество обращений к дискам, если считывать входные данные большими порциями. Приведенный далее псевдокод содержит не два вложенных цикла, а четыре, при этом два внешних цикла считывают в оперативную память страницы первого и второго аргументов операции, а внутренние два цикла представляют собой простой алгоритм, применяемый к части аргументов, находящейся (в результате работы внешних циклов) в оперативной памяти.

```
FOR Br IN страницы(R) LOOP
    FOR Bs IN страницы(S) LOOP
        FOR r IN Br LOOP
            FOR s IN Bs LOOP
                обработать_кортежи(r,s);
            END LOOP;
        END LOOP;
    END LOOP;
END LOOP;
```

В простейшем варианте алгоритма количество просмотров второго аргумента равно мощности первого аргумента $\mathrm{card}(R)$, а в блочном варианте — количеству повторений самого внешнего цикла. Чтобы сократить это количество, следует считывать сразу несколько блоков первого аргумента и выделить для этой цели максимально возможное количество оперативной памяти.

Такой вариант алгоритма вложенных циклов в PostgreSQL не применяется.

Наиболее важный вариант алгоритма вложенных циклов применим, если:

1) выполняется операция соединения, второй аргумент которой (*S*) является хранимым отношением, а не промежуточным результатом;

2) существует индекс по атрибуту соединения для этого аргумента.

В этом случае полный просмотр второго аргумента можно заменить на доступ через индекс, и внутренний цикл ограничивается просмотром только тех объектов из *S*, которые необходимы для вычисления результата соединения.

Можно охарактеризовать этот алгоритм другим способом. Как известно из теории, операция соединения эквивалентна операции вычисления декартова произведения и последующей фильтрации (селекции) по условию соединения. В общем случае алгоритм вложенных циклов буквально реализует это тождество: вложенные циклы по существу вычисляют декартово произведение, к которому «на лету» (по мере получения отдельных кортежей, входящих в прямое произведение) применяется фильтрующее условие.

Вместо этого можно на каждой итерации внешнего цикла (по первому аргументу операции соединения) применить операцию фильтрации ко второму аргументу. Предикат этой фильтрации получается из предиката соединения подстановкой значения атрибута из первого аргумента. Таким образом, предикат фильтрации для каждой итерации внешнего цикла получается разный и сравнивает значение атрибута второго аргумента с константой, полученной из первого. После выполнения такой фильтрации внутренний цикл применяется не ко всему второму аргументу, а только к результату фильтрации, т. е. к строкам, которые должны соединиться с текущей строкой первого аргумента.

Конечно, такое изменение алгоритма имеет смысл только в том случае, если для реализации операции фильтрации можно использовать индекс, а не полный просмотр, и, следовательно, второй аргумент является хранимым отношением, а не промежуточным результатом выполнения запроса. Очевидно, что применение алгоритма селекции на основе полного просмотра фактически означало бы проверку условий фильтрации одновременно с проверкой условий соединения, что по существу эквивалентно обычному алгоритму вложенных циклов.

В приведенном ниже примере вешний цикл выполняется по отношению aircrafts_data, которое просматривается полностью. Фильтрация второго аргумента соединения (отношения seats) выполняется для каждой строки первого аргумента с помощью индекса.

```
demo=# EXPLAIN (costs off)
SELECT *
FROM aircrafts_data a
   JOIN seats s ON a.aircraft_code = s.aircraft_code;
                      QUERY PLAN
------------------------------------------------------
 Nested Loop
   ->  Seq Scan on aircrafts_data a
   ->  Index Scan using seats_pkey on seats s
         Index Cond: (aircraft_code = a.aircraft_code)
```

Этот вариант алгоритма оказывается наиболее эффективным способом соединения, если размеры первого аргумента малы (возможно, после фильтрации). Однако при больших размерах аргументов другие алгоритмы соединения оказываются более эффективными, чем алгоритм вложенных циклов.

11.3.5. Алгоритм соединения на основе сортировки и слияния

Алгоритм на основе слияния применим для операций, в которых необходима проверка на равенство или неравенство (больше, меньше и т. п.) значений атрибутов входных кортежей. К таким операциям относятся соединение (с подходящими предикатами), группировка и удаление дубликатов, а также теоретико-множественные операции объединения, пересечения и разности.

Работа алгоритма состоит из двух фаз:

Сортировка. Входные отношения сортируются по атрибутам, по котором будет производиться проверка на равенство или неравенство (т. е. по атрибутам соединения, группировки или по всем атрибутам для удаления дубликатов). При этом объекты, атрибуты которых удовлетворяют предикату с одним и тем же значением второго аргумента предиката, окажутся после сортировки расположенными рядом.

