Что такое система баз данных и какие компоненты в ее составе.

Системы баз данных

Система баз данных (СБД) — компьютеризированная система для хранения информации в БД.

Компоненты СБД:

- пользователи;

- прикладные программы;

- базы данных;

- система управления базами данных.

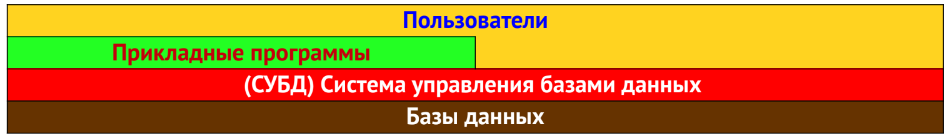


Рисунок 1 – Компоненты системы баз данных

Пользователи

I) Пользователи делятся на 4 группы:

1) администраторы;

2) разработчики баз данных;

3) прикладные программисты;

4) пользователи.

Администраторы

- администраторы данных;

- администраторы баз данных.

Администраторы данных — отвечают за управление данными, включая планирование БД, раз-

работку и сопровождение стандартов, бизнес правил (описывают основные характеристики дан-

ных с точки зрения организации) и деловых процедур, а также концептуальное и логическое

проектирование БД;

Администраторы баз данных — отвечают за физическую реализацию БД, включая физическое

проектирование, обеспечение безопасности и целостности данных, а также обеспечение макси-

мальной производительности приложений и пользователей (более технический характер по срав-

нению с администратором данных).

Разработчики баз данных

- разработчики логической базы данных;

- разработчики физической базы данных.

разработчики логической базы данных занимаются идентификацией данных, связей между дан-

ными и устанавливают ограничения, накладываемые на хранимые данные — (ответ на вопрос ЧТО?);

разработчики физической базы данных по готовой логической модели создают физическую ре-

ализацию (формирование таблиц, выбор структур хранения, методов доступа, мер защиты) — (ответ на вопрос КАК?).

Прикладные программисты

Создают приложения предоставляющие пользователям необходимые функциональные воз-

можности (действия над базой данных).

Пользователи (клиенты БД)

- наивные пользователи — осуществляют доступ к БД через прикладные программы;

- опытные пользователи — могут осуществлять доступ к БД с использованием языков запросов

или создавать собственные прикладные программы.

Иерархическая модель данных. Достоинства и недостатки.

Иерархическая модель данных

Иерархическая модель данных возникла из практики работы с данными и создавалась по сути дела

программистами. Стандартов на иерархические модели не существует.

Наиболее распространенной СУБД реализующей иерархическую модель являлась СУБД IMS.

Ключевым фактором при создании IMS была необходимость управления большим количеством де-

талей, связанных друг с другом иерархическим образом (детали узлы модули и т.п.). → →

Иерархическая модель – модель данных на основе записей, в которой все отношения между объек-

тами структурированы в виде деревьев:

1) все записи хранятся в общей древовидной структуре, имеющей одну корневую запись;

2) каждый узел дерева – запись, описывающая экземпляр объекта;

3) соединения между записями в дереве определяют связи между экземплярами объектов.

В иерархической модели связи называются отношениями «предок-потомок», а записи – сегментами.

Потомок может быть связан только с одним предком и для реализации связи с разными предками

требуется построение разных деревьев.

Отношение «предок-потомок» реализуют связи «один-ко-многим»;

Иерархическая БД – совокупность деревьев.

Все отношения, которые могут быть представлены в иерархической модели данных – бинарные.

Бинарное отношение — отношение между двумя множествами и , то есть всякое подмножество

декартова произведения этих множеств: . Бинарное отношение на множестве A любое ‒

подмножество (равенство, неравенство, эквивалентность, отношение порядка, ...).

Алгоритм перевода ER-диаграммы в иерархическую модель

Алгоритм перевода ER-диаграммы в иерархическую модель данных представлен следующими ша-

гами:

1) Для каждого объекта ER-диаграммы создается тип сегмента в иерархической модели.

Название типа сегмента соответствует названию объекта.

Атрибуты объекта задают поля записи сегмента (название, тип данных).

2) В диаграммах деревьев отношения один-ко-многим задаются с помощью отношений «предок-

потомок», причем объект со стороны много в связи становится потомком, а объект со стороны один –

предком.

3) Для реализации отношений многие-ко-многим создается два разных отношения «предок-пото-

мок», в этих отношениях один и тот же объект выступает в качестве предка (первое отношение «предок-

потомок») и в качестве потомка (второе отношение «предок-потомок»).

В одном из таких отношений «предок-потомок» реальные данные для экономии места заменяют на

ссылки (указатели) на данные из второго отношения.

4) Если отношение мощности многие-ко-многим на ER-диаграмме обладает атрибутом, то в диа-

граммах деревьев создается сегмент пересечения, который располагается между предком и потомком и

содержит значение атрибута.

Основное достоинство иерархической модели данных — представление данных в виде иерархиче-

ских связей и автоматическое ограничения целостности (не существует потомков без родителей).

К основным недостаткам иерархической модели данных относятся:

- нет разделения логической и физической модели данных;

- непредвиденные запросы требуют реорганизации БД;

- для не иерархических связей описание сильно усложняется и требует много памяти;

- сложность составления программ обработки данных;

- нет стандарта на построение подобных систем.

Сетевая модель данных. Достоинства и недостатки.

Сетевая модель данных

Сетевая модель данных – модель данных на основе записей, в которой данные представлены сете-

выми структурами типа направленного графа.

Разница между иерархической моделью данных и сетевой состоит в том, что в иерархических струк-

турах запись-потомок должна иметь в точности одного предка, а в сетевой структуре данных у потомка

может иметься любое число предков.

Узлы в такой модели представляют объекты и называются – наборами типов записей данных.

Ребра графа представляют отношения между объектами и называются наборами типов связей.

Наборы выражают отношения «один-ко-многим» или «один-к-одному» между двумя типами запи-

сей, при этом выделяют тип записи владельца – со стороны «один» связи, и тип записи члена набора –

со стороны «много» связи.

Сетевая БД состоит из наборов отношений.

Все отношения, которые могут быть представлены в сетевой модели данных – бинарные.

Алгоритм перевода ER-диаграммы в сетевую модель данных:

1) Для каждого объекта ER-диаграммы создается тип записей в сетевой модели.

Название типа записей соответствует названию объекта.

Атрибуты объекта задают поля типа записи (название, тип данных).

2) Для отношения «один-ко-многим» тип записи со стороны «один» отношения становится вла-

дельцем набора, а тип записи со стороны «много» отношения – членом набора.

Если мощность отношения «один-к-одному», то типы записей в наборе выбираются произвольно.

3) Для каждого n-арного отношения, где n > 2, создается связывающая запись, которая становится

типом записи члена для n наборов.

Владельцами наборов назначаются типы записей со стороны «один», полученных в результате

преобразований отношений «один-ко-многим».

4) Для реализации отношений «многие-ко-многим» создается связывающая запись, которая стано-

вится типом записи члена для двух наборов, владельцами которых являются типы записей, соответству-

ющие объектам связи.

5) Если отношение мощности «многие-ко-многим» на ER-диаграмме обладает атрибутом, то этот

атрибут добавляется в связывающую запись.

Основные достоинства сетевой модели данных:

- высокая производительность в статичных системах с простой структурой;

- наличие ограничений целостности в отношении «зависших» потомков;

- наличие стандарта (DBTG).

К основным недостаткам сетевой модели данных относятся:

- структура БД определяется запросами пользователей смена запросов ведет к реорганизации всей БД; ‒

- производительность непредвиденных приложений низкая – нет возможности расширения;

- определение схемы влияет на физический уровень БД (формат записей);

- сложность составления программ обработки данных;

- сложно установить ограничения на образование связей.

Реляционная модель данных. Достоинства и недостатки.

Сторонники нормализации ввели собственную терминологию и называют простые вещи специаль-

ными словами.

Таблица такого вида, как представлено выше, называется отношением.

База данных, построенная с помощью отношений, называется реляционной базой данных.

Таким образом, реляционная база данных строится из «плоских» (двумерных) наборов элементов

данных.

Отношение, или таблица,— это набор кортежей.

Кортежи являются n-мерными, т. е. если таблица имеет n столбцов, отношение называется отноше-

нием степени n.

Отношение степени 2 называется бинарным, отношение степени 3 — тернарным, а отношение сте-

пени n — n-арным.

Набор значений элементов данных одного типа, т. е. один столбец таблицы, называется доменом.

Столбец с номером k называется k-м доменом отношения.

В математике определяется как отношение, заданное на множествах , , ..., (не обязатель-

но различных), если оно представляет собой набор кортежей, таких, что первый элемент каждого корте-

жа есть элемент из , второй — элемент из и т. д.

Для описания таких отношений и операций над ними существуют точные математические обозначе-

ния, основанные на алгебре отношений или исчислении отношений (реляционная алгебра).

Реляционная алгебра — замкнутая система операций над отношениями в реляционной модели

данных.

Операции реляционной алгебры также называют реляционными операциями.

Основные функции СУБД.

СУБД (DBMS) — программное обеспечение, с помощью которого пользователи могут опреде-

лять, создавать и поддерживать БД, а также осуществлять к ней контролируемый доступ.

Фактически это прослойка между БД и пользователем (прикладной программой) для скрытия

особенностей хранения и управления данными (абстрагирование).

Функции СУБД

Для СУБД характерно наличие следующих функций и служб (сервисов) (первые восемь — Кодд,

1982):

1) СУБД должна предоставлять пользователям возможность сохранять, изменять и обновлять

данные в БД. Это основная функция СУБД.

Способ реализации этой функции должен скрывать от пользователя детали физической реали-

зации системы (абстрагирование).

2) СУБД должна иметь доступный конечным пользователям каталог, в котором хранится описа-

ние элементов данных — системный каталог — хранилище информации, описывающей данные БД

(метаданные):

- имена, типы и размеры элементов данных;

- имена связей;

- накладываемые на данные ограничения поддержки целостности;

- имена санкционированных пользователей;

- описание схем архитектуры БД;

- статистические данные (счетчики событий);

Зачем нужен системный каталог:

- централизованный контроль доступа к данным;

- определение смысла данных для понимания их предназначения (семантика);

- упрощается определение владельца данных или прав доступа к ним;

- облегчается протоколирование изменений БД;

- последствия изменений могут быть определены до их внесения в БД (усиление безопасности

и целостности данных);

3) СУБД должна иметь механизм, который гарантирует выполнение либо всех операций обнов-

ления данной транзакции, либо ни одной из них.

Транзакция — набор действий выполняемых пользователем (прикладной программой) с целью

доступа или изменения БД (при этом в БД вносится сразу несколько изменений).

Если во время внесения изменений происходит сбой, например, вносимые данные вызывают

нарушение целостности данных, то все изменения должны быть отменены (возвращение в непро-

тиворечивое состояние БД).

4) СУБД должна иметь механизм, который гарантирует корректное обновление БД при парал-

лельном выполнении операций обновления многими пользователями (реализуется в рамках под-

держки транзакций).

5) СУБД должна предоставлять средства восстановления БД на случай ее повреждения или

разрушения (реализуется в рамках поддержки транзакций).

6) СУБД должна иметь механизм, гарантирующий возможность доступа к БД только санкциони-

рованных пользователей. Сокрытие части ненужных для пользователя данных и защита от любого

несанкционированного доступа.

7) СУБД должна обладать способностью к интеграции с коммуникационным программным

обеспечением.

8) СУБД должна обладать инструментами контроля за тем, чтобы данные и их изменения соот-

ветствовали заданным правилам. Это называется целостностью базы данных, выражающееся в

корректности и непротиворечивости хранимых данных.

Целостность является одним из типов защиты БД и выражается в виде ограничений или правил

сохранения непротиворечивости данных при их изменении — если ограничения нарушаются, из-

менения в БД не вносятся.

9) СУБД должна обладать инструментами поддержки независимости программ от фактической

структуры БД.

Эта независимость достигается за счет реализации механизма поддержки представлений,

например, в рамках трехуровневой концепции.

Достичь полной логической независимости от данных очень сложно, поскольку СУБД легко

адаптируется к добавлению нового объекта (атрибута, связи и т.п.), но не к их удалению.

Некоторые СУБД запрещают вносить изменения в существующие компоненты концептуальной

схемы.

10) СУБД должна предоставлять некоторый набор различных вспомогательных служб, предна-

значенных, в том числе, для эффективного администрирования БД – импорт данных, резервное

сохранение, реорганизация индексов, сборка мусора, перераспределение памяти.

Реляционная модель данных. Кортеж. Отношение.

Замечание о терминологии

SQL и реляционная модель – вовсе не одно и то же.

Дейт уточняет:

«Если знания о реляционной модели проистекают исключительно из знания SQL, то эти

знания могут оказаться неверными. Отсюда, в частности, следует, что кое-что из уже

известного следует забыть.»

В рамках реляционной теории употребляются формальные термины — «отношение», «кортеж» и

«атрибут».

В SQL эти термины не используются, там применяются более «дружественные» слова — таблицы,

строка и столбец/колонка.

Истина заключается в том, что отношение – это не таблица, кортеж – не строка, а атрибут – не

столбец.

SQL пытался упростить один набор терминов, но он сделал все возможное для усложнения друго-

го. Имеются в виду термины оператор, функция, процедура, подпрограмма и метод — все они обозна-

чают по существу одно и то же (быть может, с незначительными вариациями).

Реляционная модель для всего вышеперечисленного использует термин оператор.

Представление данных в виде деревьев, сетевых и списковых структур в общем случае препят-

ствует многим изменениям данных, необходимым при росте базы.

Рост базы может привести к нарушению логического представления данных и, как следствие, к

изменениям в прикладных программах.

Избежать растущей сложности древовидных и сетевых структур можно с помощью метода, назы-

ваемого нормализацией. Этот метод был разработан и активно применялся Коддом.

Нормализация относится к представлению данных пользователем или к логическому описанию

данных; она не связана с физическим представлением данных.

Кортеж

Термин кортеж в отношении к базам данных приблизительно соответствует понятию строки в

таблице (так же, как термин отношение приблизительно соответствует понятию таблицы).

Соответственно, кортеж состоит из атрибутов.

Кортежем вообще называется набор взаимосвязанных величин.

Кортеж, содержащий две величины, называется парой (pair).

Кортеж, содержащий N величин, называется N-кортежем.

Таблица состоит из набора кортежей, каждый из которых содержит элементы данных одинако-

вого типа. Она, таким образом, представляет собой двумерную матрицу элементов данных.

Элементы данных обычно обрабатываются группами.

В различных системах программного обеспечения эти группы называются по-разному (сегмент,

кортеж).

Общепринятым является термин запись.

Отношение

Отношение — это формальное название таблицы.

Например, можно сказать, что база данных отделов и служащих, представленная выше, содер-

жит два отношения.

В настоящее время в неформальном контексте термины отношение и таблица принято считать

синонимами. На практике в подобном контексте термин таблица используется гораздо чаще, чем

термин отношение.

Смысл отношений

Cтолбцы в отношениях связаны с типами данных. Реляционная модель допускает неограниченный

набор типов [данных]. Это означает, что пользователи могут как применять определяемые системой или

встроенные типы, так и определять собственные. Например, определять типы можно так («...» здесь за-

меняет сами определения, которые на сей момент не важны).

TYPE EMP\_NO ... ;

TYPE NAME ... ;

TYPE DPT\_NO ... ;

TYPE MONEY ... ;

Тип ЕМР\_NO, например, можно рассматривать, как множество всевозможных табельных номеров,

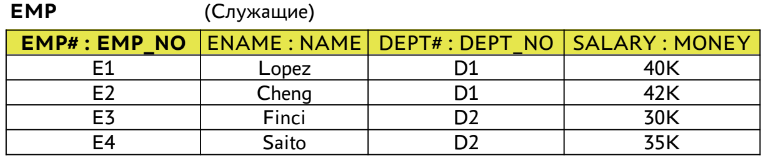
тип NAME — как множество всевозможных имен и т.д.

Каждое отношение (каждое значение отношения) состоит из двух частей:

1) набор пар (<имя\_столбца> : <имя\_типа>) — заголовок;

2) набор строк, согласованных с этим заголовком — тело.

Пример — часть отношения ЕМР, дополненного обозначениями типов столбцов.



Хотя на практике компоненты заголовка <имя\_типа> обычно опускают, необходимо учитывать, что

концептуально они всегда присутствуют.

Отношения можно представлять также следующим образом:

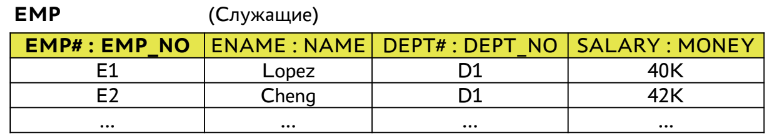
1) В определенном отношении R его заголовок представляет собой некоторый предикат (функция,

возвращающая значения истинности и принимающая ряд формальных параметров).

2) Каждая строка в теле отношения R представляет собой определенное истинное высказывание,

полученное из предиката путем подстановки определенных значений фактических параметров соответ-

ствующего типа вместо формальных параметров этого предиката (путем конкретизации).



