Физические модели баз данных

Физические модели баз данных определяют способы размещения данных в среде хранения и способы доступа к этим данным, которые поддерживаются на физическом уровне.

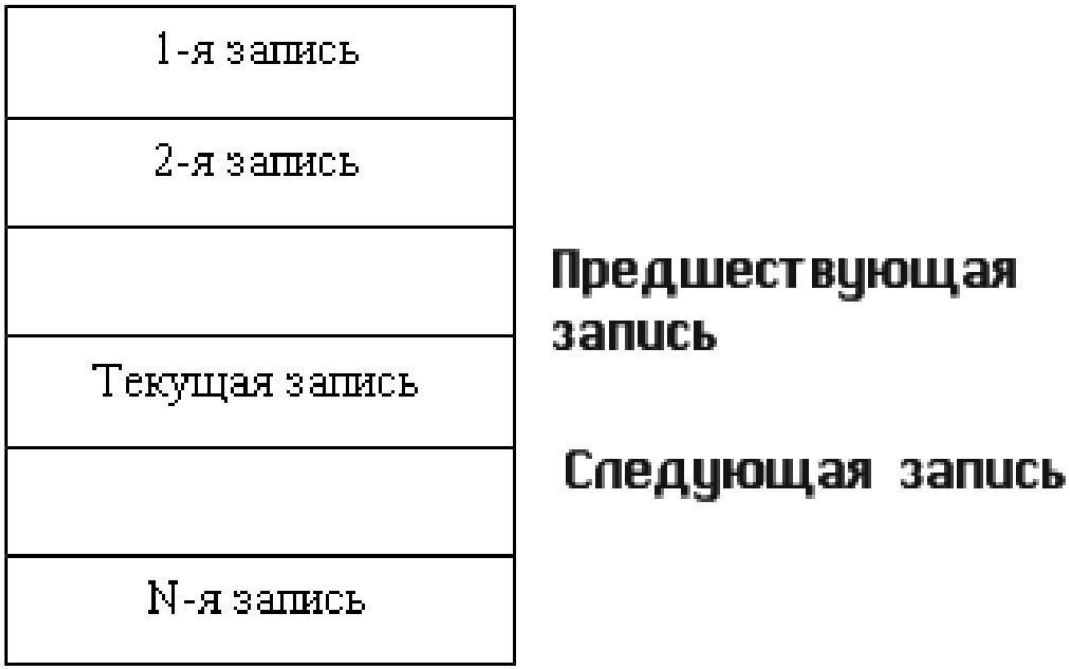
Файловые структуры, используемые для хранения информации в базах данных

В каждой СУБД по-разному организованы хранение и доступ к данным, однако существуют некоторые файловые структуры, которые имеют общепринятые способы организации и широко применяются практически во всех СУБД.

В системах баз данных файлы и файловые структуры, которые используются для хранения информации во внешней памяти, можно классифицировать следующим образом


Файлы
Прямого доступа
Последовательного доступа
Индексные
Инвертированные списки
Взаимосвязанные файлы
Плотный индекс (индексно-прямые)
Неплотный индекс (индексно-последовательный)
В-деревья
С однонаправленными цепочками
С двунаправленными цепочками

С точки зрения пользователя, *файлом* называется поименованная линейная последовательность записей, расположенных на внешних носителях, то есть всегда в файле можно определить текущую запись, предшествующую ей и следующую за ней. Всегда существует понятие первой и последней записи файла. В соответствии с методами управления доступом различают устройства внешней памяти с *произвольной адресацией* (магнитные и оптические диски) и устройства *с последовательной адресацией* (магнитофоны, стримеры).



Основная область

Содержание записей	Ссылка на синонимы
Петров	1
Степанов	2

. . .

Область переполнения

Содержание записей	Ссылка на синонимы
Петров	
Степанов	3
Степанчиков	

Файл как линейная последовательность записей

Элемент 1	Элемент 2	…	…	Элемент N	Элемент N+1

Модель хранения информации на устройстве последовательного доступа

Файлы с постоянной длиной записи, расположенные на устройствах прямого доступа (УПД), являются *файлами прямого доступа.*

В этих файлах физический адрес расположения нужной записи может быть вычислен по номеру записи (NZ). Каждая файловая система СУФ — система управления файлами поддерживает некоторую иерархическую файловую структуру, включающую чаще всего неограниченное количество уровней иерархии в представлении внешней памяти .

Для каждого файла в системе хранится следующая информация:

- имя файла;
- тип файла (например, расширение или другие характеристики);
- размер записи;
- количество занятых физических блоков;
- базовый начальный адрес;
- ссылка на сегмент расширения;
- способ доступа (код защиты).

Иерархическая организация файловой структуры хранения

Для файлов с постоянной длиной записи адрес размещения записи с номером К может быть вычислен по формуле:

BA + (K - 1) * LZ + 1, где BA — базовый адрес, LZ — длина записи.

- Файлы прямого доступа обеспечивают наиболее быстрый доступ к произвольным записям, и их использование считается наиболее перспективным в системах баз данных.

Файлы с переменной длиной записи всегда являются файлами последовательного доступа. Они могут быть организованы двумя способами:

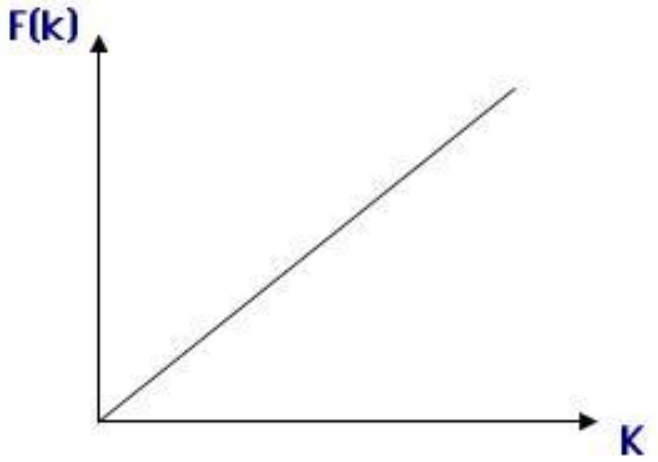
- Конец записи отличается специальным маркером.

