5 ФИЗИЧЕСКИЕ МОДЕЛИ ДАННЫХ

Существуют два принципиальных подхода к физическому хранению отношений. Наиболее распространенным является покортежное хранение отношений. Естественно, это обеспечивает быстрый доступ к целому кортежу, но при этом во внешней памяти дублируются общие значения разных кортежей одного отношения и, вообще говоря, могут потребоваться лишние обмены с внешней памятью, если нужна часть кортежа. Физической единицей хранения данных является *блок (страница)*. Размер страницы определяется дисковым контроллером и операционной системой. Одна страница может содержать несколько кортежей.

Альтернативным (менее распространенным) подходом является хранение отношения по столбцам, т.е. единицей хранения является столбец отношения с исключенными дубликатами. Естественно, что при такой организации суммарно в среднем тратится меньше внешней памяти, поскольку дубликаты значений не хранятся; за один обмен с внешней памятью в общем случае считывается больше полезной информации. Дополнительным преимуществом является возможность использования значений столбца отношения для оптимизации выполнения операций соединения. Но при этом требуются существенные дополнительные действия для сборки целого кортежа (или его части).

5.1 Индексы

Индексом называют служебную запись, предназначенную для организации доступа к основным записям файла БД. Основное назначение индекса состоит в обеспечении эффективного *прямого доступа* к кортежу отношения по ключу. Индексная запись имеет вид:

Значение ключа	Номер записи

Ключом индекса является значение атрибута (*простой ключ*) или совокупность значений нескольких атрибутов (*составной ключ*) определенного кортежа. Номер записи указывает на физическую запись, содержащую данный кортеж.

Логически файл БД можно представить состоящим из двух частей:

- индексной области, состоящей из индексных записей;
- основной области, содержащей записи БД.

Индексные записи упорядочены по значению ключа. Взаимодействие этих двух областей составляет сущность механизма индексации для ускорения доступа к данным. Физическое совмещение двух частей в одном файле не обязательно, чаще всего индексная область хранится в отдельном файле.

Если ключом индекса является возможный ключ отношения, то индекс должен обладать свойством уникальности, т.е. не содержать дубликатов ключа. На практике ситуация выглядит обычно противоположно: при объявлении первичного ключа отношения автоматически заводится

уникальный индекс, а единственным способом объявления возможного ключа, отличного от первичного, является явное создание уникального индекса. Это связано с тем, что для проверки сохранения свойства уникальности возможного ключа так или иначе требуется индексная поддержка.

Поскольку при выполнении многих операций языкового уровня требуется сортировка отношений в соответствии со значениями некоторых атрибутов, полезным свойством индекса является обеспечение последовательного просмотра кортежей отношения в диапазоне значений ключа в порядке возрастания или убывания значений ключа.

Общей идеей любой организации индекса, поддерживающего прямой доступ по ключу и последовательный просмотр в порядке возрастания или убывания значений ключа, является хранение упорядоченного списка значений ключа с привязкой к каждому значению ключа списка идентификаторов кортежей. Одна организация индекса отличается от другой главным образом в способе поиска ключа с заданным значением.

Время доступа к записи оценивается в количествах обращений к устройству внешней памяти (диску), т.к. работа с диском является наиболее длительной операцией по сравнению с манипуляциями в оперативной памяти. При таком подходе время доступа определяется целиком только способом доступа и не зависит от быстродействия конкретного накопителя.

5.1.1 Плотные индексы

В файлах с *плотным индексом* (*индексно-прямые файлы*) основная область содержит последовательность записей одинаковой длины, расположенных в произвольном порядке. Значением ключа индексной записи является значение первичного ключа отношения, поэтому ключ однозначно определяет основную запись. Таким образом, каждой записи в основной области соответствует одна запись из индексной области. Все записи индексной области упорядочены по значению ключа, что позволяет применять эффективные способы поиска записи.

Наиболее эффективным алгоритмом поиска на упорядоченном массиве является *бинарный поиск*, основанный на методе деления пополам:

1) все пространство поиска делится пополам;
2) выбирается элемент, расположенный посредине;
3) если данный элемент является искомым, поиск завершается;
4) если нет, определяется, в какой из половин он должен находится;
5) новым пространством поиска выбирается соответствующая половина и переходят к шагу 1.