Для отношения EMP предикат будет следующим:

«Служащий с табельным номером ЕМР# и фамилией ENAME работает в отделе с номером DEPT# и по-

лучает зарплату SALARY».

Здесь формальные параметры – ЕМР#, ENAME, DEPT# И SALARY. Они соответствуют четырем

столбцам переменной отношения ЕМР.

Соответствующие истинные утверждения:

«Служащий с табельным номером Е1 и фамилией Lopez работает в отделе с номером D1 и получает

зарплату 40 тыс. долл. в год.»

«Служащий с номером Е2 и фамилией Cheng работает в отделе с номером D1 и получает зарплату

42 тыс. долл. в год.»

Иными словами, типы — это объекты (множества объектов), которые могут стать предметом обсу-

ждения;

отношения — это факты (множества фактов), касающиеся объектов, которые могут стать предметом

обсуждения.

Типы связаны с отношениями точно так же, как существительные связаны с предложениями на

естественном языке.

В примере, приведенном выше, предметом обсуждения являются:

- табельные номера служащих EMP\_NO;

- имена NAME;

- номера отделов DEPT\_NO;

- значения денежных сумм MONEY.

Об обсуждаемом же предмете можно привести истинное высказывание следующего вида:

«Служащий с определенным табельным номером имеет определенное имя, работает в определенном

отделе и получает определенную зарплату».

Атомарность. согласованность, изолированность, стойкость.

Атомарность гарантирует, что никакая транзакция не будет зафиксирована в системе частично.

Будут либо выполнены все ее подоперации, либо не выполнено ни одной. Поскольку на прак-

тике невозможно одновременно и атомарно выполнить всю последовательность операций внутри

транзакции, вводится понятие «отката» (rollback) – если транзакцию не удается полностью завер-

шить, результаты всех ее до сих пор произведенных действий будут отменены и система вернется

во «внешне исходное» состояние — со стороны будет казаться, что транзакции и не было. Возмож-

но, счетчики, индексы и другие внутренние структуры могут измениться, но, если СУБД разработана

без ошибок, это не повлияет на внешнее ее поведение.

Согласованность является более широким понятием. Например, в банковской системе может

существовать требование равенства суммы, списываемой с одного счета, сумме, зачисляемой на

другой. Это бизнес-правило и оно не может быть гарантировано только проверками целостности,

его должны соблюсти программисты при написании кода транзакций. Если какая-либо транзакция

произведет списание, но не произведет зачисления, то система останется в некорректном состоя-

нии и свойство согласованности будет нарушено.

В ходе выполнения транзакции согласованность не требуется. В вышеприведенном примере

списание и зачисление будут, скорее всего, двумя разными подоперациями в рамках одной тран-

закции и между их выполнением внутри транзакции будет видно несогласованное состояние си-

стемы.

При выполнении требования изоляции никаким другим транзакциям эта несогласованность не

будет видна. А атомарность гарантирует, что транзакция либо будет полностью завершена, либо ни

одна из операций транзакции не будет выполнена. Тем самым эта промежуточная несогласован-

ность является скрытой.

Изолированность – во время выполнения транзакции параллельные транзакции не должны

оказывать влияния на ее результат. Изолированность — требование дорогое, поэтому во многих ре-

альных БД существуют режимы, не полностью изолирующие транзакцию.

Стойкость – независимо от проблем на нижних уровнях (к примеру, обесточивание системы или

сбои в оборудовании) изменения, сделанные успешно завершенной транзакцией, должны остаться

сохраненными после возвращения системы в работу. Другими словами, если пользователь получил

подтверждение от системы, что транзакция выполнена, он может быть уверен, что сделанные им

изменения не будут отменены из-за какого-либо сбоя.

Как получить реляционную модель из ER-модели?

Порядок перевода ER-модели в реляционную модель выполняется с помощью алгоритма, состоящего из шести шагов:

*Шаг 1.* Каждый *объект* на ER-диаграмме превращается в реляционное отношение (далее для краткости – таблицу), имя объекта становится именем таблицы (*следует указать понятное имя!)*.

*Шаг 2.* Каждый *атрибут объекта* становится столбцом таблицы с тем же именем (*также следует указать понятное имя!*) и требуемым типом данных.

*Шаг 3.* Уникальные (*ключевые*) атрибуты объекта превращаются в *первичный ключ* таблицы (при наличии нескольких возможных уникальных идентификаторов, выбирается наиболее подходящий для использования; если таковых атрибутов нет или они плохо подходят для долговременного использования в БД, то желательно создать *суррогатный* ключ). *Каждая таблица в БД должна иметь первичный ключ!*

*Шаг 4.* Связи *«один-ко-многим»* (в том числе и связи *«один-к-одному»*) становятся *ссылками* в уже существующих таблицах, при этом внешний ключ добавляется в виде столбца (столбцов) в таблицу, соответствующую объекту со стороны *«многие»* связи. *Внешние ключи должны ссылаться только на первичные ключи целевых таблиц!*

*Шаг 5.* Связи *«многие-ко-многим»* реализуются каждая через отдельную промежуточную таблицу:

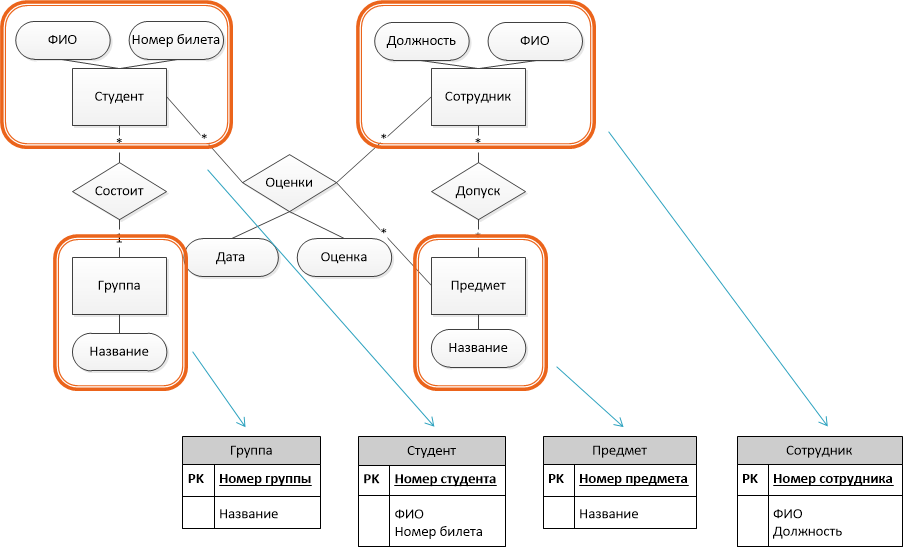
* эта промежуточная таблица обязательно будет содержать столбцы внешних ключей, ссылающиеся на соответствующие объекты связи;
* первичный ключ промежуточной таблицы для исключения дубликатов должен быть составным и включать в себя все внешние ключи на объекты, участвующие в связи.

*Шаг 6.* Если связь имеет *дополнительные* атрибуты, то, как и в случае атрибутов объектов, они становятся столбцом соответствующей таблицы:

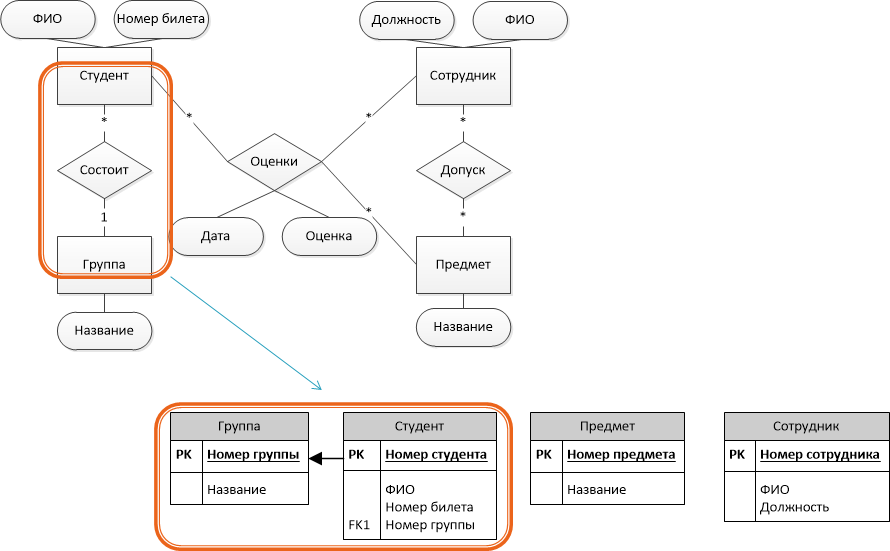
* для связей *«один-ко-многим»* (встречаются на практике редко) – в таблице со стороны *«многие»* (там, где расположен внешний ключ);
* для связей *«многие-ко-многим»* – в промежуточной таблице (при этом атрибуты, расширяющие комбинацию в связи (например, *«дата»*), также должны войти в состав *составного* первичного ключа промежуточной таблицы).

Ниже на рисунках приведены этапы выполнения шагов алгоритма для простой ER-диаграммы:

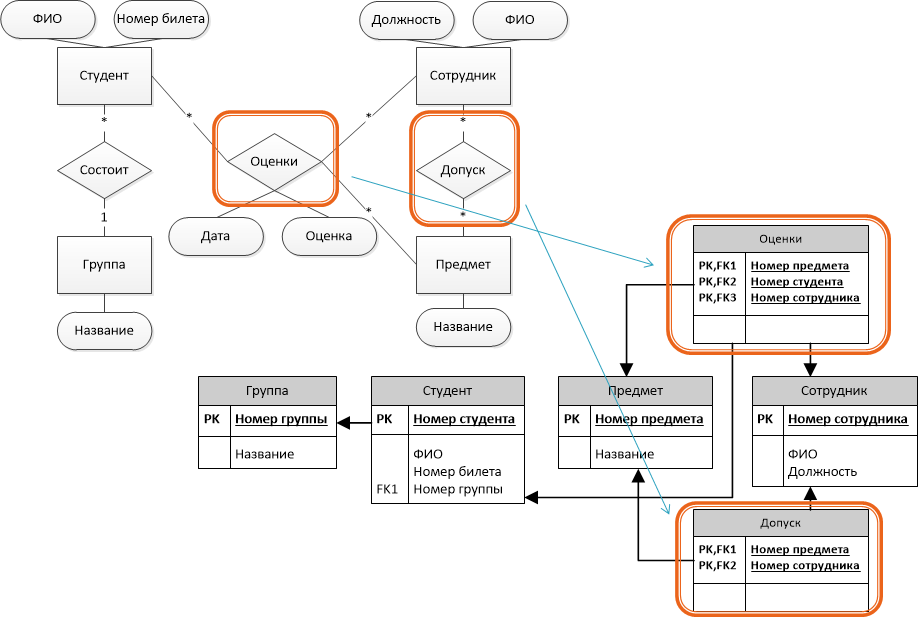
* результат выполнения шагов 1 – 3 алгоритма:



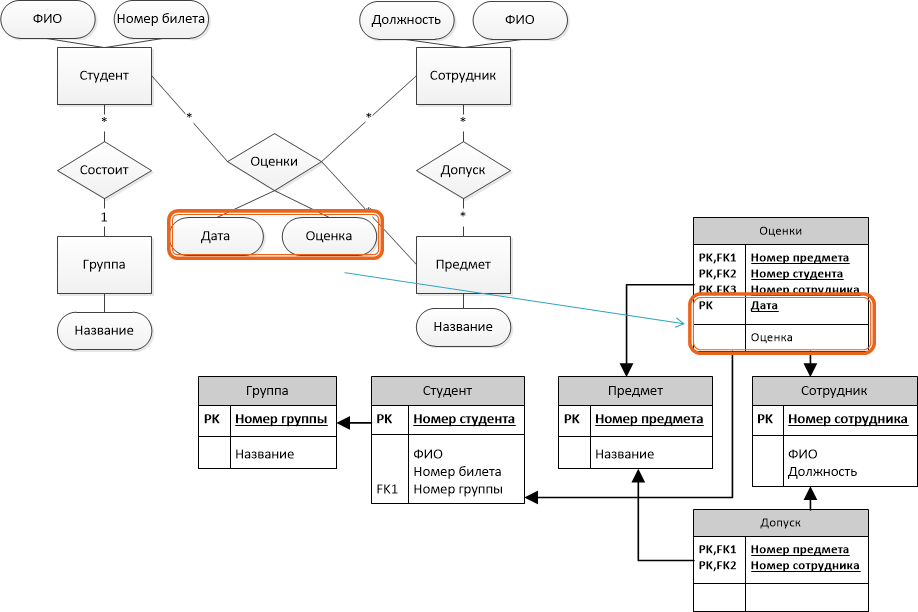
* результат выполнения шага 4 алгоритма



* результат выполнения шага 5 алгоритма:



* результат выполнения шага 6 алгоритма:



ER-модель. Сущности и связи. Примеры.

Модель «Сущность-связь»

Модель сущность-связь (или ER-модель — Entity-Relationship model) описывает взаимосвязан-

ные вещи, представляющие интерес в конкретной области знаний.

Базовая модель ER состоит из типов сущностей (которые классифицируют эти самые важные

вещи) и определяет отношения (связи), которые могут существовать между сущностями (экземпля-

рами этих типов сущностей).

В разработке программного обеспечения модель ER обычно формируется для представления

вещей, о которых необходимо помнить при выполнении бизнес-процессов. В этом качестве ER-

модель становится абстрактной моделью данных, которая определяет структуру данных или ин-

формации, которая может быть реализована в базе данных, обычно реляционной.

В процессе проектирования баз данных схема, созданная на основе ER-модели, преобразуется

в конкретную схему базы данных на основе выбранной модели данных — реляционной, объектной,

иерархической, сетевой, ...

ER-модель представляет собой формальную конструкцию, которая сама по себе не предписы-

вает никаких графических средств её визуализации и может быть просто словесным описанием в

виде текста.

В качестве ``стандартной графической нотации'', с помощью которой можно визуализировать ER-

модель, используется диаграмма «сущность-связь» (Entity-Relationship diagram, ERD, ER-диаграмма).

Понятия «ER-модель» и «ER-диаграмма» обычно не различают, однако для визуализации ER-

моделей могут быть использованы и другие графические нотации, либо вместо графического пред-

ставления может использоваться текстовое описание.

Модель была разработана Питером Ченом для проектирования базы данных и опубликована в

статье 1976 года с вариантами согласно данной идеи, существовавшими ранее. Он же предложил и

самую популярную графическую нотацию для ER-модели.

ER-диаграмма

Схема «сущность-связь» (также ERD или ER-диаграмма) — это разновидность блок-схемы, где

показано, как разные «сущности» (люди, объекты, концепции и так далее) связаны между собой

внутри системы.

ER-диаграммы чаще всего применяются для проектирования и отладки реляционных баз дан-

ных в сфере образования, исследования и разработки программного обеспечения и информаци-

онных систем для бизнеса.

ER-диаграммы (или ER-модели) полагаются на стандартный набор символов, включая прямо-

угольники, ромбы, овалы и соединительные линии, для отображения сущностей, их атрибутов и

связей.

Эти диаграммы устроены по тому же принципу, что и грамматические структуры — сущности

выполняют роль существительных, а связи — глаголов.

Множество — набор, совокупность каких-либо (вообще говоря любых) объектов — элементов

этого множества.

Два множества равны тогда и только тогда, когда содержат в точности одинаковые элементы.

Отношения между элементами множества

N:1 каждый элемент множества R является образом хотя бы одного элемента множества L.

1:1

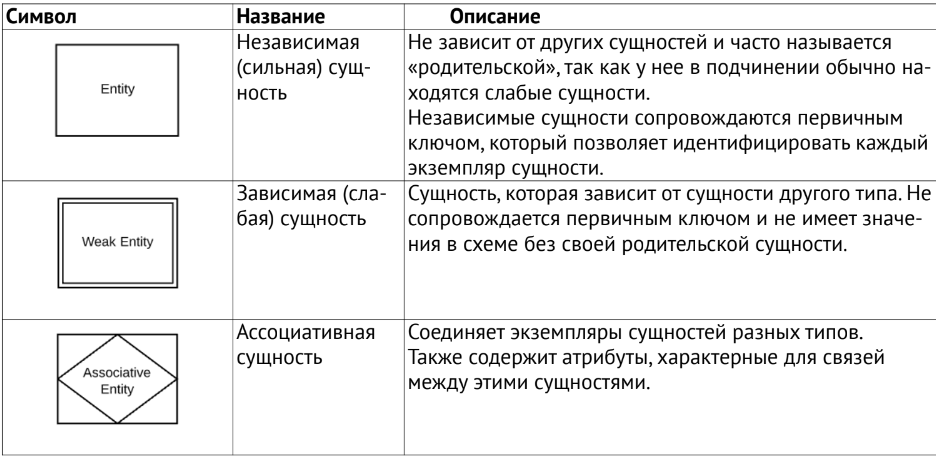
Символы ERD-сущностей

Под понятием «сущности» подразумеваются объекты или понятия, несущие важную информа-

цию. С точки зрения грамматики, они, как правило, обозначаются существительными, например,

«товар», «клиент», «заведение» или «промоакция». Ниже представлены три наиболее распростра-

ненных типа сущностей, используемых в ER-диаграммах.