Запись 1 X Запись 2 X Запись 3 X

- В начале каждой записи записывается ее длина (Здесь LZN — длина N-й записи).

LZ1 Запись 1 LZ2 Запись 2 LZ3 Запись 3

В некоторых очень редких случаях возможно построение функции, которая по значению ключа однозначно вычисляет адрес (номер записи файла). NZ = F(K). где NZ — номер записи, К — значение ключа, F( ) — функция.



Однако далеко не всегда удастся построить взаимно-однозначное соответствие между значениями ключа и номерами записей. Часто бывает, что значения ключей разбросаны по нескольким диапазонам В этом случае не удается построить взаимнооднознач-ную функцию, либо эта функция будет иметь множество незадействованных значений, которые соответствуют недопустимым значениям ключа. В подобных случаях применяют различные методы хэширования (рандомизации) и создают специальные хэш- функции.

Суть методов хэширования состоит в том, что мы берем значения ключа (или некоторые его характеристики) и используем его для начала поиска, то есть мы вычисляем некоторую хэш-функцию h(k) и полученное значение берем в качестве адреса начала поиска. Таким образом, мы допускаем, что нескольким разным ключам может соответствовать одно значение хэш-функции (то есть один адрес). Подобные ситуации называются *коллизиями.* Значения ключей, которые имеют одно и то же значение хэш-функции, называются *синонимами.*

Поэтому при использовании хэширования как метода доступа необходимо принять два независимых решения:

- выбрать хэш-функцию;
- выбрать метод разрешения коллизий.

Стратегия разрешения коллизий с областью переполнения

Первая стратегия условно может быть названа стратегией с областью переполнения. При выборе этой стратегии область хранения разбивается на 2 части:

- основную область;
- область переполнения.

Если вновь заносимая запись имеет значение функции хэширования такое же, которое использовала другая запись, уже имеющаяся в БД, то новая запись заносится в область переполнения на первое свободное место, а в записи-синониме, которая находится в основной области, делается ссылка на адрес вновь размещенной записи в области переполнения. Если же уже существует ссылка в записи-синониме, которая расположена в основной области, то тогда новая запись получает дополнительную информацию в виде ссылки и уже в таком виде заносится в область переполнения.

При этом цепочка синонимов не разрывается, но мы не просматриваем ее до конца, чтобы расположить новую запись в конце цепочки синонимов, а располагаем всегда новую запись на второе место в цепочке синонимов, что существенно сокращает время размещения новой записи. При таком алгоритме время размещения любой новой записи составляет не более двух обращений к диску, с учетом того, что номер первой свободной записи в области переполнения хранится в виде системной переменной.

При поиске записи также сначала вычисляется значение ее хэш-функции и считывается первая запись в цепочке синонимов, которая расположена в основной области. Если искомая запись не соответствует первой в цепочке синонимов, то далее поиск происходит перемещением по цепочке синонимов, пока не будет обнаружена требуемая запись. Скорость поиска зависит от длины цепочки синонимов, поэтому качество хэш-функции определяется максимальной длиной цепочки синонимов.

При удалении произвольной записи сначала определяется ее место расположения. Если удаляемой является первая запись в цепочке синонимов, то после удаления на ее место в основной области заносится вторая (следующая) запись в цепочке синонимов, при этом все указатели (ссылки на синонимы) сохраняются.

Если же удаляемая запись находится в середине цепочки синонимов, то необходимо провести корректировку указателей: в записи, предшествующей удаляемой, в цепочке ставится указатель из удаляемой записи.

Если это последняя запись в цепочке, то все равно механизм изменения указателей такой же, то есть в предшествующую запись заносится признак отсутствия следующей записи в цепочке, который ранее хранился в последней записи.

Организация стратегии свободного замещения

При этой стратегии файловое пространство не разделяется на области, но для каждой записи добавляется 2 указателя: указатель на предыдущую запись в цепочке синонимов и указатель на следующую запись в цепочке синонимов. Отсутствие соответствующей ссылки обозначается специальным символом, например нулем. Для каждой новой записи вычисляется значение хэш-функции, и если данный адрес свободен, то запись попадает на заданное место и становится первой в цепочке синонимов. Если адрес, соответствующий полученному значению хэш-функции, занят, то по наличию ссылок определяется, является ли запись, расположенная по указанному адресу, первой в цепочке синонимов. Если да, то новая запись располагается на первом свободном месте и для нее устанавливаются соответствующие ссылки: она становится второй в цепочке синонимов, на нее ссылается первая запись, а она ссылается на следующую, если таковая есть.

Индексные файлы

Несмотря на высокую эффективность хэш-адресации, в файловых структурах далеко не всегда удается найти соответствующую функцию, поэтому при организации доступа по первичному ключу широко используются **индексные** файлы. В некоторых коммерческих системах индексными файлами называются также и файлы, организованные в виде инвертированных списков, которые используются для доступа по вторичному ключу.

Индексные файлы можно представить как файлы, состоящие из двух частей. В большинстве случаев индексная область образует отдельный индексный файл, а основная область образует файл, для которого создается индекс.

Мы предполагаем, что сначала идет индексная область, которая занимает некоторое целое число блоков, а затем идет основная область, в которой последовательно расположены все записи файла.

В зависимости от организации индексной и основной областей различают 2 типа файлов: с *плотным индексом* и с *неплотным индексом*.

Файлы с плотным индексом называются также индексно-прямыми файлами, а файлы с неплотным индексом называются также индексно-последовательными файлами.

Файлы с плотным индексом, или индексно-прямые файлы

В файлах *с плотным индексом* основная область содержит последовательность записей одинаковой длины, расположенных в произвольном порядке, а структура индексной записи в них имеет следующий вид:

Здесь *значение ключа* — это значение первичного ключа, а *номер записи* — это порядковый номер записи в основной области, которая имеет данное значение первичного ключа.