Если пространство поиска состоит из N элементов, то максимальное количество шагов поиска будет равно:

$$T_{\max} = \log_2 N .$$

Оценим максимальное количество обращений к дисковой памяти при поиске записи. На диске записи хранятся в физических блоках, размер которых определяется контроллером диска. Один блок может содержать несколько индексных записей. За одно обращение к диску считывается один блок. Для поиска записи необходимо найти требуемый индексный блок, прочесть требуемую индексную запись, прочесть основной блок с требуемой записью. Если файл содержит N_i индексных блоков, то *максимальное время доступа* составит:

$$T_{\max} = \log_2 N_i + 1.$$

При последовательном доступе (без использования индекса) максимальное время доступа $T_{\max}=N_i$. Например, если N_i=1024, то при последовательном доступе нужно 1024 обращений к диску, а с использованием индекса – 10+1=11!

При добавлении новой записи она присоединяется в конец основной области. Соответствующая ей индексная запись должна быть помещена в строго определенной место, т.к. индексные записи упорядочены. Для этого индексные блоки содержат свободные области, первоначальный размер которых (*процент расширения*) определяется СУБД. Для добавления индексной записи необходимо:

- найти соответствующий индексный блок;
- прочесть его в оперативную память и модифицировать – вставить новую запись;
- записать блок обратно на диск.

Таким образом, время добавления новой записи составит:

$$T_{\max} = \log_2 N_i + 1 + 1 + 1.$$

При добавлении записей процент расширения индексных блоков уменьшается. При отсутствии в блоке свободной области (*переполнение*) возможны два решения:

1) перестройка индексной области;
2) организация дополнительной области переполнения для не помещающихся индексных записей.

Первый способ более трудоемкий, второй приводит к увеличению времени доступа к записям.

При удалении записи необходимо:

- найти соответствующий индексный блок;
- прочесть его в оперативную память и модифицировать – удалить запись с переписыванием всех последующих на ее место;
- записать блок обратно на диск;
- пометить соответствующую основную запись как удаленную.

Процент расширения индексного блока при удалении записи увеличивается, для удаления требуется такое же количество обращений к диску, как и при записи.

5.1.2 Неплотные индексы

В файлах с *неплотным индексом* (*индексно-последовательные файлы*) основная область содержит упорядоченную последовательность записей одинаковой длины. Значением ключа индексной записи является значение первичного ключа первой записи в основном блоке:

Ключ первой записи блока	**Номер блока**

Структура индексно-последовательного файла представлена на рисунке 6.1.