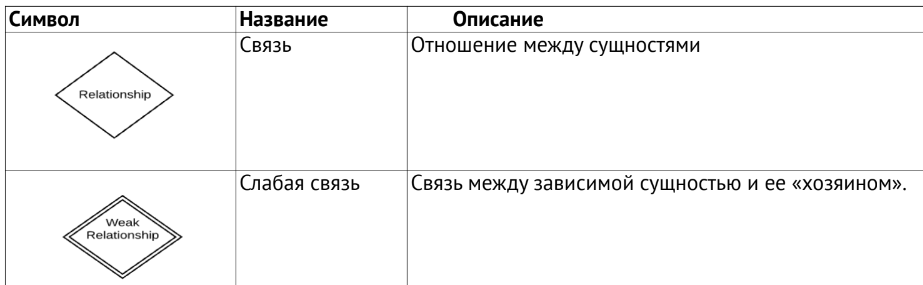
Символы ERD-связей

Связи используются в схемах «сущность-связь» для обозначения взаимодействия между двумя

сущностями.

Грамматически связи, как правило, выражаются глаголами, например, «назначить», «закрепить»,

«отследить», и несут полезную информацию, которую невозможно получить, опираясь только на типы сущностей.



Основные операторы реляционной модели – сокращение, проекция, объединение.

Аспект обработки

В распоряжении пользователя имеются операторы манипулирования таблицами, например, пред-

назначенные для поиска данных, которые генерируют новые таблицы на основании уже имеющихся

и среди которых есть, по крайней мере, операторы:

- сокращения (restrict);

- проекции (project);

- объединения (join).

Операция сокращения извлекает указанные строки из таблицы. В названии этой операции

подразумевается, что кардинальность ее результата меньше или равна кардинальности исходной та-

блицы.

Операцию сокращения иногда называют выборкой (select). Однако, соответствующий оператор со-

ответствует оператору SELECT языка SQL не полностью.

Операция проекции предназначена для извлечения определенных столбцов из таблицы.

Операция соединения предназначена для получения комбинации двух таблиц на основе общих

значений в общих столбцах.

Операция соединения

DEPT EMP

DEPT# DNAME BUDGET EMP# ENAME DEPT# SALARY

D1 Marketing 10M E1 Lopez D1 40K

D2 Development 12M E2 Cheng D1 42K

D3 Research 5M E3 Finci D2 30K

Δ E4 Saito D2 35K

DEPT.DNO ссылается на EMP.DNO

В таблицах DEPT и ЕМР есть общий столбец DEPT#, а следовательно, для этих таблиц можно вы-

полнить операцию соединения на основе общих значений в этом столбце. При выполнении данной

операции строка таблицы DEPT соединяется со строкой таблицы ЕМР и образуется более длинная

строка, но подобное происходит тогда и только тогда, когда эти две строки имеют одинаковое значе-

ние поля DEPT#.

Например, можно соединить в результирующую строку следующие строки таблиц DEPT и ЕМР

(названия столбцов приведены для наглядности). Это возможно, так как в общем столбце рассматри-

ваемых строк имеется одно и то же значение D1.

DEPT# DNAME BUDGET EMP# ENAME SALARY

D1 Marketing 10M E1 Lopez 40K

D1 Marketing 10M E2 Cheng 42K

D2 Development 12M E3 Finci 30K

D2 Development 12M E4 Saito 35K

Общий результат состоит из множества всех таких соединенных строк.

Столбец DEPT# в каждой результирующей строке встречается один раз, а не два.

Следует также отметить, что в поле DEPT# таблицы ЕМР отсутствует значение D3, т.е. нет служа-

щих, работающих в отделе D3, поэтому в данном поле нет и результирующих строк со значением D3,

хотя строка со значением D3 в таблице DEPT присутствует.

Сокращение (извлекает указанные строки из таблицы)

DEPTs where BUDGET > 8M

DEPT# DNAME BUDGET

D1 Marketing 10M

D2 Development 12M

Проекция (извлекает определенные столбцы из таблицы)

DEPTs over DEPT#, BUDGET

DEPT# BUDGET

D1 10M

D2 12M

D3 5M

Важная особенность — результат выполнения каждой из трех представленных операций — это

еще одна таблица, т.е. операторы сокращения, проекции и соединения порождают таблицы из та-

блиц.

Это — пример проявления алгебраического свойства замкнутости. Оно имеет очень большое зна-

чение, главным образом потому, что результатом выполнения операции является объект того же рода,

что и объект, над которым производилась операция, а именно — таблица.

Это значит, что

К результатам операций можно снова применить какую-либо операцию.

Например, можно сформировать проекцию соединения, соединение двух сокращений, сокраще-

ние проекции и т.д.

То есть, можно использовать вложенные выражения, т.е. выражения, в которых операнды пред-

ставлены выражениями, а не простыми именами таблиц.

Промежуточные результаты

Тот факт, что результатом выполнения каждой операции является таблица, не подразумевает, что

система обязательно должна полностью материализовать результат каждой отдельной операции в

виде реальной таблицы — промежуточный результат, который создается операцией соединения, воз-

можно, никогда и не будет существовать в виде полной материализованной таблицы, как таковой.

На практике, как правило, разработчики каждой системы настойчиво стремятся избежать полной

материализации промежуточных результатов в целях повышения производительности.

Если промежуточные результаты материализуются полностью, то стратегия вычисления выраже-

ния в целом называется материализованными вычислениями (materialized evaluation).

Если промежуточные результаты передаются последующей операции по частям, то этот подход

называется конвейерными вычислениями (pipelined evaluation).

Важно — операции применяются сразу ко всему множеству строк, а не к отдельной строке за один

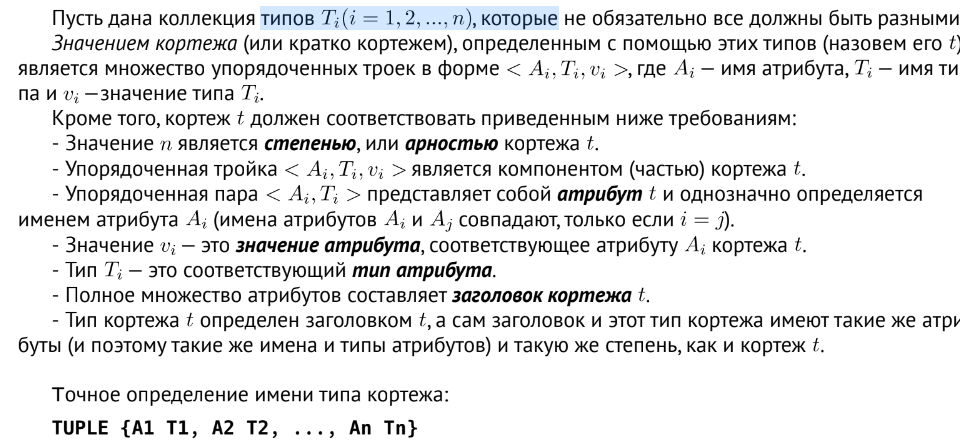
раз, т.е. операндами и результатами являются не отдельные строки, а целые таблицы, которые содер-

жат множество строк.

Почему ER-модель не является ОО-моделью?

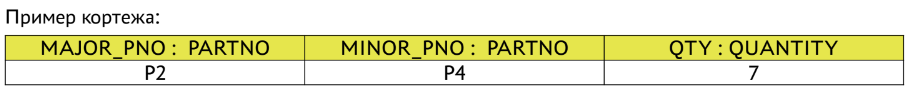
В интернете

Формальное определение кортежа.



Точное определение имени типа кортежа:

TUPLE {A1 T1, A2 T2, ..., An Tn}



Имена атрибутов: MAJOR\_PNO, MINOR\_PNO, QTY

Имена типов: PARTNO, QUANTITY

Значения: PARTNO{'P2'}, PARTNO{'P4'}, QUANTITY{7}

TUPLE {MAJOR\_PNO PARTNO, MINOR\_PNO PARTNO, QTY QUANTITY}

В неформальном изложении принято исключать имена типов из заголовка кортежа и показывать

только имена атрибутов. Поэтому показанный выше кортеж может быть неформально представлен:

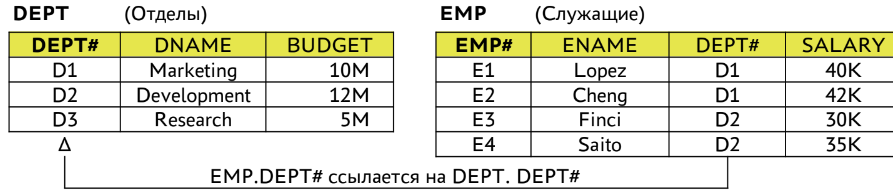


Термин «кортеж» приблизительно соответствует понятию строки, так же, как термин «отношение»

приблизительно соответствует понятию таблицы.

В реляционной модели не используется термин «поле», вместо него используется термин «атрибут»,

который в рассматриваемом контексте примерно соответствует понятию столбца таблицы.



Таблицы DEPT И ЕМР в базе данных фактически являются переменными отношения.

Соответственно, как любые переменные, имеют значения. Их значения — значения отношения,

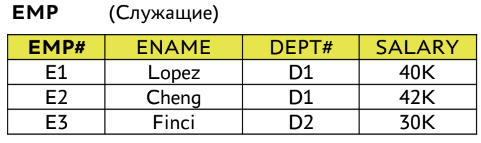
которые они принимают в разное время.

Предположим, например, что таблица ЕМР в данный момент имеет значение (значение отношения),

которое показано выше, и допустим, что мы удалили строку о сотруднике с фамилией Saito.

DELETE ЕМР WHERE ЕМР# = ЕМР# ('Е4') ;

Результат выполнения этой операции:



Концептуально это действие можно описать таким образом — старое значение отношения ЕМР

было заменено в целом совершенно другим, новым значением отношения.

Старое значение с четырьмя строками и новое значение с тремя очень похожи, но в действительно-

сти они являются разными.

Можно считать, что данная операция удаления строки — это просто альтернативный, упрощенный

способ записи для определенной операции реляционного присваивания, которая могла бы выглядеть

примерно следующим образом1

.

ЕМР := ЕМР WHERE NOT (ЕМР# = ЕМР#('Е4'));

Сначала вычисляется выражение, расположенное справа от знака присваивания, а затем его значе-

ние присваивается переменной, которая записана перед знаком присваивания.

В целом задача заключается в том, чтобы заменить «старое» значение отношения ЕМР «новым».

Таким же образом операции INSERT и UPDATE также являются просто сокращенной формой запи-

си соответствующих реляционных операций присваивания.

В реляционной теории (РТ) следует четко различать переменные отношения и сами отношения.

Для переменной отношения (relation variable) иногда используется термин relvar2

.

Термин «переменная отношения» (relvar) не является общепринятым и в документации на БД прак-

тически не встречается, хотя различать сами отношения, т.е. значения отношений, и переменные отно-

шения как таковые, с точки зрения РТ очень важно. Например, ограничения целостности и операции

обновления, применяются к переменным отношения, а не к отношениям.

1) Оператор DELETE и равносильный ему оператор присваивания записаны на языке Tutorial D

2) Различие между значениями отношения и переменными отношения фактически представляют собой особый случай раз-

личия между значениями и переменными в целом.

Ключи. Первичные, внешние. Примеры из лабораторной.

Ключи

Ключи применяются с целью связывания между собой разных таблиц в базе данных.

Первичные ключи

Первичный ключ — это атрибут или сочетание атрибутов, идентифицирующих один конкретный

экземпляр сущности.

Внешние ключи (foreign keys )

Внешний ключ создается каждый раз, когда атрибут привязывается к сущности посредством

единичной или множественной связи.

Типы

Под типом подразумевается тип данных в соответствующем поле таблицы.

Однако это также может быть и тип сущности, то есть описание ее составляющих.

Например, у сущности «книга» могут быть следующие типы:

- автор;

- название;

- дата публикации;

- издательство.

Идентификатор объекта рассматривается как ключ записи или группы записей. Такой ключ на-

зывается первичным ключом. (Primary Key)

В тех случаях, когда в качестве ключа используются несколько элементов данных (полей, агре-

гатов), их обычно указывают как соединенные символом «плюс», например НОМЕР\_РЕЙСА + ДАТА.

Такое сочетание называется сцепленным ключом.

Иногда для формирования первичного ключа необходимы три, четыре и более элементов дан-

ных. Запись о кассовом сборе, например, в приведенной ниже группе идентифицируется комбина-

цией элементов КИНОФИЛЬМ, КИНОТЕАТР, ДАТА.

КИНОФИЛЬМ КИНОТЕАТР ДАТА КАССОВЫЙ\_СБОР

Первичным ключом поэтому является КИНОФИЛЬМ+КИНОТЕАТР+ДАТА.

Первичный ключ — это такой элемент данных или такая совокупность элементов данных, ко-

торая единственным образом идентифицирует одну запись или группу записей.

Первичный ключ имеет большое значение, так как он используется для определения местопо-

ложения записи с помощью индексов или других методов адресации.

Каталог базы данных. Содержимое. Пример.

Каждая СУБД должна поддерживать функции каталога, или словаря.

Каталог обычно размещается там, где хранятся различные схемы (внешние, концептуальные, вну-

тренние) и все, что относится к отображениям («внешний-концептуальный», «концептуальный-вну-

тренний», «внешний-внешний»).

В каталоге содержится подробная информация, касающаяся различных объектов, имеющих зна-

чение для самой системы (описательная информация или метаданные):

- переменные отношения;

- индексы;

- ограничения для поддержки целостности;

- ограничения защиты;

- и пр.

Эта информация необходима для обеспечения правильной работы системы.

Например, оптимизатор использует информацию каталога об индексах и других физических

структурах хранения данных, а также прочую информацию, необходимую ему для принятия решения

о том, как выполнить тот или иной запрос пользователя.

Подсистема защиты использует информацию каталога о пользователях и установленных ограни-

чениях защиты, чтобы разрешить или запретить выполнение поступившего запроса.

Свойством реляционных систем является то, что их каталог также состоит из переменных отно-

шения, таких же как и пользовательские.

Чтобы отличать их от пользовательских их называют системными переменными отношения.

В результате пользователь может обращаться к каталогу так же, как к своим данным.

Например, в каталоге любой системы SQL обычно содержатся системные переменные отношения

TABLES и COLUMNS, назначение которых — описание известных системе таблиц, т.е. переменных отно-

шения, и столбцов этих таблиц.

Основные типы данных в БД.

Лекция №6

## Это GPT 1. Числовые типы

1.1 Целочисленные (Integer Types)

* TINYINT, SMALLINT, INT (INTEGER), BIGINT
  + Отличаются диапазоном допустимых значений и объёмом занимаемой памяти.
  + Например, в MySQL:
    - TINYINT обычно 1 байт, диапазон −128..127-128..127−128..127 (или 0..2550..2550..255 для *unsigned*).
    - SMALLINT — 2 байта.
    - INT (INTEGER) — 4 байта.
    - BIGINT — 8 байт.

1.2 Числа с плавающей точкой (Floating Point)

* REAL, FLOAT, DOUBLE (DOUBLE PRECISION)
  + Используются для хранения вещественных чисел с плавающей точкой.
  + Могут приводить к потере точности при некоторых операциях.

1.3 Десятичные (Decimal / Numeric)

* DECIMAL(precision, scale) или NUMERIC(precision, scale)
  + Применяются для точного хранения чисел с фиксированным количеством знаков после запятой (в банковских расчётах и т. п.).
  + *precision* — общее количество знаков, *scale* — количество знаков после десятичной точки.

2. Строковые (текстовые) типы

2.1 Фиксированной длины

* CHAR(n)
  + Хранит строки фиксированной длины nnn.
  + Если реальная длина строки меньше nnn, в некоторых СУБД происходит автоматическое дополнение пробелами до нужного размера.

2.2 Переменной длины

* VARCHAR(n)
  + Хранит строки переменной длины, но не более nnn.
  + Более гибок, чем CHAR, поскольку занимает ровно столько места, сколько нужно для хранения фактических данных (плюс служебная информация).

2.3 Текстовые (Text / CLOB)

* TEXT, CLOB (Character Large Object) и их разновидности (MEDIUMTEXT, LONGTEXT и т.д.)
  + Используются для хранения больших объёмов текстовых данных (несколько килобайт, мегабайт или даже гигабайт текста).

3. Дата и время

* DATE
  + Хранит дату (год, месяц, день).
* TIME
  + Хранит время (часы, минуты, секунды).
* DATETIME / TIMESTAMP
  + Хранят комбинацию даты и времени.
  + TIMESTAMP в некоторых СУБД (например, MySQL) может автоматически обновляться при изменении записи или может хранить данные в *UTC* формате для удобства.
* INTERVAL (не во всех СУБД)
  + Хранит промежутки времени (дни, часы и т.д.).

4. Булевы (логические) типы

* BOOLEAN (BOOL)
  + Хранит два значения: TRUE и FALSE.
  + В некоторых СУБД фактически представлен как TINYINT(1) (0 для FALSE, 1 для TRUE).