Значение ключа	**Номер записи**

Так как индексные файлы строятся для первичных ключей, однозначно определяющих запись, то в них не может быть двух записей, имеющих одинаковые значения первичного ключа. В индексных файлах с плотным индексом для каждой записи и основной области существует одна запись из индексной области. Все записи в индексной области упорядочены по значению ключа.

Длина доступа к произвольной записи оценивается не в абсолютных значениях, а в количестве обращений к устройству внешней памяти, которым обычно является диск. Именно обращение к диску является наиболее длительной операцией по сравнению со всеми обработками в оперативной памяти.

Наиболее эффективным алгоритмом поиска на упорядоченном массиве является **логарифмический,** или **бинарный**, поиск. При этом все пространство поиска разбивается пополам, и так как оно строго упорядочено, то определяется сначала, не является ли элемент искомым, а если пет, то в какой половине его надо искать. Следующим шагом мы определенную половину также делим пополам и производим аналогичные сравнения, и т. д., пока не обнаружим искомый элемент. Максимальное количество шагов поиска определяется двоичным логарифмом от общего числа элементов в искомом пространстве поиска:

$T_n = \log_2 N$, где N — число элементов.

Для того чтобы оценить максимальное время доступа, нам надо определить количество обращений к диску для поиска произвольной записи.

На диске записи файлов хранятся в блоках. Размер блока определяется физическими особенностями дискового контроллера и операционной системой. В одном блоке могут размещаться несколько записей. Поэтому надо определить количество индексных блоков, которое потребуется для размещения всех требуемых индексных записей, а потому максимальное число обращений к диску будет равно двоичному логарифму от заданного числа блоков плюс единица. Зачем нужна единица? После поиска номера записи в индексной области мы должны еще обратиться к основной области файла. Поэтому формула для вычисления максимального времени доступа в количестве обращений к диску выглядит следующим образом:

$$T_n = \log_2 N_{бл.инд.} + 1.$$

Рассмотрим пример и сравним время доступа при последовательном просмотре и при организации плотного индекса.

Мы имеем следующие исходные данные:

Длина записи файла (LZ) — 128 байт. Длина первичного ключа (LK) — 12 байт. Количество записей в файле (KZ) — 100 000. Размер блока (LB) — 1024 байт.

Рассчитаем размер индексной записи. Для представления целого числа в пределах 100 000 нам потребуется 3 байта, у нас допустима только четная адресация, поэтому нам надо отвести 4 байта для хранения номера записи, тогда длина индексной записи будет равна сумме размера ключа и ссылки на номер записи, то есть:

LI = LK + 4 = 12 + 4 = 16 байт.

Определим количество индексных блоков, которое требуется для обеспечения ссылок на заданное количество записей. Для этого сначала определим, сколько индексных записей может храниться в одном блоке:

KIZB = LB/LI = 1024/16 = 64 индексных записи в одном блоке.

Теперь определим необходимое количество индексных блоков:

KIB = KZ/KZIB = 100 000/64 = 1563 блока.

Мы округлили в большую сторону, потому что пространство выделяется целыми блоками, и последний блок у нас будет заполнен не полностью.

А теперь можем вычислить максимальное количество обращений к диску при поиске произвольной записи:

$T_{\text{поиска}} = \log_2 KIB + 1 = \log_2 1563 + 1 = 11 + 1 = 12$ обращений к диску.

Количество блоков, которое необходимо для хранения всех 100 000 записей, мы определим по следующей формуле:
КВО = KZ/(LB/LZ) – 100 000/(1024/128) - 12500 блоков.
И это означает, что максимальное время доступа равно 12500 обращений к диску. Да, действительно, выигрыш существенный.
При операции добавления осуществляется запись в конец основной области. В индексной области необходимо произвести занесение информации в конкретное место, чтобы не нарушать упорядоченности. Поэтому вся индексная область файла разбивается на блоки и при начальном заполнении в каждом блоке остается свободная область (процент расширения).
После определения блока, в который должен быть занесен индекс, этот блок копируется в оперативную память, там он модифицируется путем вставки в нужное место новой записи
Определим максимальное количество обращений к диску, которое требуется при добавлении записи, — это количество обращений, необходимое для поиска записи плюс одно обращение для занесения измененного индексного блока и плюс одно обращение для занесения записи в основную область.

$$T_{\text{добавления}} = \log_2 N + 1 + 1 + 1.$$

Когда исчезает свободная область, возникает переполнение индексной области. В этом случае возможны два решения: либо перестроить заново индексную область, либо организовать область переполнения для индексной области, в которой будут храниться не поместившиеся в основную область записи.

Однако первый способ потребует дополнительного времени на перестройку индексной области, а второй увеличит время на доступ к произвольной записи и потребует организации дополнительных ссылок в блоках па область переполнения.

При удалении записи возникает следующая последовательность действий: запись в основной области помечается как удаленная (отсутствующая), в индексной области соответствующий индекс уничтожается физически, то есть записи, следующие за удаленной записью, перемещаются на ее место и блок, в котором хранился данный индекс, заново записывается па диск. При этом количество обращений к диску для этой операции такое же, как и при добавлении новой записи

Файлы с неплотным индексом, или индексно-последовательные файлы

Попробуем усовершенствовать способ хранения файла: будем хранить его в упорядоченном виде и применим алгоритм двоичного поиска для доступа к произвольной записи. Тогда время доступа к произвольной записи будет существенно меньше. Для нашего примера это будет:

$T = \log_2 КВО = \log_2 12500 = 14$ обращений к диску.

И это существенно меньше, чем 12 500 обращений при произвольном хранении записей файла. Однако и поддержание основного файла в упорядоченном виде также операция сложная.