Слияние. Выполняется вариант процедуры слияния, зависящий от выполняемой операции.

При выполнении соединения входные аргументы просматриваются в порядке возрастания атрибутов сортировки, при этом для групп объектов из первого и второго операндов, для которых условие соединения истинно, выполняется алгоритм вложенных циклов. Если размеры этих групп относительно малы по сравнению с размерами отношений, то выполнение этих вложенных циклов потребует значительно меньше ресурсов, чем полный алгоритм вложенных циклов. При этом дополнительная фильтрация внутри вложенных циклов не

потребуется, так как все обрабатываемые пары объектов должны быть включены в результат (потому что условие соединения для них выполняется). Корректность этого алгоритма гарантируется тем, что все объекты, которые должны быть соединены с некоторым объектов другого аргумента, оказываются расположенными рядом после сортировки.

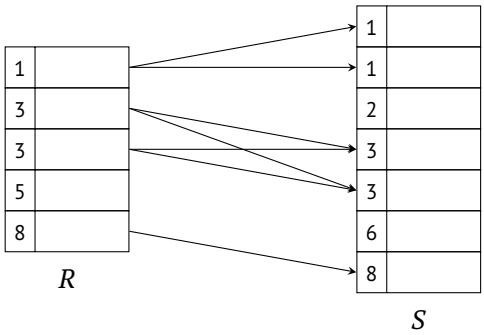

Рис. 11.3.2. Алгоритм соединения на основе слияния

Алгоритм слияния показан на рис. 11.3.2. Приведенный ниже псевдокод описывает частный случай алгоритма слияния, реализующего операцию внутреннего эквисоединения в предположениях о том, что:

1) кортежи аргументов (*R* и *S*) упорядочены по ключам соединения;

2) ключ соединения имеет уникальные значения во втором аргументе (*S*).

```
r := следующий(R);
s := следующий(S);
WHILE NOT последний(r) AND NOT последний(s) LOOP
    IF r < s THEN r := следующий(R);
    ELSIF s < r THEN s := следующий(S);
    ELSIF s = r THEN
        LOOP -- вложенный цикл
            обработать_кортежи(r,s);
            s := следующий(S);
            EXIT WHEN последний(s) OR s != r;
        END LOOP;
    END IF;
END LOOP;
```

Важная особенность алгоритма слияния состоит в том, что результат получается упорядоченным по значению ключа слияния.

В приведенном ниже примере плана запроса сортировка входных отношений выполняется в узлах Sort плана, слияние — в узле Merge Join, а сортировка результата соединения не требуется:

```
demo=# EXPLAIN (costs off)
SELECT *
FROM bookings b
   JOIN tickets t ON b.book_ref = t.book_ref
ORDER BY b.book_ref;
               QUERY PLAN
----------------------------------------
 Merge Join
   Merge Cond: (t.book_ref = b.book_ref)
   ->  Sort
         Sort Key: t.book_ref
         ->  Seq Scan on tickets t
   ->  Sort
         Sort Key: b.book_ref
         ->  Seq Scan on bookings b
```

Алгоритм на основе сортировки и слияния оказывается особенно эффективным, если один или оба входных отношения уже упорядочены так, как нужно для процедуры слияния — в этом случае фаза сортировки, естественно, исключается. Результат может оказаться отсортированным, в частности, в том случае, если предыдущая операция также была выполнена этим алгоритмом. Другой важный случай, в котором результат получается упорядоченным, — выборка диапазона значений из упорядоченного индекса.

```
demo=# EXPLAIN (costs off)
SELECT *
FROM bookings b
   JOIN tickets t ON b.book_ref = t.book_ref
ORDER BY b.book_ref;
                        QUERY PLAN
-----------------------------------------------------------
 Merge Join
   Merge Cond: (b.book_ref = t.book_ref)
   ->  Index Scan using bookings_pkey on bookings b
   ->  Index Scan using tickets_book_ref_idx on tickets t
```

Если алгоритм применяется для операций группировки или устранения дубликатов, для каждой группы объектов (из единственного в этом случае операнда) вырабатывается ровно один объект результата, поэтому вместо вложенных циклов достаточно выполнить только один цикл по группе объектов, имеющих одинаковые значения атрибутов группировки.

Теоретико-множественные операции объединения, пересечения и разности вообще не требуют вложенных циклов, поскольку решение о включении объек-

та в результат принимается на основе сравнения объектов из первого и второго аргументов (в предположении, что входные аргументы операций не содержат дубликатов).