5. Бинарные и двоичные большие объекты

* BINARY(n), VARBINARY(n)
  + Аналоги CHAR/VARCHAR, но для двоичных (не текстовых) данных фиксированной или переменной длины.
* BLOB (Binary Large Object)
  + Аналог TEXT/CLOB, но для хранения больших объёмов двоичных данных (например, изображений, аудио, видео, зашифрованных файлов и т.д.).
  + В MySQL встречаются разновидности: TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB.

6. Специализированные / Расширенные типы

Некоторые системы управления базами данных предлагают дополнительные типы данных:

* UUID / GUID
  + Для хранения глобально уникальных идентификаторов.
  + Например, UUID (16 байт) в PostgreSQL.
* JSON / JSONB
  + Для хранения структурированных данных в формате JSON.
  + Например, PostgreSQL (JSON / JSONB), MySQL (JSON).
* XML
  + Специализированный тип для хранения XML-структур.
* GEOMETRY / SPATIAL
  + Хранение геометрических объектов (точек, линий, полигонов) в GIS-приложениях (например, PostGIS для PostgreSQL).
* ARRAY, HSTORE, RANGE (в PostgreSQL)
  + ARRAY: массивы значений.
  + HSTORE: структура «ключ—значение».
  + RANGE: диапазоны (int4range, tsrange и др.).

Итоги

1. Числовые (целые, с плавающей точкой, десятичные)
2. Строковые (фиксированная/переменная длина, большие текстовые)
3. Дата и время (DATE, TIME, DATETIME/TIMESTAMP, INTERVAL)
4. Логические (BOOLEAN)
5. Бинарные (BLOB, BINARY/VARBINARY)
6. Дополнительные (JSON, XML, массивы и т.п. в зависимости от СУБД)

Типы данных postgres

Лекция №6

Postgres

Записи varchar(n) и char(n) являются синонимами character varying(n) и character(n),

соответственно.

Записи character без указания длины соответствует character(1).

Если длина не указана для character varying, этот тип будет принимать строки любого размера.

Тип text позволяет хранить строки произвольной длины.

Тип text не в стандарте SQL, но его поддерживают многие СУБД SQL.

Значения типа character физически дополняются пробелами до n символов, хранятся и отобража-

ются в таком виде.

При сравнении двух значений типа character дополняющие пробелы считаются незначащи-

ми и игнорируются.

С правилами сортировки, где пробельные символы являются значащими, это поведение может при-

водить к неожиданным результатам:

SELECT 'a '::CHAR(2) collate "C" < E'a\n'::CHAR(2)

вернёт true, хотя в локали «C» символ пробела считается больше символа новой строки.

При приведении значения character к другому символьному типу дополняющие пробелы

отбрасываются.

Дополняющие пробелы несут смысловую нагрузку в типах character varying и text, в про-

верках по шаблонам, и в регулярных выражениях.

Какие именно символы можно сохранить в этих типах данных, зависит от того, какой набор симво-

лов был выбран при создании базы данных.

В любом случае символ с кодом 0 (NUL) сохранить нельзя, вне зависимости от выбранного

набора символов.

Для хранения короткой строки (до 126 байт) требуется дополнительный 1 байт плюс размер самой

строки, включая дополняющие пробелы для типа character. Для строк длиннее требуется не 1, а 4

дополнительных байта. Система может автоматически сжимать длинные строки, так что физический

размер на диске может быть меньше. Очень длинные текстовые строки переносятся в отдельные та-

блицы, чтобы они не замедляли работу с другими столбцами.

1. Числовые типы

1.1 Целочисленные

* SMALLINT (2 байта)  
  Диапазон: −32768..32767-32768..32767−32768..32767.
* INTEGER (INT) (4 байта)  
  Диапазон: −2147483648..2147483647-2147483648..2147483647−2147483648..2147483647.
* BIGINT (8 байт)  
  Диапазон: −9223372036854775808..9223372036854775807-9223372036854775808..9223372036854775807−9223372036854775808..9223372036854775807.

1.2 Целочисленные автоинкрементные

* SERIAL (эквивалент INTEGER + автоинкрементная последовательность).
* BIGSERIAL (эквивалент BIGINT + автоинкрементная последовательность).

Внимание: SERIAL и BIGSERIAL — это не отдельные типы как таковые, а удобные «псевдотипы» для объявления столбца со встроенной последовательностью.

1.3 Числа с плавающей точкой

* REAL (4 байта)
* DOUBLE PRECISION (8 байт)

1.4 Десятичные (точные)

* NUMERIC(precision, scale) или DECIMAL(precision, scale)
  + Для хранения чисел с фиксированной точностью.
  + Полезно, когда важна точная арифметика (например, финансовые расчёты).

1.5 Денежный тип

* MONEY
  + Специализированный тип для денежных значений.
  + Поддерживает форматированный вывод.

2. Строковые (символьные) типы

* CHAR(n)
  + Строка фиксированной длины (дополняется пробелами, если фактическая длина меньше).
* VARCHAR(n)
  + Строка переменной длины, но не более nnn.
* TEXT
  + Строка переменной длины без ограничений (фактически ограничена только доступной памятью/местом).

В PostgreSQL обычно предпочитают TEXT и VARCHAR(n) (или вовсе без указания nnn, так как PostgreSQL допускает VARCHAR без размера).

3. Дата и время

* DATE
  + Хранит календарную дату (год, месяц, день).
* TIME [ (p) ] [ WITHOUT TIME ZONE ]
  + Хранит время суток (часы, минуты, секунды). Параметр ppp задаёт точность в долях секунды.
* TIME [ (p) ] WITH TIME ZONE (не так часто используется)
* TIMESTAMP [ (p) ] [ WITHOUT TIME ZONE ]
  + Дата и время (без учёта часовых поясов).
* TIMESTAMP [ (p) ] WITH TIME ZONE (обычно сокращённо timestamptz)
  + Дата и время с учётом часового пояса. Физически хранит в UTC, а показывает с учётом локальных настроек.
* INTERVAL [ (p) ]
  + Хранит период времени (например, 1 день 2 часа 30 минут).

4. Булевый (логический) тип

* BOOLEAN
  + Принимает значения TRUE / FALSE.
  + Также понимает t/f, yes/no, 1/0 и т.д. при вводе данных.

5. Бинарные данные

* BYTEA
  + Хранение двоичных (бинарных) строк произвольной длины (например, файлы, изображения).

6. JSON-тип

* JSON
  + Хранит строку в формате JSON без парсинга на уровне хранения (проверяется только синтаксическая корректность).
* JSONB
  + Бинарное представление JSON, позволяющее быстрее выполнять поиск и индексировать данные внутри JSON.

JSONB обычно предпочтительнее для «оперативной» работы с данными, т. к. поддерживает эффективные индексы (GiN, B-Tree и т.д.).

7. UUID

* UUID (16 байт)
  + Для хранения уникальных идентификаторов формата UUID (RFC 4122).
  + Генерация может выполняться функциями uuid\_generate\_v4() (через расширение uuid-ossp) или другими методами.

8. Сетевые типы

* CIDR
  + Сетевая адресация (например, 192.168.100.0/24).
* INET
  + IP-адрес (IPv4 или IPv6) с необязательной маской подсети.
* MACADDR / MACADDR8
  + MAC-адрес (старый 6-байтовый формат или новый 8-байтовый).

9. Геометрические типы

* POINT (x, y)
* LINE (векторное задание прямой)
* LSEG (отрезок)
* BOX (прямоугольник)
* PATH (путь в виде набора точек)
* POLYGON (полигон)
* CIRCLE (окружность)

Для полноценной GIS-функциональности обычно используют расширение [PostGIS](https://postgis.net/" \t "_new), предоставляющее ещё более широкий набор геометрических и географических типов.

10. Типы для полнотекстового поиска

* TSVECTOR
  + «Лексемизированный» текст, подходит для быстрого полнотекстового поиска.
* TSQUERY
  + Специальный тип для представления поискового запроса (конъюнкции, дизъюнкции и т.п.).

11. Битовые строки

* BIT [ (n) ]
  + Строка фиксированной длины nnn бит.
* BIT VARYING [ (n) ] или VARBIT
  + Строка переменной длины, максимум nnn бит.

12. Массивы (ARRAY)

* Тип\_элемента []
  + Практически любой встроенный (или пользовательский) тип в PostgreSQL может быть представлен в виде массива.
  + Например, INT[], TEXT[], UUID[].
  + Можно хранить и одномерные, и многомерные массивы.

13. Перечисления (ENUM)

* CREATE TYPE имя AS ENUM ('значение1', 'значение2', ...)
  + Пользовательский тип, принимающий одно из оговорённых значений.
  + Удобен для полей, где набор возможных значений ограничен (например, status может быть 'draft', 'published', 'archived' и т. п.).

14. Диапазоны (Range Types)

PostgreSQL поддерживает «диапазоны» для различных базовых типов:

* int4range (для INTEGER)
* int8range (для BIGINT)
* numrange (для NUMERIC)
* tsrange (для TIMESTAMP WITHOUT TIME ZONE)
* tstzrange (для TIMESTAMP WITH TIME ZONE)
* daterange (для DATE)

Диапазон хранит «начало» и «конец» интервала (включительно или не включительно). Полезно для задач, связанных со временем (забронировать промежутки), диапазонами версий и т.д.

15. Собственные (пользовательские) типы

* Composite Types
  + Позволяют определять структуры (набор полей) в качестве пользовательских типов.
* Domains
  + Позволяют создавать логические домены на основе уже имеющихся типов, дополняя их пользовательскими ограничениями (CHECK).

Итоговый список основных типов данных в PostgreSQL

1. Числовые
   * SMALLINT, INT, BIGINT, SERIAL, BIGSERIAL, NUMERIC, REAL, DOUBLE PRECISION, MONEY
2. Символьные
   * CHAR, VARCHAR, TEXT
3. Дата и время
   * DATE, TIME [WITH TIME ZONE], TIMESTAMP [WITH TIME ZONE], INTERVAL
4. Логические
   * BOOLEAN
5. Бинарные
   * BYTEA
6. JSON
   * JSON, JSONB
7. UUID
   * UUID
8. Сетевые
   * CIDR, INET, MACADDR(/MACADDR8)
9. Геометрические
   * POINT, LINE, LSEG, BOX, PATH, POLYGON, CIRCLE
10. Полнотекстовый поиск
    * TSVECTOR, TSQUERY
11. Битовые строки
    * BIT, BIT VARYING (или VARBIT)
12. Массивы
    * [ ] для любого базового типа
13. Перечисления (ENUM)
14. Диапазоны (RANGE)
    * int4range, int8range, numrange, tsrange, tstzrange, daterange
15. Пользовательские типы
    * Composite Types, Domains

Индексирование. В-деревья. Хеширование. Пример простой функции хеширования.

При выполнении запроса к таблице пользователя часто интересуют только некоторые записи в

ней, например записи, имеющие определенное значение в некотором поле.

Индекс – это файл, помогающий движку базы данных быстро найти такие записи, не просмат-

ривая всю таблицу.

Наиболее распространенными способами реализации индексов являются:

- статическое хеширование;

- расширяемое хеширование;

- B-деревья.

Эффективность выполнения некоторых запросов может значительно повысить подходящая ор-

ганизация таблиц. Представим телефонный справочник – по сути, большая таблица, записи кото-

рой содержат имена, адреса и номера телефонов абонентов.

Фамилия Имя Отчество Адрес Тел. номер

Эта таблица отсортирована сначала по фамилиям, а затем по именам.

Предположим, что нам нужно узнать номер телефона конкретного человека.

Существенно ускорить поиск помогает тот факт, что записи отсортированы по имени.

Например, можно выполнить бинарный поиск, который в худшем случае потребует просмот-

реть log2N записей, где N – общее число записей в справочнике. Это очень быстро.

Например, если N = 1 000 000, то log2N < 20, то есть чтобы найти нужного человека в справоч-

нике, содержащем номера миллиона человек, никогда не придется просматривать больше 20 запи-

сей.

Телефонный справочник отлично приспособлен для поиска абонента по имени, но не подходит

для быстрого поиска, например по номеру телефона или по адресу.

Единственный способ получить эту информацию из телефонной книги – просмотреть каждую

запись в ней, что потребует просмотреть в среднем N/2 записей.

Для эффективного поиска абонентов по номеру телефона нужен справочник, отсортированный

по номерам телефонов (такие справочники еще называют «обратными телефонными справочни-

ками»). Однако такой справочник удобен, только если известен номер телефона. Если необходимо

найти в таком справочнике номер телефона конкретного абонента, то снова придется просмотреть

каждую запись.

Этот пример наглядно иллюстрирует важное ограничение организации таблиц — таблицу мож-

но организовать (упорядочить) только каким-то одним способом. Если необходимо, чтобы поиск

выполнялся быстро и по номеру телефона и по имени абонента, потребуются две отдельные копии

телефонной книги, каждая со своей организацией. А если понадобится возможность быстро нахо-

дить номер телефона по известному адресу, потребуется третий экземпляр телефонного справоч-

ника, отсортированный по адресу. Это справедливо и в отношении таблиц в базе данных.

Чтобы иметь возможность эффективно находить в таблице записи с определенным

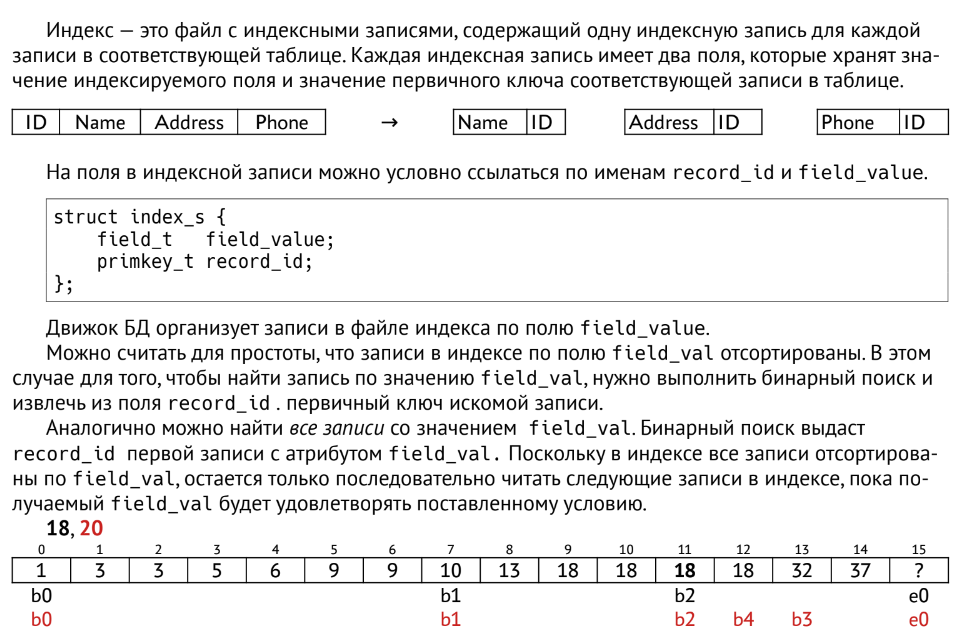
значением некоторого поля, нужна версия таблицы, организованная по этому полю.

Механизмы баз данных это удовлетворяют эту потребность, поддерживая индексы.

Таблица может иметь один или несколько индексов, каждый из которых определен для отдель-

ного поля. Каждый индекс действует подобно версии таблицы, организованной по соответствую-

щему полю.



Насколько эффективно такое применение индексов?

В отсутствие индексов лучшее, что можно предпринять при обработке любого запроса, – вы-

полнить последовательный поиск в таблице.

Если атрибут в таблице равномерно распределен и имеет вид «A = константа», справедливо

правило — полезность индекса для поля A в таблице пропорциональна количеству уникальных зна-

чений в этом поле.

Согласно этому правилу, индекс наиболее полезен, когда индексируемое поле является первич-

ным ключом таблицы, потому что каждая запись имеет свое уникальное значение ключа.

И наоборот, индекс будет бесполезен, если число уникальных значений в поле A меньше коли-

чества записей в блоке чтения.

Но, есть нюансы (прямой доступ против последовательного).

Статические хеш-индексы

Статическое хеширование – это, пожалуй, самый простой способ реализации индексов.

Это не самая эффективная стратегия, но достаточно простая и понятная для реализации.

Статическое хеширование

Статический хеш-индекс использует фиксированное число N ячеек, пронумерованных от 0 до

N−1. Индекс также использует хеш-функцию, отображающую значения в ячейки. По результатам

хеширования поля field\_val каждая индексная запись помещается в свою ячейку.

Статический хеш-индекс действует следующим образом:

- чтобы сохранить индексную запись, ее нужно поместить в ячейку, вычисленную хеш-функци-

ей;

- чтобы найти индексную запись, нужно вычислить хеш-функцию ключа поиска и просмотреть

соответствующую ячейку;

- чтобы удалить индексную запись, нужно сначала найти ее (как указано выше), а затем удалить

из ячейки.

Стоимость поиска с помощью хеш-индекса обратно пропорциональна количеству ячеек.

Если индекс содержит B блоков (единиц чтения) и N ячеек, то на каждую ячейку будет прихо-

диться около B/N блоков, поэтому для поиска в ячейке потребуется обратиться к B/N блоков.