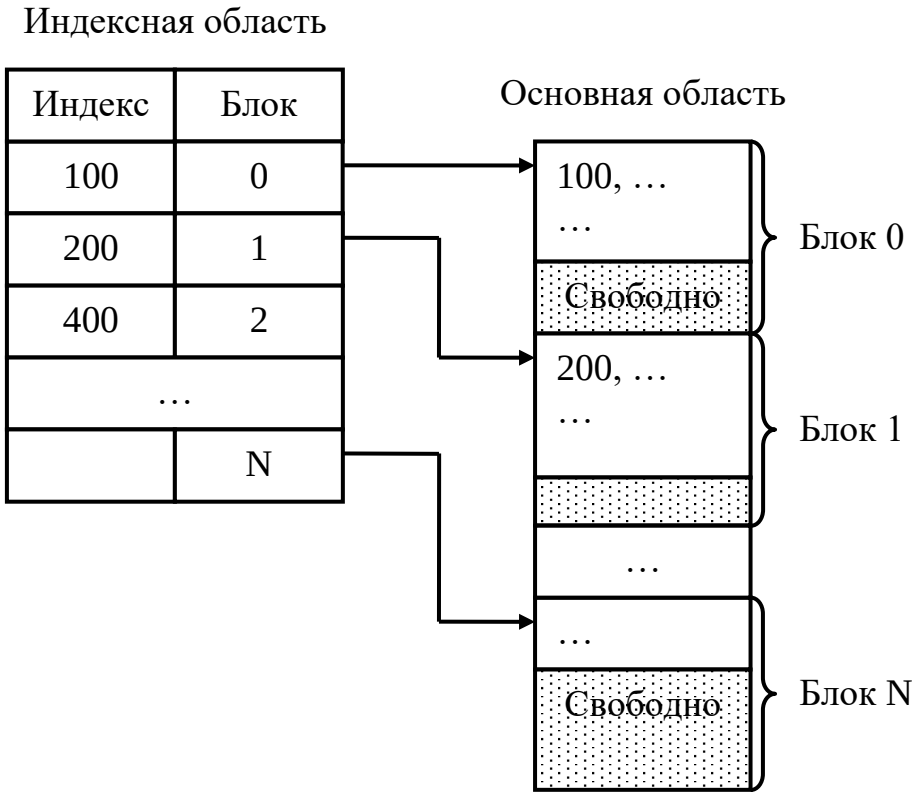

Рис. 6.1. Структура индексно-последовательного файла

Все записи индексной области по-прежнему упорядочены по значению ключа. Каждая запись из индексной области соответствует блоку в основной области. Таким образом, упорядоченное хранение основных записей позволяет уменьшить количество хранимых индексных записей. Это приводит к уменьшению количества индексных блоков N_i и, соответственно, времени доступа по сравнению с плотным индексом.

Новая запись должна заноситься в строго определенный основной блок на соответствующее место. Поэтому алгоритм добавления записи в индексно-последовательный файл состоит из следующих шагов:

- найти соответствующий основной блок;
- прочесть его в оперативную память и модифицировать – вставить новую запись на ее место;
- записать блок обратно на диск.

Индексная область при добавлении новой записи не корректируется. Процент расширения для блока основной области (Full-factor) определяется СУБД, при переполнении блока происходит перестройка всего индексно-последовательного файла.

При удалении записи она физически уничтожается в основной области, при этом индексная область обычно не корректируется, даже если удаляется первая запись блока. Поэтому количество обращений к диску такое же, как при добавлении новой записи.

Пример. Предположим, что БД содержит 2^{18}=262144 записей длиной 2^7=128 байт; размер блока – 2^{10}=1024 байт (8 записей в блоке); размер ключа – 5 байт. Тогда для хранения записей БД потребуется 262144/8 = =32768 основных блоков.

При использовании плотного индекса для ссылки на 262144 записей необходимо поле номера записи длиной 3 байта. Тогда длина индексной записи равна 5+3 = 8 байт и индексный блок будет содержать 1024/8 = 128 записей. Всего потребуется 262144/128 = 2048 индексных блоков. Максимальное время доступа составит:

$$\log_2 2048 + 1 = 11 + 1 = 12.$$

При неплотном индексе для ссылки на 32768=2^{15} основных блока необходимо поле номера блока длиной 2 байта (2^{16}). Тогда длина индексной записи составит 5+2 = 7 байт и индексный блок будет содержать 1024/7 = =147 записей. Всего потребуется 32768/147 = 223 индексных блока. Максимальное время доступа:

$$\log_2 223 + 1 = 8 + 1 = 9.$$

Таким образом, использование неплотного индекса сокращает время доступа на 25 %.

5.2 B-деревья

Наиболее популярным подходом к организации индексов в базах данных является использование техники B-деревьев. Она основана на мультииндексировании – применении индексирования для ускорения доступа к индексным блокам.