Неплотный индекс строится именно для упорядоченных файлов. Для этих файлов используется принцип внутреннего упорядочения для уменьшения количества хранимых индексов. Структура записи индекса для таких файлов имеет следующий вид:

Значение ключа первом записи блока Номер блока с этой записью

В индексной области мы теперь ищем нужный блок по заданному значению первичного ключа. Так как все записи упорядочены, то значение первой записи блока позволяет нам быстро определить, в каком блоке находится искомая запись. Все остальные действия происходят в основной области.

Время сортировки больших файлов весьма значительно, но поскольку файлы поддерживаются сортированными с момента их создания, накладные расходы в процессе добавления новой информации будут гораздо меньше. Оценим время доступа к произвольной записи для файлов с неплотным индексом.

Тогда длина индексной записи будет равна:

LI = LK + 2 = 14 + 2 - 14 байт.

Тогда количество индексных записей в одном блоке будет равно:

KIZB = LB/LI = 1024/14 = 73 индексные записи в одном блоке.

Определим количество индексных блоков, необходимое для хранения требуемых индексных записей:

KIB = KBO/KZIB = 12500/73 = 172 блока.

Тогда время доступа по прежней формуле будет определяться:

$T_{поиска} = \log_2 KIB + 1 = \log_2 172 + 1 = 8+1 = 9$ обращений к диску.

Т.е. при переходе к неплотному индексу время доступа уменьшилось практически в полтора раза. Поэтому можно признать, что организация неплотного индекса дает выигрыш в скорости доступа.

Рассмотрим процедуры добавления и удаления новой записи при подобном индексе.

Здесь новая запись должна заноситься сразу в требуемый блок на требуемое место, которое определяется заданным принципом упорядоченности на множестве значений первичного ключа. Поэтому сначала ищется требуемый блок основной памяти, в который надо поместить новую запись, а потом этот блок считывается, затем в оперативной памяти корректируется содержимое блока и он снова записывается на диск на старое место. Должен быть задан процент первоначального заполнения блоков, применительно к основной области. В MS SQL server этот процент называется Full-factor и используется при формировании кластеризованных индексов. Кластеризованными называются как раз индексы, в которых исходные записи физически упорядочены по значениям первичного ключа. При внесении новой записи индексная область не корректируется.

Количество обращений к диску при добавлении новой записи равно количеству обращений, необходимых для поиска соответствующего блока плюс одно обращение, которое требуется для занесения измененного блока на старое место.

$T_{\text{добавлений}} = \log_2 N + 1 + 1$ обращений.

Организация индексов в виде B-tree (B-деревьев) (balanced)

Развитие предыдущего метода: если мы построим неплотный индекс, то сама индексная область может быть рассмотрена нами как основной файл, над которым надо снова построить неплотный индекс, а потом снова над новым индексом строим следующий и так до того момента, пока не останется всего один индексный блок.

В общем случае получим некоторое дерево, каждый родительский блок которого связан с одинаковым количеством подчиненных блоков, число которых равно числу индексных записей, размещаемых в одном блоке. Количество обращений к диску при этом для поиска любой записи одинаково и равно количеству уровней в построенном дереве.

На первом уровне число блоков основной области, 12 500 блоков.

Второй уровень образуется из неплотного индекса, вычислили, что это 172 блока. А теперь над этим вторым уровнем снова построим неплотный индекс.

Длина индексной записи равна 14 байтам. Количество индексных записей в одном блоке нам тоже известно, и оно равно 73. Поэтому сразу определим, сколько блоков нам необходимо для хранения ссылок на 172 блока. $KIB_3 = KIB_2/KZIB = 172/73 = 3$ блока

И над третьим уровнем строим новый, и на нем будет всего один блок, в котором будет всего три записи. Поэтому число уровней в построенном дереве равно четырем, и соответственно количество обращений к диску для доступа к произвольной записи равно четырем - всегда одно и то же, одинаковое для доступа к любой записи.