N = 17, B = 8 ячейка полностью в одном блоке -> гарантированно одно чтение

N = 17, B = 20 ячейка располагается в двух блоках -> может быть два чтения

Расширяемое хеширование

Стоимость поиска при использовании индексов на основе статического хеширования обратно

пропорциональна количеству ячеек – чем больше ячеек используется, тем меньше блоков в каж-

дой из них.

Наиболее оптимально, когда каждую ячейку приходится ровно один блок.

Если бы размер индекса никогда не изменялся, то рассчитать это идеальное количество ячеек

легко. Но на практике индексы растут по мере добавления новых записей в базу данных.

Если исходить из текущего размера индекса, то впоследствии, при его увеличении, каждая

ячейка будет содержать несколько блоков индекса.

Если выбрать большее количество ячеек, ориентируясь на потребности в будущем, то пустые и

почти пустые в данный момент ячейки будут напрасно расходовать значительный объем дискового

пространства до тех пор, пока индекс не вырастет и их не заполнит.

Эту проблему решает стратегия, известная как расширяемое хеширование.

Суть этой стратегии заключается в использовании достаточно большого количества ячеек, что-

бы гарантировать, что каждая ячейка никогда не будет содержаться более чем в одном блоке, или,

что эквивалентно, в одном блоке будет содержаться несколько ячеек.

Поскольку в одном блоке могут располагаться несколько ячеек, это позволяет большому коли-

честву ячеек совместно использовать меньшее количество блоков и тем самым не допустить

напрасного расходования дискового пространства.

Совместное использование блоков ячейками обеспечивается с помощью двух файлов – файла

ячеек и каталога ячеек.

Файл ячеек содержит блоки индекса, а каталог ячеек отображает ячейки в блоки.

Каталог можно рассматривать как массив целых чисел, по одному для каждой ячейки. Пусть это

массив Dir. Тогда если индексная запись хешируется в ячейку b, то запись будет сохранена в блоке

Dir[b] файла ячеек.

Допустим, что в блоке размещаются 3 ячейки, всего ячеек 8, хеш-функция h(x)=x mod 8, в ин-

дексе семь записей с идентификаторами 1, 2, 4, 5, 7, 8, 12.

Каталог ячеек: [0 1 2 1 0 1 2 1]

Файл ячеек: [(4, r4) (8, r8) (12, r12)] [(1, r1) (5, r5) (7, r7)] [(2, r2)]

Надо заметить, что этот подход не работает, когда имеется слишком много записей с одина-

ковым значением field\_val. Поскольку эти записи всегда будут хешироваться в одну и ту же

ячейку, стратегия хеширования никак не сможет распределить их по нескольким ячейкам. В этом

случае ячейка будет занимать столько блоков, сколько потребуется для хранения этих записей.

Индексы на основе B-дерева

Предыдущие две стратегии индексирования были основаны на хешировании.

Индексы на основе B-дерева — подход на основе сортировки. Основная идея заключается в

сортировке индексных записей по значениям field\_val.

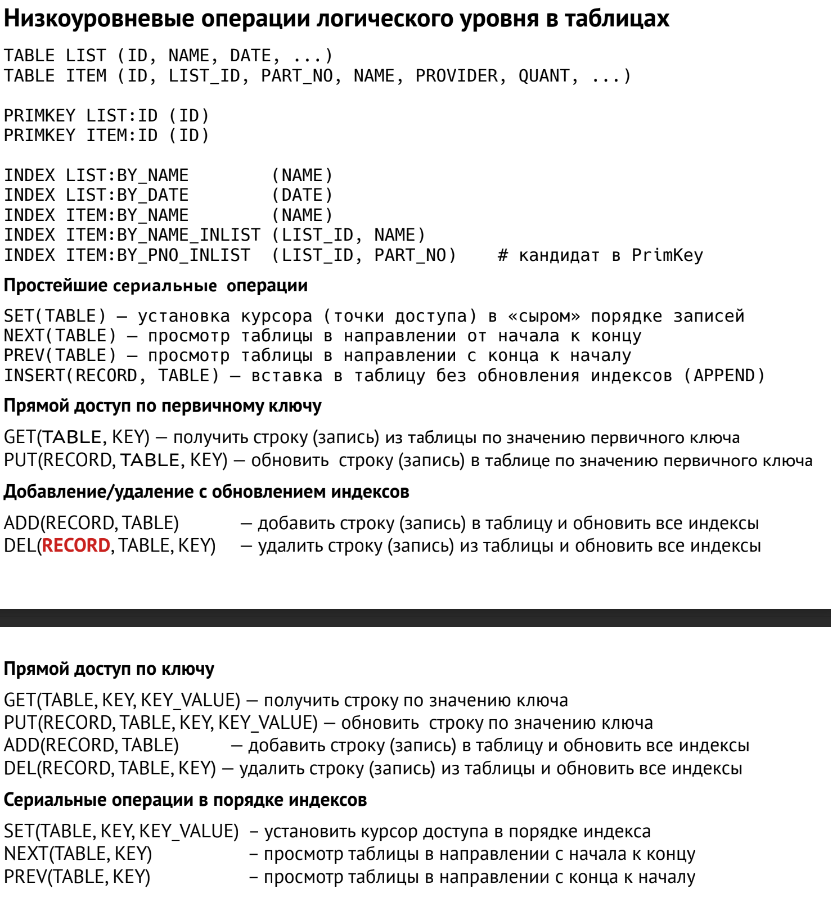
Индексный файл – это последовательность индексных записей с полями field\_val и

record\_id. При работе с индексным файлом для нас важно иметь возможность как можно бы-

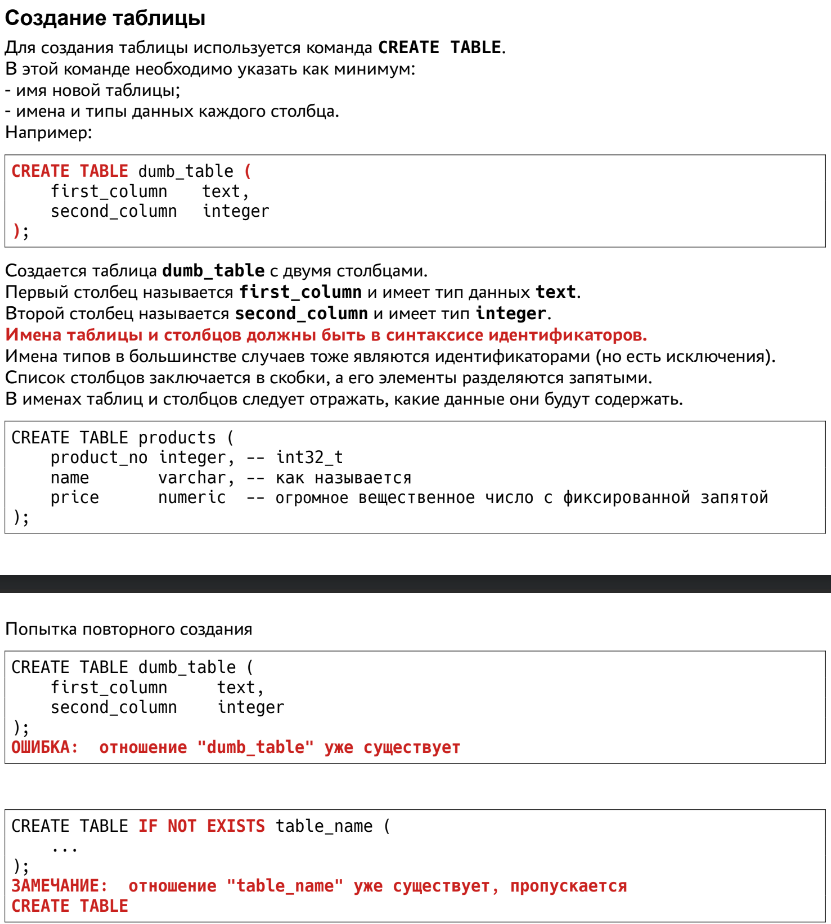
стрее находить идентификаторы записей (record\_id) по значениям индексированного поля

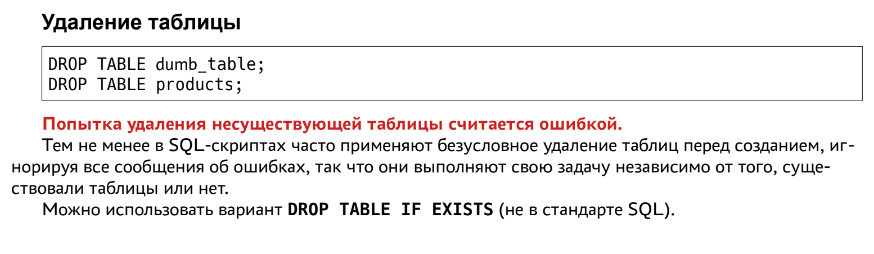
(field\_val .

Кроме поиска поддерживается вставка и удаление.



Создание таблиц в SQL, удаление.





1. Компоненты БД:

- пользователи;

- прикладные программы;

- базы данных;

- система управления базами данных.



Пользователи

I) Пользователи делятся на 4 группы:

1) администраторы;

2) разработчики баз данных;

3) прикладные программисты;

4) простые (наивные) пользователи.

Администраторы

- администраторы данных;

- администраторы баз данных.

Администраторы данных — отвечают за управление данными, включая планирование БД, раз-

работку и сопровождение стандартов, бизнес правил (описывают основные характеристики дан-

ных с точки зрения организации) и деловых процедур, а также концептуальное и логическое проектирование БД;

Администраторы баз данных — отвечают за физическую реализацию БД, включая физическое проектирование, обеспечение безопасности и целостности данных, а также обеспечение максимальной производительности приложений и пользователей (более технический характер по сравнению с администратором данных).

Разработчики баз данных

- разработчики логической базы данных;

- разработчики физической базы данных.

разработчики логической базы данных занимаются идентификацией данных, связей между данными и устанавливают ограничения, накладываемые на хранимые данные — (ответ на вопрос ЧТО?);

разработчики физической базы данных по готовой логической модели создают физическую реализацию (формирование таблиц, выбор структур хранения, методов доступа, мер защиты) — (ответ на вопрос КАК?).

Прикладные программисты

Создают приложения предоставляющие пользователям необходимые функциональные возможности (действия над базой данных).

Пользователи (клиенты БД)

- наивные пользователи — осуществляют доступ к БД через прикладные программы;

- опытные пользователи — могут осуществлять доступ к БД с использованием языков запросов или создавать собственные прикладные программы.

Прикладные программы

- пользователи;

- прикладные программы;

- базы данных;

- система управления базами данных.

Прикладные программы обеспечивают удобный для пользователей доступ к БД.

Реализуются с использованием языков программирования 3-го (процедурные языки — C,COBOL, Fortran, Ada, Pascal) или 4-го поколения (SQL, QBE). 4-е поколение (“4GL”) непроцедурные ‒ языки, возможно генерирование прикладного приложения по параметрам, заданных пользователем.

Языки программирования делятся на:

- языки представления информации (языки запросов или генераторы форм и отчетов);

- специализированные языки (электронных таблиц и БД);

- генераторы приложений для определения, вставки, обновления или извлечения сведений из БД;

- языки очень высокого уровня для генерации кода приложений.

База данных

- пользователи;

- прикладные программы;

- базы данных;

- система управления базами данных.

База данных (БД)— совокупность логически связанных данных, хранящихся в компьютеризованной системе и отражающих некоторую предметную область человеческой деятельности.

БД — единое, большое хранилище данных (набор интегрированных записей с самоописанием), содержащее данные с минимальной долей избыточности, к которым может обращаться большое число пользователей.

Описание данных называется системным каталогом или словарем данных, а сами элементы описания — метаданные (данные о данных) обеспечивают независимость между программами и данными.  
  
Система управления базами данных

- пользователи;

- прикладные программы;

- базы данных;

- система управления базами данных.

Система управления базами данных (СУБД) — программное обеспечение, с помощью которого пользователи могут определять, создавать и поддерживать БД, а также осуществлять к ней контролируемый доступ.

Главные преимущества СУБД — преодоление ограничений файловых систем (в основном из-за

обеспечения централизованного управления данными).

2. Агрегатное выражение и агрегатные функции

Агрегатное выражение представляет собой применение агрегатной функции к строкам, которые выбраны запросом.

Агрегатная функция сводит множество входных значений к одному выходному, например, к сумме или среднему.

Обычно строки данных передаются агрегатной функции в неопределённом порядке и во многих

случаях это не имеет значения, например функция min выдаёт один и тот же результат независимо от порядка поступающих данных.

Есть некоторые агрегатные функции (array\_agg, string\_agg), которые выдают результаты, зависящие от порядка данных. Для таких агрегатных функций в список аргументов можно добавить предложение ORDER BY, чтобы задать нужный порядок.

3. Администратор базы данных и администратор данных

Администраторы данных — отвечают за управление данными, включая планирование БД, разработку и сопровождение стандартов, бизнес правил (описывают основные характеристики данных с точки зрения организации) и деловых процедур, а также концептуальное и логическое проектирование БД;

Администраторы баз данных — отвечают за физическую реализацию БД, включая физическое проектирование, обеспечение безопасности и целостности данных, а также обеспечение максимальной производительности приложений и пользователей (более технический характер по сравнению с администратором данных).

4. Применение правил сортировки SQL. Примеры.

Предложение COLLATE переопределяет правило сортировки выражения. Оно добавляется после выражения:

expr COLLATE collation

где collation — идентификатор правила, возможно дополненный именем схемы.

Предложение COLLATE «связывает» выражение сильнее, чем операторы, поэтому при необходимости следует использовать скобки.

Если правило сортировки явно не определено, система либо выбирает в зависимости от столбцов, которые используются в выражении, либо, если в выражении столбцов нет, переключается на установленное для базы данных правило сортировки по умолчанию.

Предложение COLLATE имеет два распространённых применения:

1) переопределение порядка сортировки в предложении ORDER BY, например:

SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";

2) переопределение правил сортировки при вызове функций или операторов, возвращающих зависимые от локали результаты, например:

SELECT \* FROM tbl WHERE a > 'foo' COLLATE "C";

В последнем случае предложение COLLATE добавлено к аргументу оператора, на действие которого нужно повлиять. Не имеет значения, к какому именно аргументу оператора или функции добавляется

COLLATE, поскольку правило сортировки, применяемое к оператору или функции, выбирается при анализе всех аргументов, а явное предложение COLLATE переопределяет правила сортировки.

SELECT \* FROM tbl WHERE a COLLATE "C" > 'foo';

Но будет ошибкой:

SELECT \* FROM tbl WHERE (a > 'foo') COLLATE "C";

поскольку правило сортировки нельзя применить к результату оператора >, который имеет несравниваемый тип данных boolean.

Локаль (возможно относится к сортировке)

Локаль — это сочетание языковых и культурных аспектов. Они включают в себя:

- язык сообщений;

- различные наборы символов;

- лексикографические соглашения;

- форматирование даты и времени;

- форматирование чисел;

- и прочее.

В локали существуют различные категории информации, которые программа может использовать –

они описаны как макросы.

LC\_ALL — локаль целиком

LC\_COLLATE — сортировку строк

LC\_CTYPE — классы символов

LC\_MESSAGES — локализированные сообщения на родном языке

LC\_MONETARY — форматирование значений денежных единиц

LC\_NUMERIC — форматирование не денежных числовых значений

LC\_TIME — форматирование значений дат и времени

LC\_COLLATE — эта категория определяет правила сравнения, используемые при сортировке и регулярных выражениях, включая равенство классов символов и сравнение многосимвольных элементов.

Эта категория локали изменяет поведение функций, которые используются для сравнения строк с

учётом местного алфавита. Например, немецкая ẞß эсцет (sharp s) рассматривается как «ss».

strcoll("Das Große Eszett", "Das Grosse Eszett"); //

LC\_CTYPE — эта категория определяет интерпретацию последовательности байт в символы (напри-

мер, одиночный или многобайтовый символ), классификацию символов (например, буквенный или

цифровой) и поведение классов символов.

LC\_MONETARY — эта категория определяет форматирования, используемое для денежных значений.

Она изменяет информацию, возвращаемую функцией, которая описывает способ отображения числа,

например, необходимо ли использовать в качестве десятичного разделителя точку или запятую.

LC\_MESSAGES — эта категория изменяет язык отображаемых сообщений и указывает, как должны

выглядеть положительный и отрицательный ответы.

LC\_NUMERIC — эта категория определяет правила форматирования, используемые для не денежных

значений, например, разделительный символ тысяч и дробной части (точка в англоязычных странах, и

запятая во многих других).

LC\_TIME — эта категория управляет форматированием значений даты и времени. Например,

большая часть Европы использует 24-часовой формат, тогда как в США используют 12-часовой.

LC\_ALL — всё вышеперечисленное.