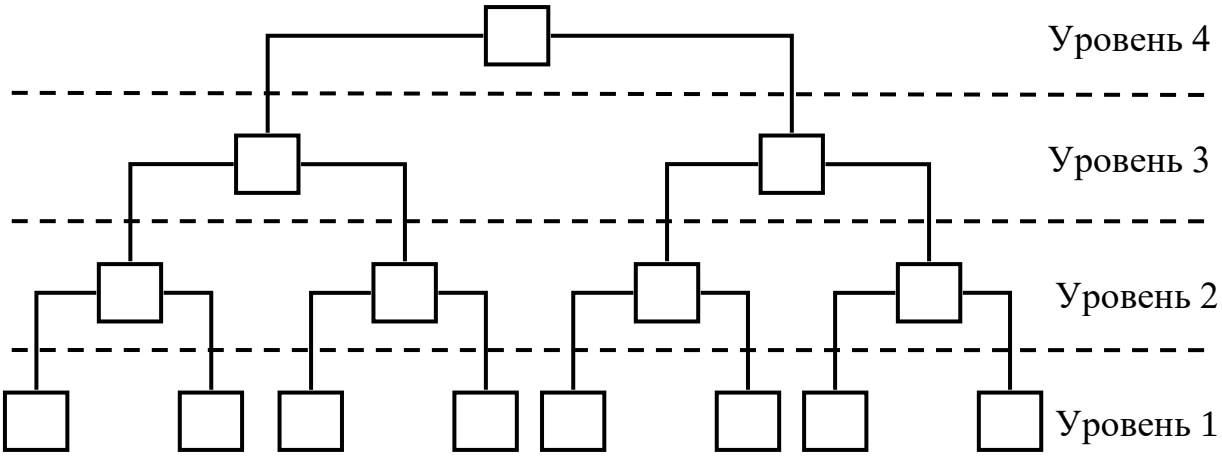

Рис. 6.2. B-дерево

Действительно, сама индексная область может рассматриваться как основной файл, для которого можно построить неплотный индекс. Над новым индексом строится следующий, и так до тех пор, пока мы не получим индексную область, состоящую из одного блока. В итоге мы будем иметь дерево, представленное на рисунке 6.2, у которого:

- внутренние узлы соответствуют блокам, содержащим индексные записи для индексов;
- листовые узлы соответствуют блокам, содержащим индексные записи для основных записей;
- узел-родитель содержит индексы для узлов-потомков.

Поиск в B-дереве – это прохождение от корня к листу в соответствии с заданным значением ключа. При этом *длина пути* (количество проходимых узлов) от корня дерева к любому его листу одна и та же. Такое дерево относится к категории *сбалансированных (balanced) деревьев* или *B-де-ревьев*.

Заметим, что поскольку деревья сильно ветвистые и сбалансированные, то для выполнения поиска по любому значению ключа потребуется одно и то же (и обычно небольшое) число обменов с внешней памятью. Более точно, в сбалансированном дереве, где длины всех путей от корня к листу одни и те же, если во внутреннем блоке помещается n ключей, то при хранении m записей требуется дерево глубиной $\log_n(m)$, где $\log_n$ вычисляет логарифм по основанию n. Если n достаточно велико (обычный случай), то глубина дерева невелика, и производится быстрый поиск.

Основной «изюминкой» B-деревьев является автоматическое поддержание свойства сбалансированности. Рассмотрим в качестве примера операции занесения и удаления записей.

При занесении новой записи выполняется:

- Поиск листового блока. Фактически производится обычный поиск по ключу. Если в B-дереве не содержится ключ с заданным значением, то будет получен номер блока, в которой ему надлежит содержаться, и соответствующие координаты внутри блока.

– Помещение записи на место. Естественно, что вся работа производится в буферах оперативной памяти. Листовой блок, в который требуется занести запись, считывается в буфер, и в нем выполняется операция вставки. Размер буфера должен превышать размер блока внешней памяти.

Если после выполнения вставки новой записи размер используемой части буфера не превосходит размера блока, то на этом выполнение операции занесения записи заканчивается. Буфер может быть немедленно вытолкнут во внешнюю память, или временно сохранен в оперативной памяти в зависимости от политики управления буферами.

Если же возникло переполнение блока (т.е. размер используемой части буфера превосходит размер блока), то выполняется расщепление блока. Для этого запрашивается новый блок внешней памяти, используемая часть буфера разбивается грубо говоря пополам (так, чтобы вторая половина также начиналась с ключа), и вторая половина записывается во вновь выделенный блок, а в старом блоке модифицируется значение размера свободной памяти. Естественно, модифицируются ссылки по списку листовых блоков.

Чтобы обеспечить доступ от корня дерева к заново заведенному блоку, необходимо соответствующим образом модифицировать внутренний блок, являющийся предком ранее существовавшего листового блока, т.е. вставить в него соответствующее значение ключа и ссылку на новый блок. При выполнении этого действия может снова произойти переполнение теперь уже внутреннего блока, и он будет расщеплен на два. В результате потребуется вставить значение ключа и ссылку на новый блок во внутренний блок-предка выше по иерархии и т.д.

Предельным случаем является переполнение корневого блока B-дерева. В этом случае он тоже расщепляется на два, и заводится новый корневой блок дерева, т.е. его глубина увеличивается на единицу.