11.3.6. Соединение на основе хеширования

Алгоритм на основе хеширования применим для тех же операций, для которых работает алгоритм на основе сортировки и слияния, с дополнительным условием на предикат соединения, который в этом случае должен быть условием равенства значений атрибутов. Другими словами, алгоритм применим для тех операций, где требуется проверка на совпадение значений всех или некоторых атрибутов входных отношений.

Базовый вариант алгоритма включает две фазы:

Распределение. Объекты первого аргумента размещаются в корзинах в соответствии со значениями функции хеширования.

Проверка. Объекты второго аргумента поочередно хешируются и сопоставляются со всеми объектами первого аргумента, хранящимися в соответствующей корзине. В случае совпадения значений атрибутов формируется объект результата.

Алгоритм можно описать следующим псевдокодом:

```
-- распределение
FOR r IN R LOOP
    B = номер_корзины(r);
    вставить r в корзину B хеш-таблицы;
END LOOP;

-- проверка
FOR s IN S LOOP
    B = номер_корзины(s);
    FOR b IN B LOOP
        обработать_кортежи(s,b);
    END LOOP;
END LOOP;
```

Выполнение любой версии алгоритма на основе хеширования включает применение функции хеширования к атрибутам входных объектов, подлежащих сравнению, распределение объектов по корзинам в соответствии со значениями функции хеширования и применение алгоритма вложенных циклов в каждой корзине. Корректность такого алгоритма гарантируется тем, что объекты,

попадающие в разные корзины, имеют разные значения функции хеширования и, следовательно, значения сравниваемых атрибутов также различаются. Поэтому объекты, попавшие в разные корзины, не могут образовать пару, подлежащую включению в результат.

Оценка сложности этого алгоритма включает однократный полный просмотр обоих аргументов для распределения по корзинам и сумму стоимостей применения алгоритма вложенных циклов в каждой корзине. При большом количестве корзин и достаточно равномерном распределении объектов по корзинам эта сложность будет существенно ниже, чем сложность алгоритма вложенных циклов.

Схема работы алгоритма хеширования представлена на рисунке 11.3.3.

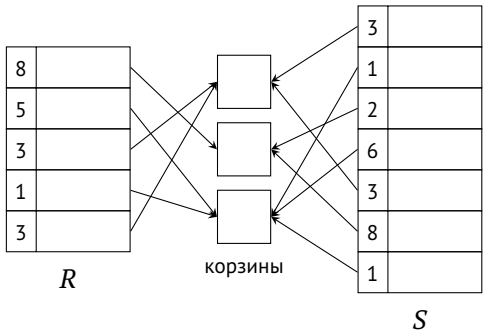

Рис. 11.3.3. Алгоритм соединения на основе хеширования

В приведенном ниже примере плана запроса, использующего соединение на основе хеширования, распределение выполняется в узле Hash плана, а проверка — в узле Hash Join:

```
demo=# EXPLAIN (costs off)
SELECT *
FROM bookings b
   JOIN tickets t ON b.book_ref = t.book_ref;
               QUERY PLAN
-----------------------------------------
 Hash Join
   Hash Cond: (t.book_ref = b.book_ref)
   ->  Seq Scan on tickets t
   ->  Hash
         ->  Seq Scan on bookings b
```

Более сложные варианты алгоритма предназначены для операций с таблицами очень большого размера.

Так, алгоритм гибридного соединения на основе хеширования сначала распределяет входные аргументы по корзинам и объединяет корзины в группы таким образом, чтобы каждая группа корзин могла быть одновременно загружена в оперативную память. Далее для каждой группы корзин выполняется обычный алгоритм соединения на основе хеширования. При этом второй аргумент можно либо предварительно распределить по группам, либо применить многократный просмотр второго аргумента (отдельный просмотр для каждой группы корзин).

Базовый алгоритм хеширования начинает вырабатывать результаты после полного считывания первого аргумента. В некоторых случаях такая задержка оказывается нежелательной. Симметричный алгоритм соединения на основе хеширования использует два набора корзин, по одному для каждого из аргументов. Считывание обоих входных аргументов начинается одновременно (поочередно или параллельно). К каждому прочитанному кортежу применяется функция хеширования, затем этот кортеж помещается в корзину своего аргумента и сопоставляется со всеми кортежами из соответствующей корзины другого аргумента.