Список поддерживаемых категорий можно получить с помощью утилиты оболочки locale

$ locale

LANG=ru\_RU.utf8

LC\_CTYPE="ru\_RU.utf8"

LC\_NUMERIC="ru\_RU.utf8"

LC\_TIME="ru\_RU.utf8"

LC\_COLLATE="ru\_RU.utf8"

LC\_MONETARY="ru\_RU.utf8"

LC\_MESSAGES="ru\_RU.utf8"

LC\_PAPER="ru\_RU.utf8"

LC\_NAME="ru\_RU.utf8"

LC\_ADDRESS="ru\_RU.utf8"

LC\_TELEPHONE="ru\_RU.utf8"

LC\_MEASUREMENT="ru\_RU.utf8"

LC\_IDENTIFICATION="ru\_RU.utf8"

LC\_ALL=  
  
chat gpt (примеры сортировок)  
  
  
**1. Сортировка по ASCII-порядку («C» локаль)**

Допустим, по умолчанию в базе данных установлена локаль ru\_RU.UTF-8, но нам в одном запросе нужно отсортировать столбец a по «С»-локали (классический ASCII-порядок). В таком случае можно сделать так:

SELECT a, b, c

FROM tbl

ORDER BY a COLLATE "C";

**2. Сортировка по разным локалям для разных столбцов**

Предположим, у нас есть таблица **people** со столбцами first\_name, last\_name и nickname. Нужно отсортировать:

1. Сначала по фамилии (last\_name) с русской локалью ru\_RU.UTF-8,

2. Затем по имени (first\_name) в «C»-локали,

3. Наконец, по псевдониму (nickname) – по умолчанию (пусть он тоже будет ru\_RU.UTF-8 или любая дефолтная локаль в базе).

Тогда запрос может выглядеть так:

SELECT first\_name, last\_name, nickname

FROM people

ORDER BY

last\_name COLLATE "ru\_RU.UTF-8",

first\_name COLLATE "C",

nickname;  
  
  
  
**3. Сортировка с учётом регистра: комбинированные выражения**

Предположим, что мы хотим сначала привести строки в верхний регистр, а потом сортировать уже результат (и переопределять при этом колляцию). Можно написать примерно так:

SELECT id, UPPER(some\_text COLLATE "C") AS upper\_text

FROM my\_table

ORDER BY UPPER(some\_text COLLATE "C");

**4. Сортировка конкатенированных строк**

Иногда приходится сортировать по результату объединения строк. Например:

SELECT id,

last\_name || ', ' || first\_name AS full\_name

FROM people

ORDER BY (last\_name || ', ' || first\_name) COLLATE "C";

Тут мы явно указываем, что при сортировке конкатенированной строки last\_name || ', ' || first\_name использовать «C»-локаль.

При необходимости можно прописать колляцию отдельно для каждого из столбцов при конкатенации, если они, к примеру, сами имеют разные локали. Однако обычно достаточно указать одну колляцию для конечного результата.

5. Концептуальная схема. Что это? Кто создает?

Концептуальная схема представляет собой единое интегрированное определение данных в рамках предприятия, которое не привязано к какому-либо отдельному применению данных и не зависит от того, как данные физически хранятся или к ним осуществляется доступ.

Целью этой концептуальной схемы является предоставление последовательного определения значений и взаимосвязей данных, которые можно использовать для интеграции, совместного использования и управления целостностью данных.

Концептуальная схема описывает все элементы данных и отношения между ними вместе с ограничениями целостности. Для каждой базы данных существует только одна концептуальная схема.

Отчет ANSI/SPARC был задуман как основа для компьютерных систем, способных обмениваться информацией. Все поставщики баз данных приняли терминологию трех схем, но реализовали ее несовместимыми способами. В течение следующих двадцати лет различные группы пытались определить стандарты для концептуальной схемы и ее сопоставления с базами данных и языками программирования. К сожалению, ни у одного из поставщиков не было сильного стимула делать свои форматы совместимыми с форматами конкурентов. Было

подготовлено несколько отчетов, но не стандартов.

По мере развития практики администрирования данных и появления новых графических методов термин «схема» уступил место термину «модель».

Концептуальная модель представляет собой представление данных, которое согласовывается между конечными пользователями и администраторами баз данных, охватывающее те объекты, о которых важно хранить данные, значение данных и взаимосвязь данных друг с другом.

6. Вложенные SELECT. Примеры.

GPT:   
Используйте вложенные SELECT, когда для решения задачи нужно динамически получить промежуточные данные внутри одного SQL-запроса, без необходимости делать несколько запросов и собирать результаты «на стороне» (в приложении).

1. Самый распространённый случай – когда мы используем другой запрос для фильтрации записей. Например, выбираем всех сотрудников, у которых зарплата выше средней зарплаты по отделу 10:

SELECT empno, ename, sal, deptno

FROM emp

WHERE deptno = 10

AND sal > (SELECT AVG(sal)

FROM emp

WHERE deptno = 10);

1. **Вложенный SELECT с оператором IN**

Иногда нужно проверить, что значение столбца входит в некий набор, возвращаемый другим запросом. Например, получим список сотрудников, которые работают в тех же отделах, что и KING:

SELECT empno, ename, deptno

FROM emp

WHERE deptno IN (SELECT deptno

FROM emp

WHERE ename = 'KING');

1. **Подзапрос, возвращающий несколько полей (вместо таблицы в FROM)**

Подзапрос может выступать в роли виртуальной таблицы в блоке FROM. Например, выберем всех сотрудников и среднюю зарплату по соответствующему департаменту, присоединив две таблицы: саму таблицу сотрудников emp и «виртуальную таблицу» с агрегатами:

SELECT e.empno,

e.ename,

e.sal,

e.deptno,

avg\_dept\_sal

FROM emp e

JOIN (

SELECT deptno, AVG(sal) AS avg\_dept\_sal

FROM emp

GROUP BY deptno

) d ON e.deptno = d.deptno

ORDER BY e.deptno, e.sal DESC;

7. Логическая схема. Что это? Кто создает?

**Логическая схема** (Logical schema) — это абстрактное представление структуры базы данных, показывающее, **какие** таблицы (или сущности) и **какие** отношения (связи) между ними существуют, **какие** поля (атрибуты) входят в каждую таблицу, и **какими** типами данных они обладают. При этом не учитываются особенности конкретной СУБД или физического хранения данных.

**Кто создаёт логическую схему?**

1. Как правило, логическую схему **разрабатывают** специалисты, занимающиеся проектированием БД:

• **Архитекторы баз данных** (Database Architects),

• **Системные аналитики**,

• **Разработчики**, отвечающие за логику приложения и структуру данных.

2. На этапе проектирования БД обычно выделяют три уровня моделей:

1. **Концептуальная модель** — какие сущности и связи (самая абстрактная модель, например ER-диаграмма).

2. **Логическая схема** (Logical model) — «переосмысление» концептуальной модели с учётом конкретных типов данных, ключей, отношений «один-ко-многим» и т.д.

3. **Физическая схема** — реализация в виде конкретных таблиц, индексирования, партиций, с учётом специфики конкретной СУБД (PostgreSQL, Oracle, MySQL и т.п.).

Таким образом, **логическая схема** — это более детализированный уровень проектирования, чем «чисто концептуальная» модель, но ещё **не** учитывает технические аспекты хранения (индексы, партиционирование и прочие детали физической реализации).

8. Скалярный подзапрос SQL.

Скалярный подзапрос — это обычный запрос SELECT в скобках, который возвращает ровно одну строку и один столбец.

Результат такого запроса SELECT используется в окружающем его выражении.

В качестве скалярного подзапроса нельзя использовать запросы, возвращающие более одной строки или столбца.

Если в результате выполнения подзапрос не вернёт никаких строк, скалярный результат считается равным NULL.

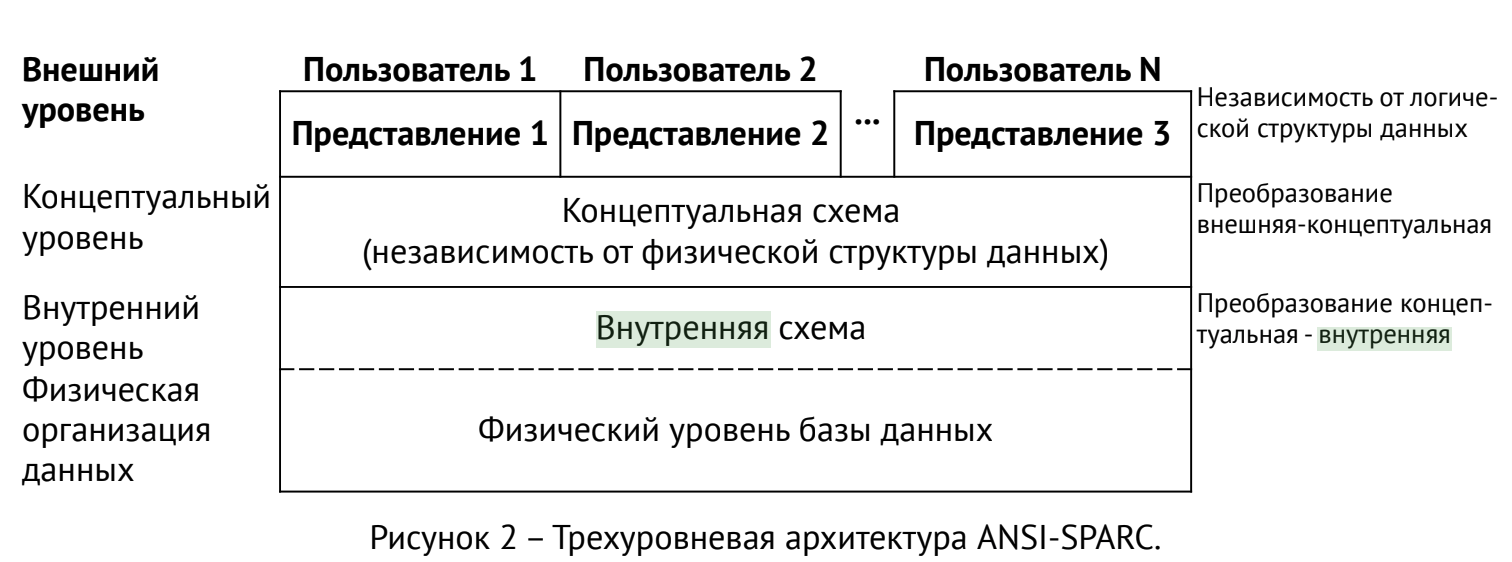
В подзапросе можно ссылаться на переменные из окружающего запроса — в процессе одного вычисления подзапроса они будут считаться константами.

Например, следующий запрос находит самый населённый город в каждом штате:

SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name) FROM states;

9. - (внутренняя схема) или представление компьютера.

Внутренняя схема на самом нижнем уровне содержит определения хранимых записей, методов представления, полей данных и индексов. Для каждой базы данных существует только одна внутренняя схема.

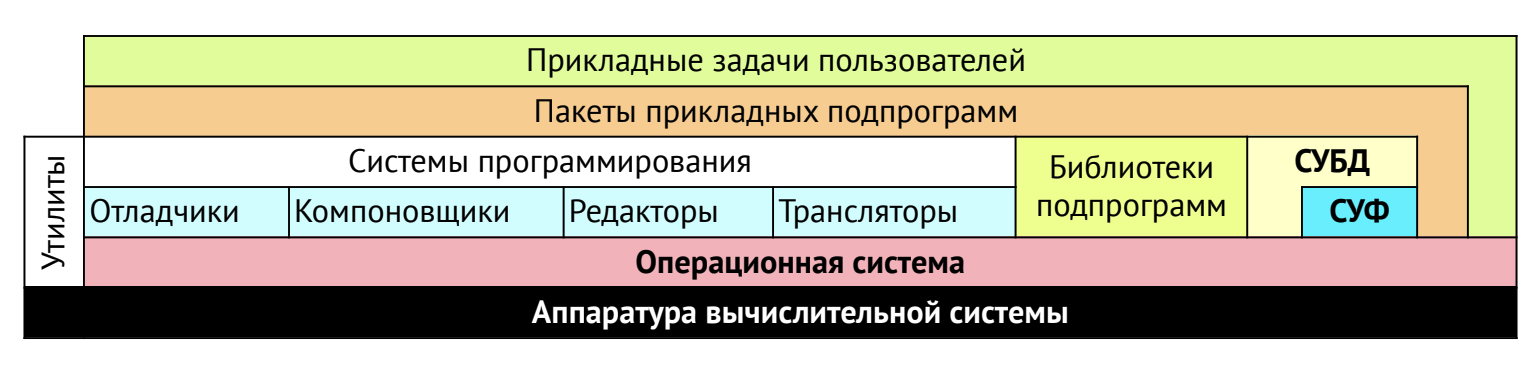


Внутренний уровень

Внутренний уровень — это физическое представление базы данных в компьютере. Уровень описывает, как информация хранится в БД, в частности, описывает физическую реализацию. Уровень предназначен для достижения оптимальной производительности и обеспечения экономного распределения дискового пространства.

На этом уровне содержится описание структур данных и описание организации файлов базы данных, а также осуществляется взаимодействие СУБД с методами доступа операционной системы для работы с данными.

Уровень содержит информацию о распределении дискового пространства, описание подробностей хранимых записей (реальная длина — байты), сведения о размещении записей, сжатии и шифровании данных.



Внутренняя схема описывает определения хранимых записей, методов представления, описания полей данных, индексов и т.п.

10. Ограничение для таблиц SQL  
  
Ограничения и особенности генерируемых выражений и таблиц, их содержащих:

1) в генерируемом выражении могут использоваться только постоянные функции и не могут фигу-

рировать подзапросы или ссылки на какие-либо значения, не относящиеся к данной строке;

2) генерируемое выражение не может обращаться к другому генерируемому столбцу;

3) генерируемое выражение не может обращаться к системным столбцам, кроме tableoid;

4) для генерируемого столбца нельзя задать значение по умолчанию;

5) генерируемый столбец не может быть частью ключа секционирования.

6) есть ограничения по наследованию.

7) права доступа к генерируемым столбцам существуют отдельно от прав для столбцов, из которых

он вычисляется, поэтому их можно организовать так, чтобы определённый пользователь мог прочитать

генерируемый столбец, но не мог читать оригиналы.

Секционирование таблица разделяется на несколько частей меньшего размера: ‒

- горизонтальное секционирование (части таблицы LOGS содержат строки по месяцам или фикс.)

- вертикальное секционирование (части таблицы NEWS содержат разные ее столбцы).

Ограничения

- ограничение DEFAULT;

- ограничения-проверки (CHECK);

- ограничения NOT NULL;

- ограничения уникальности;

- первичные ключи;

- внешние ключи;

- ограничения-исключения.

Ограничения-проверки

Ограничение-проверка — наиболее общий тип ограничений. В его определении можно указать, что

значение данного столбца должно удовлетворять логическому выражению (проверке истинности).

Например, цену товара можно ограничить положительными значениями:

CREATE TABLE products (

product\_no integer,

name varchar,

price numeric CHECK (price > 0) -- логическое выражение

);

Ограничение определяется после объявления типа данных, аналогично тому, как определяется значение по умолчанию.

Значения по умолчанию и ограничения могут указываться в любом порядке.

Ограничение-проверка состоит из ключевого слова CHECK, за которым идёт выражение в скобках.

Выражение должно включать столбец, для которого задаётся ограничение, иначе оно не имеет большого смысла.

Ограничению можно присвоить отдельное имя — это улучшит сообщения об ошибках и позволит

ссылаться на это ограничение, если придется его изменять:

CREATE TABLE products (

product\_no integer,

name varchar,

price numeric CONSTRAINT positive\_price CHECK (price > 0)

);

Синтаксис:

[ CONSTRAINT constraint\_name ] constraint\_definition

Ограничение-проверка может также ссылаться на несколько столбцов. Например, чтобы гарантировать, что цена со скидкой будет всегда меньше обычной:

CREATE TABLE products (

product\_no integer,

name varchar,

price numeric CHECK (price > 0), -- ограничение столбца

discounted\_price numeric CHECK (discounted\_price > 0), -- ограничение столбца

CHECK (price > discounted\_price) -- ограничение таблицы

);

Первые два ограничения определяются «постфиксно».

Третье представлено отдельным элементом в списке элементов определения таблицы (огр. табл.).

Определения столбцов и определения ограничений можно переставлять в произвольном порядке.

Первые два ограничения — это ограничения столбцов.

Третье ограничение — ограничение таблицы, поскольку записано отдельно от определений столб-

цов.

Ограничения столбцов также можно записать в виде ограничений таблицы, тогда как обратное не

всегда возможно, так как подразумевается, что ограничение столбца ссылается только на связанный

столбец. Хотя PostgreSQL этого не требует, но для совместимости с другими СУБД лучше следовать

это правилу.

Ранее приведённый пример можно переписать и так:

CREATE TABLE products (

product\_no integer,

name varchar(120),

price numeric,

CHECK (price > 0),

discounted\_price numeric,

CHECK (discounted\_price > 0),

CHECK (price > discounted\_price)

);

Ограничениям таблицы можно присваивать имена так же, как и ограничениям столбцов:

CREATE TABLE products (

product\_no integer,

name varchar(120),

price numeric,

CHECK (price > 0),

discounted\_price numeric,

CHECK (discounted\_price > 0),

CONSTRAINT valid\_discount CHECK (price > discounted\_price)

);

Ограничение-проверка удовлетворяется, если выражение принимает значение TRUE или NULL.