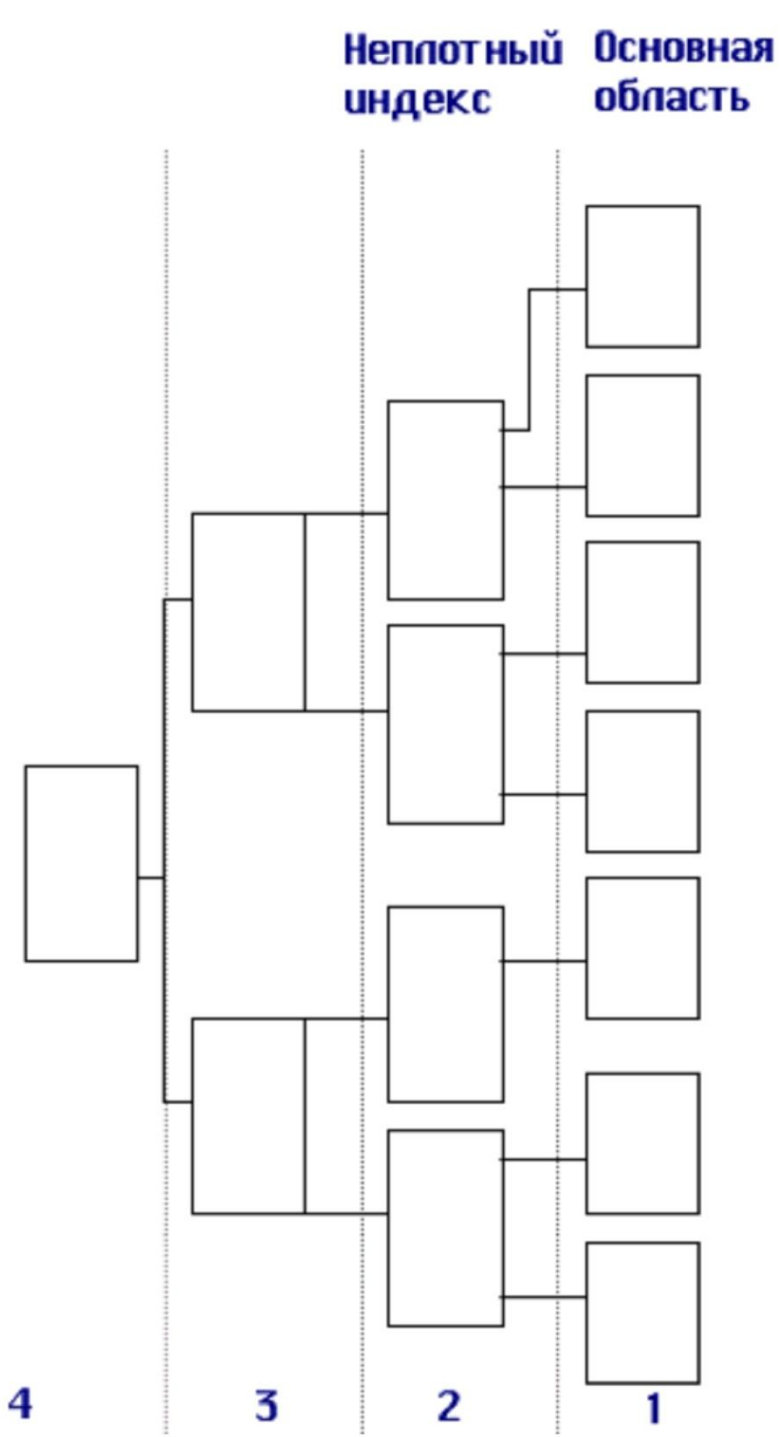
$T_д = R_{уравн.} = 4$

1 уровень 12500 блоков

2 уровень 172 блоков

3 уровень 3 блоков

4 уровень 1 блоков

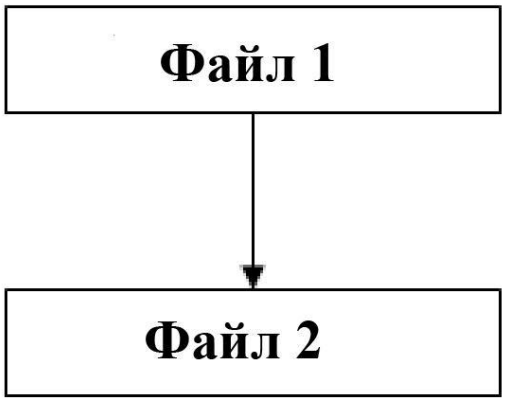


Построенное B-дерево

Для моделирования отношений 1:M (один-ко-многим) и M:M (многие-ко-мно-гим) на файловых структурах используется принцип организации цепочек записей внутри файла и ссылки на номера записей для нескольких взаимосвязанных файлов.

Моделирование отношения 1:M с использованием однонаправленных указателей

В этом случае связываются два файла, например F1 и F2, причем предполагается, что одна запись в файле F1 может быть связана с несколькими записями в файле F2. Условно это можно представить в виде, изображенном на рис.



Ключ	Запись	Ссылка-указатель на первую запись в «Подчиненном» файле, с которой начинается цепочка записей файла F2, связанных с данной записью файла F1

Алгоритм нахождения нужных записей «подчиненного» файла

Шаг 1. Ищется запись в «основном» файле в соответствии с его организацией Если требуемая запись найдена, то переходим к шагу 2, в противном случае выводим сообщение об отсутствии записи основного файла.

Шаг 2. Анализируем указатель в основном файле если он пустой, для этой записи нет ни одной связанной с ней записи в «подчиненном файле», и выводим соответствующее сообщение, в противном случае переходим к шагу 3.

Шаг 3. По ссылке-указателю в найденной записи основного файла переходим прямым методом доступа по номеру записи на первую запись в цепочке «Подчиненного» файла. Переходим к шагу 4.

Шаг 4. Анализируем текущую запись на содержание если это искомая запись, то мы заканчиваем поиск, в противном случае переходим к шагу 5.

Шаг 5. Анализируем указатель на следующую запись в цепочке если он пуст, то выводим сообщение, что искомая запись отсутствует, и прекращаем поиск, в противном случае по ссылке-указателю переходим на следующую запись в «подчиненном файле» и снова переходим к шагу 4.

Структура записи «подчиненного» файла.

Номер записи	**Указатель на следующую запись в цепочке**	**Содержимое записи**

Алгоритм удаления записи из цепочки «подчиненного» файла

Шаг 1. Ищется удаляемая запись в соответствии с ранее рассмотренным алгоритмом. Единственным отличием при этом является обязательное сохранение в специальной переменной номера предыдущей записи в цепочке, допустим, это переменная NP.

Шаг 2. Запоминаем в специальной переменной указатель на следующую запись в найденной записи, например, заносим его в переменную NS. Переходим к шагу 3.

Шаг 3. Помечаем специальным символом, например символом звездочка (*), найденную запись, то есть в позиции указателя на следующую запись в цепочке ставим символ «*» — это означает, что данная запись отсутствует, а место в файле свободно и может быть занято любой другой записью.

Шаг 4. Переходим к записи с номером, который хранится в NP, и заменяем в ней указатель на содержимое переменной NS.

Для того чтобы эффективно использовать дисковое пространство при включении новой записи в «подчиненный файл», ищется первое свободное место, т. е. запись, помеченная символом «*», и на ее место заносится новая запись, после этого производится модификация соответствующих указателей. При этом необходимо различать 3 случая

Добавление записи на первое место в цепочке.

Добавление записи в конец цепочки.

Добавление записи на заданное место в цепочке.

Инвертированные списки

До сих пор мы рассматривали структуры данных, которые использовались для ускорения доступа по первичному ключу. Однако часто в базах данных требуется проводить операции доступа по вторичным ключам. Вторичным ключом является набор атрибутов, которому соответствует набор искомых записей. Это означает, что существует множество записей, имеющих одинаковые значения вторичного ключа.