Предположим, что пара кортежей *r, s* удовлетворяет предикат соединения, т. е. должна войти в результат. Если *r* прочитан раньше, чем *s,* то пара будет найдена при обработке *s,* потому что *r* уже записан в корзину первого аргумента. Если же *r* поступит позже, чем *s,* то пара будет найдена при просмотре корзины второго аргумента. Следовательно, этот алгоритм корректно строит результат операции соединения.

По сравнению с базовым алгоритмом симметричный требует значительно больше оперативной памяти (т. к. в корзинах накапливаются оба аргумента, а не только меньший по размеру), однако он начинает возвращать результаты, как только будет найдена первая пара подходящих входных кортежей.

11.3.7. Многопотоковое соединение

Все приведенные выше алгоритмы реализуют операцию соединения двух отношений. Однако во многих запросах необходимо соединять большее количество отношений, поэтому на первый взгляд кажется полезным ввести также алгоритмы, выполняющие соединение более чем двух входных отношений.

Действительно, расширить алгоритмы (например, вложенных циклов и на основе сортировки и слияния) таким образом, чтобы они могли обрабатывать

большее количество входных отношений, совсем несложно. Однако в промышленных системах управления базами данных такие модификации применяются редко.

Причина состоит в том, что количество операций сравнения, которые необходимо выполнить для вычисления результата, растет экспоненциально с ростом числа входных потоков. Поскольку время работы любых алгоритмов соединения обычно определяется временем процессора, необходимого для их выполнения (а не временем обмена с дисками), применение многопоточных операций оказывается менее эффективным.

Однако в случае алгоритма слияния можно поддерживать упорядочение входных отношений по значениям ключа последнего прочитанного объекта из каждого отношения. В этом случае экспоненциальный рост сложности не происходит. Поэтому многопоточное соединение для алгоритма слияния может иметь смысл, и оно действительно применяется в системе PostgreSQL при выполнении внешней сортировки.

11.4. Итоги главы и библиографические комментарии

В главе представлены основные структуры хранения, применяемые в системах управления базами данных, структуры индексов и алгоритмы поиска, а также базовые варианты алгоритмов выполнения основных алгебраических операций (соединения, произведения, удаления дубликатов, группировки и теоретико-множественного пересечения, объединения и разности). Структуры хранения, используемые в настоящее время в системе PostgreSQL, во многом основаны на идеях, реализованных в раннем прототипе [57].

Структура B-дерева, впервые описанная в [8], является первой динамической индексной структурой, наиболее известной и широко применяемой в самых различных СУБД. Публикации, описывающие улучшения для некоторых специальных случаев, продолжают появляться до настоящего времени.

Индексные структуры на основе методов хеширования применялись начиная с ранних СУБД, однако динамические варианты индексов на основе хеширования появились только в конце 70-х гг.: расширяемое хеширование [24] и линейное хеширование [43]. Заметим, что, несмотря на существенные преимущества этих структур, динамическое хеширование применяется на практике достаточно редко.

Среди пространственных индексных структур наиболее значительное распространение получили R-деревья, введенные в [33]. Улучшенный вариант этой структуры, R*-деревья, предложен и исследован в [60]. Исследования различных алгоритмов расщепления переполненных вершин для R-деревьев продолжаются до настоящего времени. Несбалансированные структуры для многомерного индексирования (k-d-B-деревья) предложены в [52].

Анализ различных алгоритмов выполнения алгебраических операций можно найти в [35] и в [46], а также в ранней работе [32].

11.5. Упражнения

Упражнение 11.1. Создайте таблицу, содержащую географические координаты и момент времени (такие атрибуты могут быть, например, у фотографии). Заполните таблицу искусственно сгенерированными значениями.

Упражнение 11.2. Для созданной таблицы постройте двумерный индекс по координатам. (Воспользуйтесь типом данных point и индексом GiST.)

Упражнение 11.3. Постройте еще один индекс по всем колонкам созданной таблицы. (Используйте расширение btree_gist.)

Упражнение 11.4. Напишите запрос, находящий фотографии, сделанные на расстоянии не более двух километров от заданной точки в течение заданного дня. Определите, как зависит скорость выполнения запроса от наличия одного или другого индекса из двух предыдущих упражнений.

Упражнение 11.5. Постройте индекс триграмм, встречающихся в индексируемом тексте. (Воспользуйтесь индексом GIN и расширением pg_trgm.)

Упражнение 11.6. С помощью команды EXPLAIN изучите планы выполнения запросов из упражнений к главе 4. Определите узлы плана, выполняющие выборку данных, соединение, сортировку. Укажите способы выполнения этих операций, которые выбрал планировщик запросов.