Результатом многих выражений с операндами NULL будет значение NULL, поэтому такие ограничения не будут препятствовать записи NULL в связанные столбцы.

Чтобы гарантировать, что столбец не содержит значения NULL, следует использовать ограничение

NOT NULL.

Важное замечание

В Postgres предполагается, что условия ограничений CHECK являются постоянными, то есть при

одинаковых данных в строке они всегда будут выдавать одинаковый результат.

Поэтому postgres поддерживает ограничения CHECK только для тех данных, которые относятся к

только создаваемой или изменяемой строке.

CHECK, нарушающий это правило, в некоторых простых случаях может работать, однако, в общем

случае может случиться, что база данных придёт в состояние, когда условие ограничения окажется

ложным. Например, если изменились другие строки, участвующие в его вычислении.

В результате восстановление выгруженных данных может оказаться невозможным.

Во время восстановления возможен сбой, даже если полное состояние базы данных согласуется с

условием ограничения, по причине того, что строки загружаются не в том порядке, в котором это усло-

вие будет соблюдаться.

Поэтому для определения ограничений, затрагивающих другие строки и другие таблицы, следует

использовать ограничения UNIQUE, EXCLUDE или FOREIGN KEY, либо реализовать проверки в триггере.

Триггеры из дампа восстанавливаются уже после восстановления данных, поэтому вышеупомянутого

сбоя не будет.

Пользовательские функции в ограничениях

Может случиться так, что в выражении CHECK используется пользовательская функция, поведе-

ние которой впоследствии будет изменено.

Postgres не контролирует этого, и если строки в таблице перестанут удовлетворять ограничению

CHECK, это останется незамеченным. В итоге при попытке загрузить выгруженные позже данные мо-

гут возникнуть проблемы. Поэтому подобные изменения рекомендуется осуществлять следующим об-

разом:

1) удалить ограничение (используя ALTER TABLE);

2) изменить определение функции;

3) пересоздать ограничение той же командой, которая при этом перепроверит все строки таблицы.

Ограничения NOT NULL

Ограничение NOT NULL просто указывает, что столбцу нельзя присваивать значение NULL:

CREATE TABLE products (

product\_no integer NOT NULL,

name varchar(120) NOT NULL,

price numeric

);

Ограничение NOT NULL всегда записывается как ограничение столбца и функционально эквива-

лентно ограничению:

CHECK (column\_name IS NOT NULL)

Однако, если check-ограничению можно назначить имя, то ограничению NOT NULL имя назначить

нельзя.

Для столбца можно определить больше одного ограничения, для чего их нужно указать одно за

другим:

CREATE TABLE products (

product\_no integer NOT NULL,

name varchar(120) NOT NULL,

price numeric NOT NULL CHECK (price > 0)

);

Порядок указания ограничений не имеет значения — порядок проверки ограничений не зависит

от порядка из указания.

Для ограничения NOT NULL есть и обратное ограничение — NULL.

Суть его состоит в указании, что столбец может иметь значение NULL (поведение по умолчанию).

Ограничение NULL отсутствует в стандарте SQL и было добавлено в Postgres только для совмести-

мости с некоторыми другими СУБД.

Его можно использовать для быстрого переключения ограничения NULL/NOT NULL в скрипте.

CREATE TABLE products (

product\_no integer NULL,

name varchar(120) NULL,

price numeric NULL

);

При необходимости в любое из них можно вставить ключевое слово NOT, если потребуется.

Тем не менее, при проектировании баз данных чаще всего большинство столбцов должны быть по-

мечены как NOT NULL.

Ограничения уникальности

Ограничения уникальности гарантируют, что данные в определённом столбце или группе столбцов

уникальны среди всех строк таблицы. Ограничение в виде ограничения столбца:

CREATE TABLE products (

product\_no integer UNIQUE, -- уникальность столбца

name varchar(120),

price numeric

);

Ограничение в виде ограничения таблицы:

CREATE TABLE products (

product\_no integer,

name varchar(120),

price numeric,

UNIQUE (product\_no) -- уникальность столбца

);

Чтобы определить ограничение уникальности для группы столбцов, его следует записать в виде

ограничения таблицы, при этом имена столбцов перечисляются через запятую:

CREATE TABLE example (

a integer,

b integer,

c integer,

UNIQUE (a, c) -- уникальность группы столбцов

);

Ограничение уникальности для группы столбцов указывает, что сочетание значений перечислен-

ных столбцов должно быть уникально во всей таблице, тогда как значения каждого столбца по отдель-

ности уникальными не должны быть (и обычно не будут).

Уникальному ограничению можно назначить имя:

CREATE TABLE products (

product\_no integer CONSTRAINT must\_be\_unique UNIQUE,

name varchar(120),

price numeric

);

Ограничение уникальности нарушается, если в таблице оказывается несколько строк, у которых

совпадают значения всех столбцов, включённых в ограничение.

При этом два значения NULL при сравнении никогда не считаются равными.

Из этого следует, что даже при наличии ограничения уникальности в таблице можно сохранить

строки с дублирующимися значениями, если те содержат NULL в одном или нескольких столбцах,

участвующих в ограничении.

Это поведение соответствует стандарту SQL, но существуют СУБД, которые ведут себя иначе.

Первичные ключи

Ограничение первичного ключа означает, что образующий его столбец или группа столбцов может

быть уникальным идентификатором строк в таблице. Для этого требуется, чтобы значения были од-

новременно уникальными и отличными от NULL. Два следующих определения почти эквивалентны:

CREATE TABLE products (

product\_no integer UNIQUE NOT NULL,

name varchar(120),

price numeric

);

CREATE TABLE products (

product\_no integer PRIMARY KEY,

name varchar(120),

price numeric

);

Первичные ключи могут включать несколько столбцов.

Синтаксис похож на запись ограничений уникальности:

CREATE TABLE example (

a integer,

b integer,

c integer,

PRIMARY KEY (a, c)

);

При добавлении первичного ключа автоматически создаётся уникальный индекс B-дерево для ‒

столбца или группы столбцов, перечисленных в первичном ключе, и данные столбцы помечаются как

NOT NULL.

Таблица может иметь максимум один первичный ключ.

ОШИБКА: таблица "ex2" не может иметь несколько первичных ключей

СТРОКА 3: b integer PRIMARY KEY

Ограничений уникальности и ограничений NOT NULL, которые функционально почти равнозначны

первичным ключам, может быть сколько угодно, но назначить ограничением первичного ключа можно

только одно.

Теория реляционных баз данных говорит, что первичный ключ должен быть в каждой таблице и

ему стоит следовать, даже если это требование не является обязательным в данной реализации.

Внешние ключи

Ограничение внешнего ключа указывает, что значения столбца (или группы столбцов) должны со-

ответствовать значениям в некоторой строке другой таблицы.

Это называется ссылочной целостностью двух связанных таблиц.

Например, имеем таблицу:

CREATE TABLE products (

product\_no integer PRIMARY KEY,

name varchar(120),

price numeric

);

Пусть существует таблица заказов этих продуктов. Нужно, чтобы в таблице заказов содержались

только заказы действительно существующих продуктов. Поэтому в ней определяется ограничение

внешнего ключа, ссылающееся на таблицу продуктов:

CREATE TABLE orders (

id integer PRIMARY KEY,

product\_no integer REFERENCES products (product\_no),

quantity integer

);

С таким ограничением создать заказ со значением product\_no, отсутствующим в таблице

products (и не равным NULL), будет невозможно.

В такой схеме таблицу orders называют подчинённой таблицей, а products — главной.

Соответственно, столбцы называют так же подчинённым и главным (или ссылающимся и целевым).

Подчиненную таблицу можно записать короче:

CREATE TABLE orders (

id integer PRIMARY KEY,

product\_no integer REFERENCES products, -- ссылается на products PRIMATRY\_KEY

quantity integer

);

Если список столбцов опущен, внешний ключ будет связан с первичным ключом главной таблицы

неявно.

Ограничению внешнего ключа можно назначить имя.

Внешний ключ может ссылаться на группу столбцов. В этом случае его нужно записать в виде

обычного ограничения таблицы:

CREATE TABLE t1 (

a integer PRIMARY KEY,

b integer,

c integer,

FOREIGN KEY (b, c) REFERENCES master\_table (c1, c2)

);

Число и типы столбцов в ограничении должны соответствовать числу и типам целевых столбцов.

Дерево

Используя ограничение внешнего ключа, ссылающегося на свою же таблицу, можно организовать

иерархическую структуру связей (в направлении к корню):

CREATE TABLE tree (

node\_id integer PRIMARY KEY,

parent\_id integer REFERENCES tree,

name varchar(120),

...

);

Для узла верхнего уровня parent\_id будет равен NULL, а записи со значением parent\_id, отлич-

ным от NULL, будут ссылаться только на существующие строки таблицы.

Таблица может содержать несколько ограничений внешнего ключа — это используется для связи

таблиц в отношении многие-ко-многим. В частности, это можно использовать для организации дерева

и даже произвольного графа.

Положим, есть таблицы продуктов и заказов, но нужно, чтобы один заказ мог содержать несколько

продуктов. Это выглядит так:

CREATE TABLE products (

product\_no integer PRIMARY KEY,

name varchar(120),

price numeric

);

CREATE TABLE orders (

id integer PRIMARY KEY,

shipping\_address text,

...

);

CREATE TABLE order\_items (

product\_no integer REFERENCES products,

order\_id integer REFERENCES orders,

quantity integer,

PRIMARY KEY (product\_no, order\_id)

);

В последней таблице первичный ключ накрывает внешние ключи.  
  
  
11. SQL. Изменения таблиц. Команда Alter

Изменение таблиц (ALTER)

Созданную и заполненную данными таблицу можно модифицировать в отношении структуры и/

или ограничений. Postgres для этой цели предоставляет набор команд модификации таблиц.

Можно:

- изменить порядок сортировки в БД;

- переименовывать базы данных;

- изменять параметры базы данных;

- добавлять/удалять столбцы;

- добавлять/удалять ограничения;

- изменять значения по умолчанию;

- изменять типы столбцов;

- переименовывать столбцы;

- переименовывать таблицы;

db=> \h ALTER

db=> \h ALTER TABLE

12. Добавление и удаление данных в SQL.

Добавление/удаление столбца

ALTER TABLE table\_name ADD COLUMN [ IF NOT EXISTS ]column\_name type [ DEFAULT value ];

Новый столбец заполняется заданным для него значением по умолчанию (DEFAULT) или значением

NULL, если ничего не будет указано в качестве DEFAULT.

ALTER TABLE products ADD COLUMN description text;

Можно сразу определить ограничения столбца, используя обычный синтаксис:

ALTER TABLE products ADD COLUMN description text CHECK (description <> '');

Можно использовать любые конструкции, допустимые в определении столбца в команде CREATE TABLE.

Значение по умолчанию, если есть, должно удовлетворять данным ограничениям. Поэтому сначала нужно заполнить столбец и только потом добавить ограничение.

Удаление столбца

ALTER TABLE table\_name DROP COLUMN [ IF EXISTS ] column\_name;

Данные, которые были в удаляемом столбце, исчезают.

Вместе со столбцом удаляются все ограничения таблицы, его включающие.

Если на столбец ссылается ограничение внешнего ключа из другой таблицы, это ограничение не

будет удалено, если не предпринять дополнительных мер.

Разрешить удаление всех зависящих от этого столбца объектов можно, добавив указание CASCADE:

ALTER TABLE products DROP COLUMN description CASCADE;

Добавление/удаление ограничения

Для добавления ограничения используется синтаксис ограничения таблицы:

ALTER TABLE products ADD CHECK (name <> '');

ALTER TABLE products ADD CONSTRAINT some\_name UNIQUE (product\_no);

ALTER TABLE products ADD FOREIGN KEY (product\_group\_id) REFERENCES product\_groups;

Ограничение NOT NULL нельзя записать в виде ограничения таблицы, поэтому делаем так:

ALTER TABLE products ALTER COLUMN product\_no SET NOT NULL;

Ограничение автоматически проходит проверку и, если данные таблицы ему удовлетворяют, огра-

ничение будет добавлено.

Удаление ограничения

Чтобы удалить ограничение, нужно знать его имя.

Если имя не было явно присвоено пользователем, это неявно сделала система. Поэтому нужно сна-

чала выяснить это имя, используя, например, команду psql:

\d[S+] name -- описание таблицы, представления, последовательности или индекса.

Узнав имя, можно удалить ограничение:

ALTER TABLE products DROP CONSTRAINT constraint\_name;

У ограничений NOT NULL нет имён, поэтому для его удаления следует делать так:

ALTER TABLE products ALTER COLUMN product\_no DROP NOT NULL;

Изменение значения по умолчанию

Новое значение по умолчанию:

ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;

Это не влияет на существующие строки таблицы, а задает значение по умолчанию для последующих команд INSERT.

Удаление значения по умолчанию:

ALTER TABLE products ALTER COLUMN price DROP DEFAULT;

При этом по сути значению по умолчанию просто присваивается NULL. Как следствие, ошибки не

будет, если вы попытаетесь удалить значение по умолчанию, не определённое явно, так как неявно оно существует и равно NULL.

Изменение типа данных столбца

ALTER TABLE table\_name ALTER COLUMN column\_name TYPE new\_type

Пример:

ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);

Операция будет успешна, только если все существующие значения в столбце

могут быть неявно приведены к новому типу.

Если требуется более сложное преобразование, необходимо добавить указание USING, определяющее, как получить новые значения из старых.

Важно:

Postgres попытается преобразовать к новому типу значение столбца по умолчанию и все связанные

с этим столбцом ограничения. Преобразование может оказаться некорректным, и в результате случится

нехорошее. Поэтому, перед тем как менять тип столбца, следует удалить все его ограничения, а потом заново установить ограничения, модифицированные в соответствии с новым типом.

Переименование столбца

ALTER TABLE table\_name RENAME COLUMN column\_name TO new\_column\_name;

ALTER TABLE products RENAME COLUMN product\_no TO product\_number;

Переименование таблицы

ALTER TABLE table\_name RENAME TO new\_table\_name;

ALTER TABLE products RENAME TO items;

12. Предложение FROM в SQL. Пример.

Предложение FROM

Предложение FROM образует таблицу из одной или нескольких ссылок на таблицы, разделённых

запятыми.

FROM table\_reference [, table\_reference [, ...]]

Здесь ссылкой на таблицу table\_reference может быть:

- имя таблицы (как вариант, с именем схемы);

- производная таблица, например подзапрос;

- соединение таблиц;

- произвольная комбинация вышеупомянутых вариантов.

Если в предложении FROM перечисляются несколько ссылок, для них применяется перекрёстное соединение (CROSS JOIN — декартово произведение их строк).

Список FROM преобразуется в промежуточную виртуальную таблицу, которая может пройти че-

рез преобразования WHERE, GROUP BY и HAVING, и в итоге определит результат табличного выра-

жения.

Если в ссылке на таблицу указывается таблица, являющаяся родительской в иерархии наследования, в результате будут получены строки не только этой таблицы, но и всех её дочерних таблиц.

Чтобы выбрать строки только одной родительской таблицы, перед её именем нужно добавить ключевое слово ONLY. При этом будут получены только столбцы указанной таблицы — дополнительные столбцы дочерних таблиц в результат не попадут.

Чтобы явно указать, обработку всех дочерних таблиц, следует дописать после родительской \*.

13. Предложение WHERE в SQL. Пример.

Предложение WHERE

После обработки предложения FROM каждая строка полученной виртуальной таблицы проходит

проверку по условию ограничения через предложение WHERE:

WHERE search\_condition

где search\_condition — любое выражение, выдающее результат типа boolean (предикат).

Если результат условия равен true, эта строка остаётся в выходной таблице, а если результат ра-

вен false или NULL, отбрасывается.

В условии ограничения, как правило, используется минимум один столбец из таблицы, получен-

ной на выходе FROM. Примеры:

SELECT ... FROM foo WHERE c1 > 5

SELECT ... FROM foo WHERE c1 IN (1, 2, 3)

SELECT ... FROM foo WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM foo WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = foo.c1 + 10)

SELECT ... FROM foo WHERE c1 BETWEEN

(SELECT c3 FROM t2 WHERE c2 = foo.c1 + 10) AND 100

SELECT ... FROM foo WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > foo.c1)

foo — название таблицы, порождённой в предложении FROM. Строки, которые не соответствуют

условию WHERE, исключаются из foo.

14. Berkeley DB. Возможности и особенности.

Berkeley DB

1) Berkeley DB — это встроенная библиотека баз данных, которая предоставляет приложениям

масштабируемые, высокопроизводительные, защищенные транзакциями сервисы управления данными.

2) Berkeley DB является «встроенной», поскольку она напрямую связана с приложением. Она ра-

ботает в том же адресном пространстве, что и приложение. В результате для операций с базой данных не требуется никаких межпроцессных взаимодействий ни по сети, ни между процессами на одном компьютере.

3) Berkeley DB предоставляет простой API вызова функций для доступа к данным и управления

ими для ряда языков программирования, включая C, C++, Java, Perl, Tcl, Python и PHP.

4) Все операции с базой данных происходят внутри библиотеки.