Для обеспечения ускорения доступа по вторичным ключам используются структуры, называемые инвертированными списками

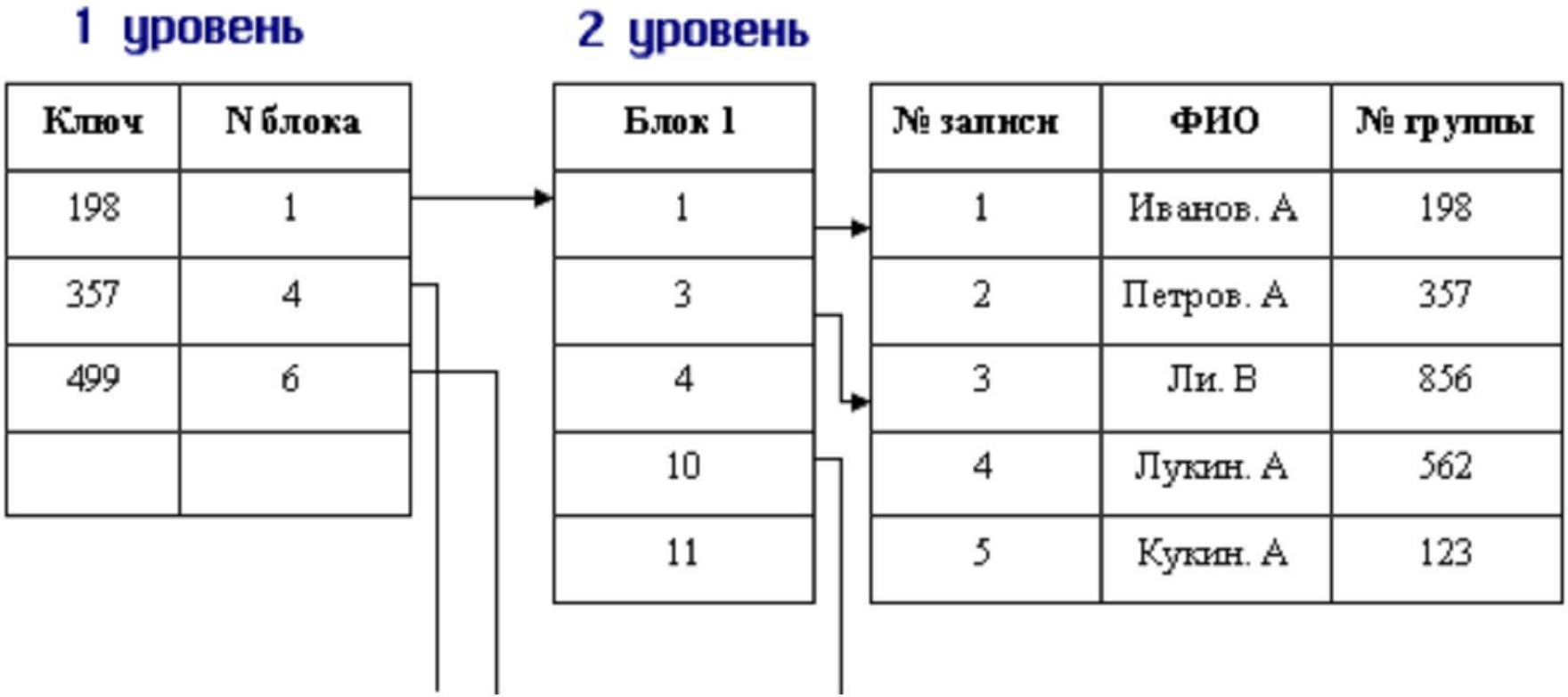
Инвертированный список в общем случае — это двухуровневая индексная структура.

Здесь на первом уровне находится файл или часть файла, в которой упорядочено расположены значения вторичных ключей. Каждая запись с вторичным ключом имеет ссылку на номер первого блока в цепочке блоков, содержащих номера записей с данным значением вторичного ключа.

На втором уровне находится цепочка блоков, содержащих номера записей, содержащих одно и то же значение вторичного ключа. При этом блоки второго уровня упорядочены по значениям вторичного ключа.

И наконец, на третьем уровне находится собственно основной файл. На первом шаге мы ищем в области первого уровня заданное значение вторичного ключа, а затем по ссылке считываем блоки второго уровня, содержащие номера записей с заданным значением вторичного ключа, а далее уже прямым доступом загружаем в рабочую область пользователя содержимое всех записей, содержащих заданное значение вторичного ключа

Построение инвертированного списка по номеру группы для списка студентов



При модификации основного файла происходит следующая последовательность действий:

Изменяется запись основного файла.

Исключается старая ссылка на предыдущее значение вторичного ключа.

Добавляется новая ссылка на новое значение вторичного ключа.

При этом следует отметить, что два последних шага выполняются для всех вторичных ключей, по которым созданы инвертированные списки. И, разумеется, такой процесс требует гораздо больше временных затрат, чем просто изменение содержимого записи основного файла без поддержки всех инвертированных списков.

Поэтому не следует безусловно утверждать, что введение индексных файлов (в том числе и инвертированных списков) всегда ускоряет обработку информации в базе данных.

Если база данных достаточно стабильна и ее содержимое практически не меняется, то построение вторичных индексов действительно может ускорить процесс обработки информации.

Модели физической организации данных при бесфайловой организации

Файловая структура и система управления файлами являются прерогативой операционной среды, поэтому принципы обмена данными подчиняются законам операционной системы. По отношению к базам данных эти принципы могут быть далеки от оптимальности. СУБД подчиняется несколько иным принципам и стратегиям управления внешней памятью, чем те, которые поддерживают операционные среды для большинства пользовательских процессов или задач.