При удалении записи выполняются следующие действия:

– Поиск записи по ключу. Если запись не найдена, то значит удалять ничего не нужно.
– Реальное удаление записи в буфере, в который прочитан соответствующий листовой блок.

Если после выполнения этой подоперации размер занятой в буфере области оказывается таковым, что его сумма с размером занятой области в листовых блоках, являющихся левым или правым братом данного блока, больше, чем размер блока, операция завершается.

Иначе производится слияние с правым или левым братом, т.е. в буфере производится новый образ блока, содержащего общую информацию из данного блока и его левого или правого брата. Ставший ненужным листовой блок заносится в список свободных блоков. Соответствующим образом корректируется список листовых блоков.

Чтобы устранить возможность доступа от корня к освобожденному блоку, нужно удалить соответствующее значение ключа и ссылку на освобожденный блок из внутреннего блока – его предка. При этом может

возникнуть потребность в слиянии этого блока с его левым или правыми братьями и т.д.

Предельным случаем является полное опустошение корневого блока дерева, которое возможно после слияния последних двух потомков корня. В этом случае корневой блок освобождается, а глубина дерева уменьшается на единицу.

Как видно, при выполнении операций вставки и удаления свойство сбалансированности B-дерева сохраняется, а внешняя память расходуется достаточно экономно.

Проблемой является то, что при выполнении операций модификации слишком часто могут возникать расщепления и слияния. Чтобы добиться эффективного использования внешней памяти с минимизацией числа расщеплений и слияний, применяются более сложные приемы, в том числе:

- упреждающие расщепления, т.е. расщепления блока не при его переполнении, а несколько раньше, когда степень заполненности блока достигает некоторого уровня;
- переливания, т.е. поддержание равновесного заполнения соседних блоков;
- слияния 3-в-2, т.е. порождение двух листовых блоков на основе содержимого трех соседних.

Пример. Предположим, что БД содержит 2^{18}=262144 основных блока; индексный блок содержит 2^8=256 записей, индексная область – 262144/256 = 1024 блока.

При использовании неплотного индекса максимальное время доступа составит:

$$\log_2 1024 + 1 = 10 + 1 = 11.$$

При использовании B-дерева количество блоков на уровне 1 будет равно 262144/256 = 1024; на уровне 2 будет 1024/256 = 4 блока; на уровне 3 – 4/256 = 1 блок, содержащий 4 записи – корневой блок. Таким образом, мы будем иметь B-дерево глубиной $\log_{256} 262144 = 3$ (округление в большую сторону). Тогда время доступа всегда будет равно 3 + 1 = 4.

5.3 Хэширование

Альтернативным и все более популярным подходом к организации индексов является использование техники хэширования. Общей идеей методов хэширования является применение к значению ключа некоторой функции свертки (хэш-функции), вырабатывающей значение меньшего размера. Свертка значения ключа затем используется для доступа к записи.

В самом простом, классическом случае свертка ключа используется как адрес в таблице, содержащей ключи и записи. Основным требованием к хэш-функции является равномерное распределение значения свертки. При возникновении коллизий (одна и та же свертка для нескольких значений ключа) образуются цепочки переполнения. Главным ограничением этого метода является фиксированный размер таблицы. Если таблица заполнена

слишком сильно или переполнена, т.е. возникнет слишком много цепочек переполнения, то главное преимущество хэширования – доступ к записи почти всегда за одно обращение к таблице – будет утрачено. Расширение таблицы требует ее полной переделки на основе новой хэш-функции (со значением свертки большего размера).

В случае баз данных такие действия являются абсолютно неприемлемыми. Поэтому обычно вводят промежуточные таблицы-справочники, содержащие значения ключей и адреса записей, а сами записи хранятся отдельно. Тогда при переполнении справочника требуется только его переделка, что вызывает меньше накладных расходов.

Чтобы избежать потребности в полной переделке справочников, при их организации часто используют технику B-деревьев с расщеплениями и слияниями. Хэш-функция при этом меняется динамически, в зависимости от глубины B-дерева. Путем дополнительных технических ухищрений удается добиться сохранения порядка записей в соответствии со значениями ключа. В целом методы B-деревьев и хэширования все более сближаются.