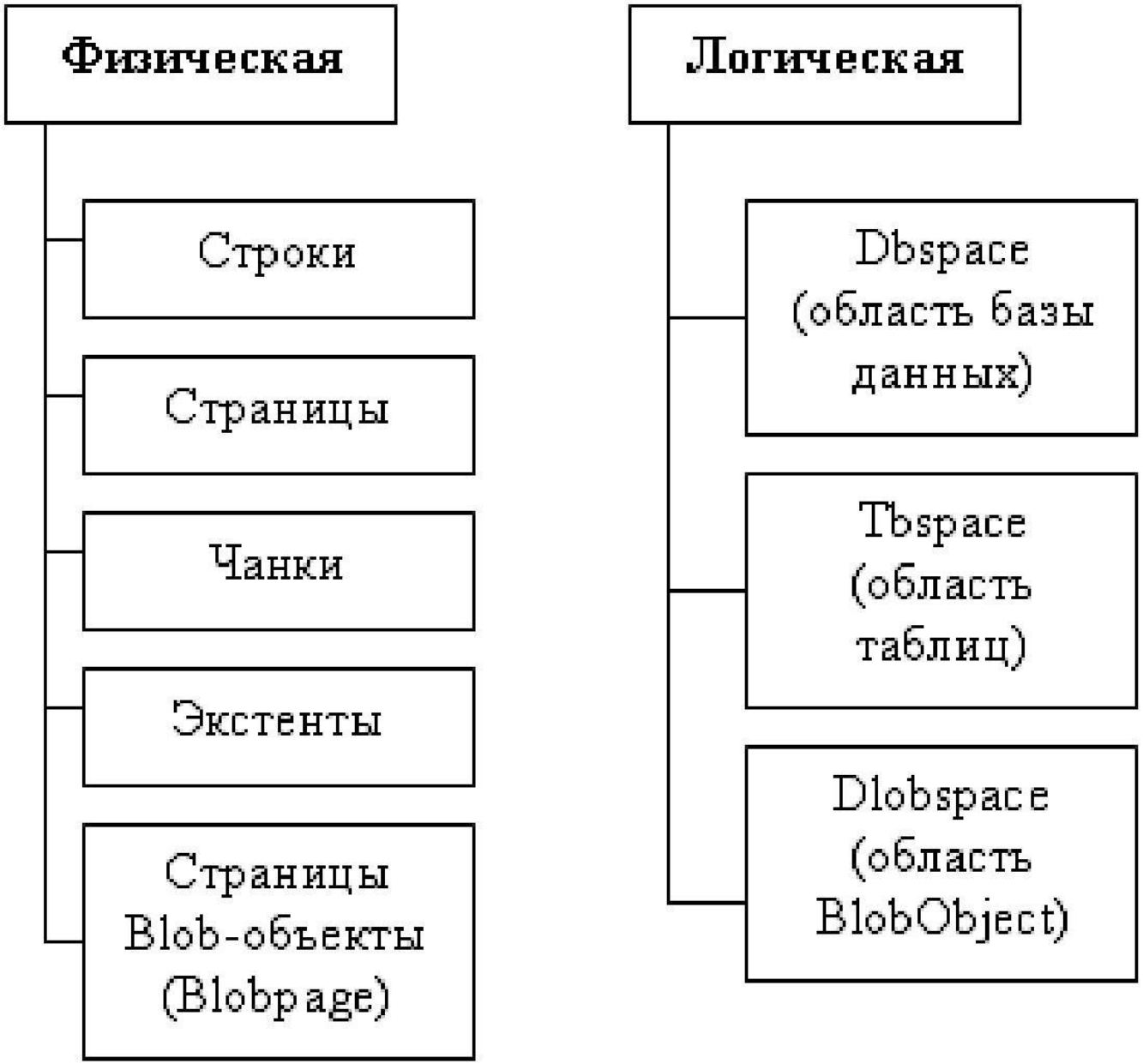
Это и послужило причиной того, что СУБД взяли на себя непосредственное управление внешней памятью. При этом пространство внешней памяти предоставляется СУБД полностью для управления, а операционная среда не получает непосредственного доступа к этому пространству.

Физическая организация является в настоящий момент наиболее динамичной частью СУБД. Стремительно расширяются возможность устройств внешней памяти, дешевеет оперативная память, увеличивается ее объем и поэтому изменяются сами принципы организации физических структур данных.

И можно предположить, что и в дальнейшем эта часть современных СУБД будет постоянно меняться. Поэтому при рассмотрении моделей данных, используемых для физического хранения и обработки, мы коснемся только наиболее общих принципов и тенденций.

При распределении дискового пространства рассматриваются две схемы структуризации: физическая, которая определяет хранимые данные,

и логическая, которая определяет некоторые логические структуры, связанные с концептуальной моделью данных


Физическая
Строки
Страницы
Чанки
Экстенты
Страницы Blob-объекты (Blobpage)
Логическая
Dbspace (область базы данных)
Tbspace (область таблиц)
Dlobspace (область BlobObject)

Чанк (chank) — представляет собой часть диска, физическое пространство на диске, которое ассоциировано одному процессу (on line процессу обработки данных).

Чанком может быть назначено неструктурированное устройство, часть этого устройства, блочно-ориентированное устройство или просто файл UNIX.

Чанк характеризуется маршрутным именем, смещением (от физического начала устройства до начальной точки на устройстве, которая используется как чанк), размером, заданным в Кбайтах или Мбайтах.

При использовании блочных устройств и файлов величина смещения считается равной нулю.

Логические единицы образуются совокупностью экстентов, то есть таблица моделируется совокупностью экстентов.

Экстент — это непрерывная область дисковой памяти.

Для моделирования каждой таблицы используется 2 типа экстентов: первый и последующие.

Первый экстент задается при создании нового объекта типа таблица, его размер задается при создании. EXTENTSIZE — размер первого экстента, NEXT SIZE — размер каждого следующего экстента.

Минимальный размер экстента в каждой системе свой, но в большинстве случаев он равен 4 страницам, максимальный — 2 Гбайтам.

Новый экстент создается после заполнения предыдущего и связывается с ним специальной ссылкой, которая располагается на последней странице экстента. В ряде систем экстенты называются сегментами, но фактически эти понятия эквиваленты.

При динамическом заполнении БД данными применяется специальный механизм адаптивного определения размера экстентов.

Внутри экстента идет учет свободных станиц.

Между экстентами, которые располагаются друг за другом без промежутков, производится своеобразная операция конкатенации, которая просто увеличивает размер первого экстента.

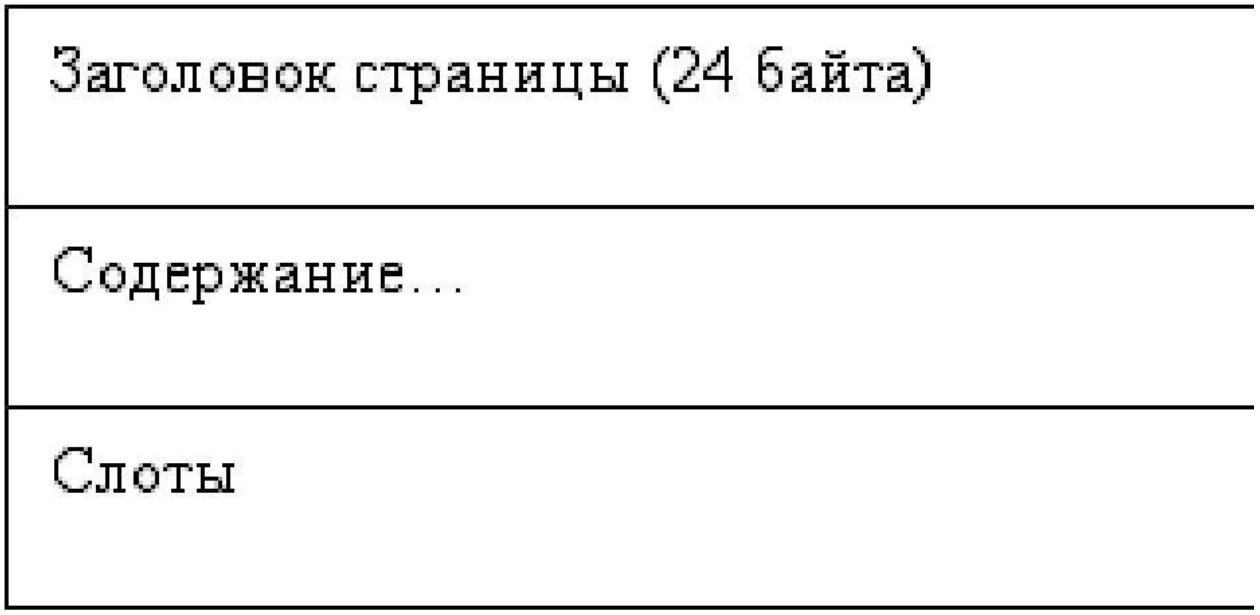
Механизм удвоения размера экстента: если число выделяемых экстентов для процесса растет в пропорции, кратной 16, то размер экстента удваивается каждые 16 экстентов.
Экстенты состоят из четырех типов страниц: страницы данных, страницы индексов, битовые страницы и страницы blob-объектов.
Blob — это сокращение Binary Larg Object, и соответствует оно неструктурированным данным.
В ранних СУБД такие данные относились к типу Memo.
В современных СУБД к этому типу относятся неструктурированные большие текстовые данные, картинки, просто наборы машинных кодов. Для СУБД важно знать, что этот объект надо хранить целиком, что размеры этих объектов от записи к записи могут резко отличаться и этот размер в общем случае неограничен.
Основной единицей осуществления операций обмена (ввода-вывода) является страница данных.
Все данные хранятся постранично.
При табличном хранении данные на одной странице являются однородными, то есть станица может хранить только данные или только индексы.

Слот — это 4-байтовое слово, 2 байта соответствуют смещению строки на странице и 2 байта — длина строки.

Слоты характеризуют размещение строк данных на странице.

На одной странице хранится не более 255 строк. В базе данных каждая строка имеет уникальный идентификатор в рамках всей базы данных, часто называемый RowID — номер строки, он имеет размер 4 байта и состоит из номера страницы и номера строки на странице. Под номер страницы отводится

3 байта, поэтому при такой идентификации возможна адресация к 16 777 215 страниц



При упорядочении строк на страницах не происходит физического перемещения строк, все манипуляции происходят со слотами.

При переполнении страниц создается специальный вид страниц, называемых страницами остатка. Строки, не уместившиеся на основной странице, связываются (линкуются) со своим продолжением на страницах остатка с помощью ссылок-указателей «вперед» (то есть на продолжение), которые содержат номер страницы и номер слота на странице.

Страницы индексов организованы в виде B-деревьев. Страницы blob предназначены для хранения слабоструктурнрованной информации, содержащей тексты большого объема, графическую информацию, двоичные коды. Эти данные рассматриваются как потоки байтов произвольного размера, в страницах данных делаются ссылки на эти страницы.

Битовые страницы служат для трассировки других типов страниц. В зависимости от трассируемых страниц битовые страницы строятся по 2-битовой или 4-битовой схеме. 4-битовые страницы служат для хранения сведений о столбцах типа Varchar, Byte, Text, для остальных типов данных используются 2-битовые страницы.

Битовая структура трассирует 32 страницы. Каждая битовая структура представлена двумя 4-байтными словами. Каждая i-я позиция описывает одну i-ю страницу. Сочетание разрядов в i-х позициях двух слов обозначает состояние данной страницы: ее тип и занятость.

При обработке данных СУБД организует специальные структуры в оперативной памяти, называемые разделяемой памятью, и специальные структуры во внешней памяти, называемые журналами транзакций. Разделяемая память служит для кэширования данных при работе с внешней памятью с целью сокращения времени доступа, кроме того, разделяемая память служит для эффективной поддержки режимов одновременной параллельной работы пользователей с базой данных.

Журнал транзакций служит для управления корректным выполнением транзакций.