Несколько процессов или несколько потоков в одном процессе могут использовать базу данных одновременно, поскольку каждый из них использует библиотеку Berkeley DB.

5) Все низкоуровневое обслуживание, такое как блокировки, журналирование транзакций, управление общими буферами, управление памятью и т. д., библиотекой обрабатываются прозрачно.

6) Библиотека Berkeley DB чрезвычайно переносима. Он работает практически под всеми вариантами UNIX и Linux, Windows и многими встраиваемыми операционными системами реального времени.

7) Она работает как на 32-битных, так и на 64-битных системах.

8)Может быть развернута на высокопроизводительных интернет-серверах, настольных компьютерах, а также на карманных компьютерах, телеприставках, сетевых коммутаторах и в других местах.

9) Berkeley DB хорошо масштабируется. Сама библиотека базы данных довольно компактна (около мегабайта памяти, занимаемой исполняемым кодом на обычных архитектурах), но при этом может использовать гигабайты памяти и терабайты дискового пространства, если дисковое оборудование это позволяет.

10) Каждый из файлов базы данных Berkeley DB может содержать до 256 терабайт данных при

условии, что базовая файловая система способна поддерживать файлы такого размера.

Приложения Berkeley DB часто используют несколько файлов базы данных и объем данных, кото-

рым может управлять приложение Berkeley DB, ограничен только ограничениями, накладываемыми

операционной системой, файловой системой и физическим оборудованием.

11) Berkeley DB также поддерживает высокий уровень параллелизма, что позволяет тысячам пользователей одновременно работать с одними и теми же файлами базы данных.

15. Berkeley DB. Понятие базы данных.

**Berkeley DB. Возможности и особенности**

**1. Встроенная (embedded) библиотека**

• Berkeley DB — это встроенная библиотека баз данных, которая предоставляет приложениям масштабируемые, высокопроизводительные, защищённые транзакциями сервисы управления данными.

• «Встроенная» означает, что библиотека работает в том же адресном пространстве, что и приложение, без дополнительных процессов или межпроцессных взаимодействий.

**2. Простые интерфейсы вызова**

• Berkeley DB предоставляет программный интерфейс (API) для ряда языков (C, C++, Java, Perl, Tcl, Python, PHP).

• Нет отдельного «сервера»; все операции происходят внутри библиотеки.

• Благодаря отсутствию сетевых или межпроцессных вызовов операции максимально быстры.

**3. Прозрачное обслуживание низкоуровневых механизмов**

• Библиотека сама занимается блокировками, транзакционным журналированием, управлением буферами, памятью и прочим.

• Разработчику не нужно писать код для решения низкоуровневых вопросов конкурентного доступа.

**4. Переносимость**

• Berkeley DB работает почти под всеми вариантами Unix/Linux, Windows и многими встраиваемыми (embedded/real-time) ОС.

• Поддерживаются и 32-битные, и 64-битные платформы.

**5. Масштабируемость**

• Ядро библиотеки «компактно» (примерно мегабайт кода), но при этом может работать с гигабайтами оперативной памяти и терабайтами на дисках.

• Один файл базы данных способен хранить до 256 ТБ (при условии поддержки таких размеров со стороны файловой системы).

• Можно использовать несколько файлов баз данных; единственное ограничение — ограничения оборудования и ОС.

**6. Высокий уровень параллелизма**

• Тысячи пользователей (или потоков, или процессов) могут одновременно работать с одними и теми же данными.

• Прозрачная блокировка не даёт пользователям мешать друг другу.

**7. ACID-транзакции**

• По умолчанию Berkeley DB обеспечивает «жёсткую» (строгую) модель транзакций ACID: атомарность, непротиворечивость, изолированность, долговечность.

• Возможно ослабление изоляции для повышения производительности.

**8. Восстановление (recovery)**

• После сбоя Berkeley DB может автоматически восстановить базу данных до согласованного состояния с учётом всех зафиксированных транзакций.

**9. Различные структуры хранения**

• Поддерживаются несколько основных типов хранения:

• Хэш-таблицы (подходят для быстрого доступа по ключу);

• B-деревья (хороши для диапазонных запросов и доступа к «соседним» ключам);

• Хранение по номерам записей (record numbers), когда удобнее автогенерировать ключи;

• Очереди (FIFO) для случаев, когда записи обрабатываются в порядке поступления.

Таким образом, **Berkeley DB** — это быстрый и компактный ключ-значение (key-value) движок, «встраиваемый» непосредственно в приложение, обеспечивающий надёжные транзакции и высокий уровень параллелизма.

**1. Не реляционная, не объектная и не сетевая СУБД**

• Berkeley DB не является реляционной базой данных и не имеет встроенного SQL (хотя отдельные проекты, например, MySQL, могут использовать её движок как низкоуровневое хранилище).

• В отличие от классических СУБД, Berkeley DB не навязывает никакого языка запросов и не хранит информацию о структуре (схеме) внутри базы данных.

**2. База данных как набор пар (ключ, значение)**

• Записи в Berkeley DB представляют собой пары *(ключ, значение)*, где и ключ, и значение — это просто массивы (строки) байтов.

• Никакой внутренней «схемы» у Berkeley DB нет. Библиотека не знает, что внутри ключа и значения; эта логика полностью на стороне приложения.

• Объём ключа или значения может доходить до 4 ГБ, что позволяет хранить, например, изображения или аудио.

**3. Принцип работы**

• Приложение вызывает функции библиотеки для вставки, поиска, удаления и обновления записей.

• Нет parsing SQL, нет оптимизатора. Всё, что делает Berkeley DB, — это быстрые операции над ключами.

• Приложение само отвечает за интерпретацию структуры записей и организацию индексов (кроме случаев, когда используется встроенная поддержка вторичных индексов, см. ниже).

**4. «Встроенный» характер**

• Berkeley DB не запускается как отдельный «сервер» — это просто набор функций, вызываемых приложением.

• Транзакционный и конкурентный доступ происходит внутри самой библиотеки, что даёт высокую скорость и управляемость.

**5. Минимальный набор операций**

• Вставить (put) пару *(ключ, значение)* в таблицу.

• Найти (get) по ключу.

• Удалить (delete) по ключу.

• Обновить (update) через механизмы поиска.

• При этом «значение» никак не анализируется библиотекой, и все операции над его внутренним содержимым ложатся на плечи приложения.

**16. Berkeley DB. Вторичные ключи**

В тексте выше явного описания «вторичных ключей» (secondary keys) нет, но в контексте Berkeley DB «вторичные ключи» обычно относятся к **вторичным индексам** (secondary indices):

1. **Что такое вторичный ключ (secondary key)**

• Основной ключ (primary key) — это тот, который хранится в «основной» таблице (B-дереве, хэш-таблице и т. д.).

• Вторичный ключ — это дополнительный индекс, который позволяет искать записи по альтернативному полю (или полям), отличному от основного ключа.

• В Berkeley DB вторичные ключи реализуются как ещё одна база данных (файл, структура хранения). Она хранит значения вторичных ключей и ссылается на основной ключ, позволяя «найти» основную запись.

2. **Как это работает**

• Для каждой записи в основной базе данных есть соответствующие записи в одной или нескольких «вторичных» базах данных (индексах).

• Когда приложение вставляет новую пару *(primary\_key, value)*, оно также вычисляет и вставляет *(secondary\_key, primary\_key)* во вторичный индекс.

• При поиске по вторичному ключу Berkeley DB возвращает соответствующие primary\_key, а уже по нему приложение (или сама библиотека) находит нужное значение в основной базе.

3. **Особенности реализации**

• Вторичный индекс чаще всего строится на основе B-дерева.

• Приложение может поддерживать несколько разных вторичных ключей (к примеру, по одному индексу искать «по имени», по другому — «по дате», и т. д.).

• Все транзакционные свойства (ACID) распространяются и на операции с вторичными индексами. Если операция добавления или удаления основной записи откатывается, откатывается и модификация соответствующего вторичного индекса.

4. **Преимущества и ограничения**

• Позволяет быстро искать данные по альтернативным полям без необходимости дублировать основную структуру данных.

• Требует дополнительного места на диске для хранения вторичных ключей и дополнительного времени для поддержания согласованности данных (синхронизации основного и вторичного хранилища).

Таким образом, **«вторичные ключи»** в Berkeley DB дают возможность создавать дополнительные индексы и эффективно выполнять запросы по неосновным полям, при этом сохраняя все особенности: транзакции, блокировки, поддержку больших объёмов данных и т. д.

17. Транзакции. Начало, завершение, откат.

Транзакции

Явным образом транзакции создаются командой START TRANSACTION и завершают-

ся командой COMMIT (фиксация, регистрация) или ROLLBACK (откат, прерывание).

Выполнять START TRANSACTION, чтобы начать блок транзакции, необязательно –

блок неявно начинает любая команда SQL. Поведение Postgres можно представить как

неявное выполнение COMMIT после каждой команды, которой не предшествует START

TRANSACTION (или BEGIN), и поэтому такое поведение часто называется «автофиксаци-

ей». Другие реляционные СУБД тоже могут обеспечивать автофиксацию.

Транзакции могут разбиваться на части – субтранзакции.

Субтранзакции можно также запускать в других субтранзакциях.

Транзакция верхнего уровня и её дочерние подтранзакции формируют иерархию или

дерево, поэтому по отношению к основной транзакции используется термин «транзакция

верхнего уровня».

Субтранзакции запускаются внутри транзакций и позволяют разбивать более крупные

транзакции на компоненты, которые могут фиксироваться или прерываться, не влияя на ро-

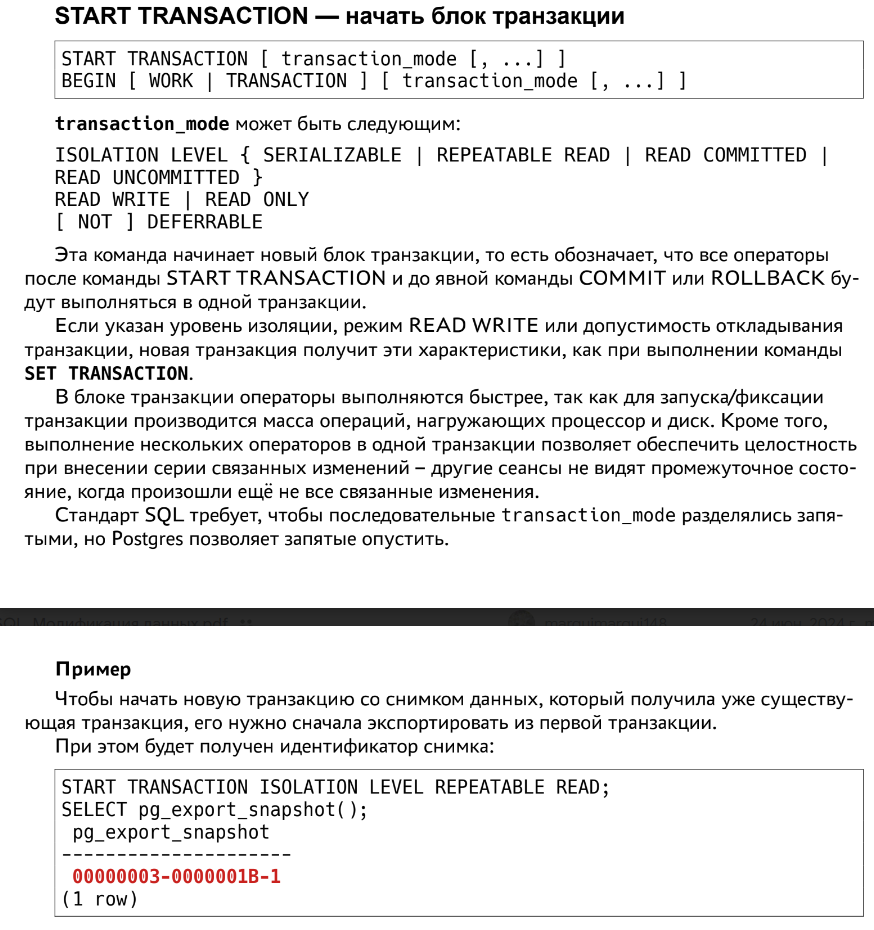
дительские транзакции, которые, соответственно, могут продолжать выполняться.

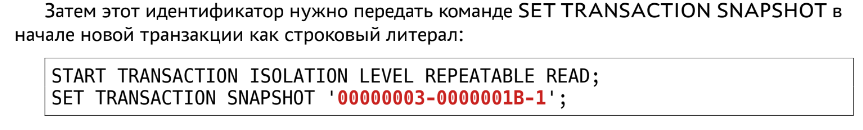
Такое решение позволяет обрабатывать ошибки проще и часто используется при разра-

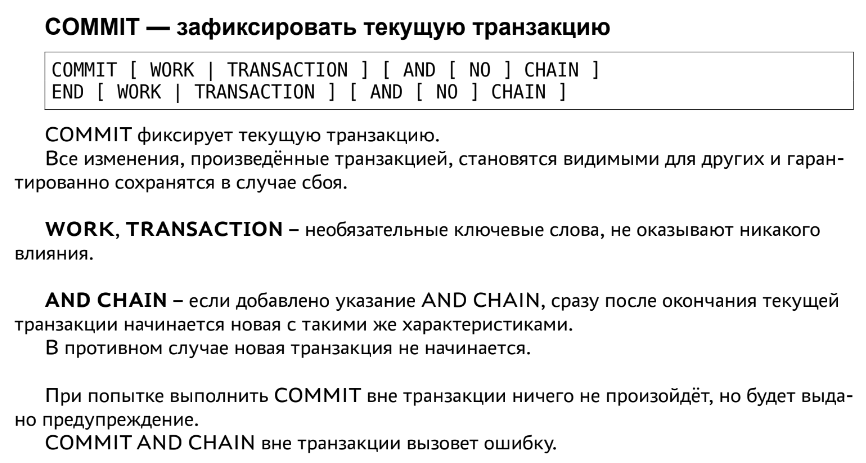
ботке приложений. В английском языке слово субтранзакция зачастую сокращается до

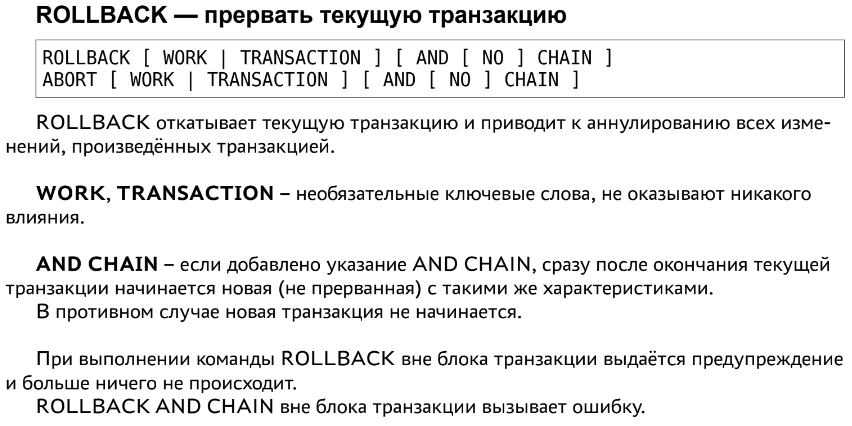
subxact.

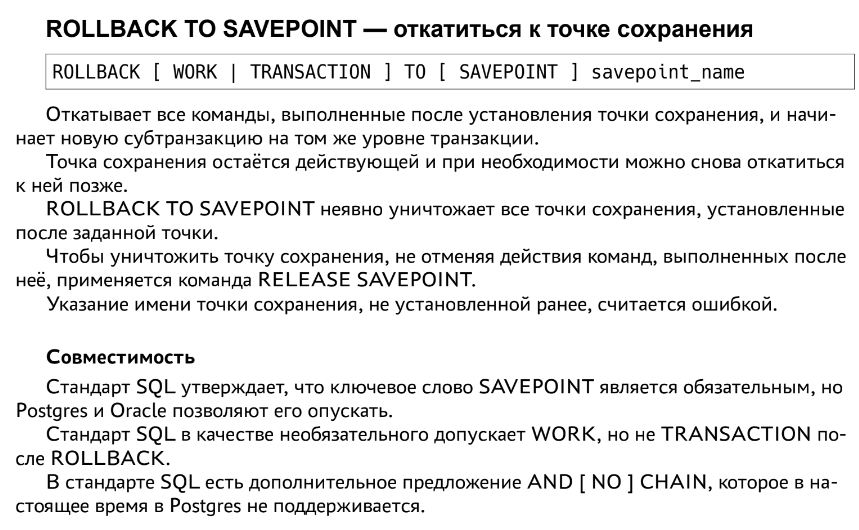
Субтранзакции могут явным образом запускаться при помощи команды SAVEPOINT.











**19. Транзакции и их необходимость**

**Транзакция** — это единица работы с базой данных, которая гарантирует, что все операции внутри неё выполняются как единое целое. Транзакция должна либо завершиться полностью (с фиксацией всех изменений), либо откатиться до исходного состояния (если возникает ошибка или необходимость отмены).

**Основные свойства транзакций (ACID):**

1. **Atomicity (Атомарность)**:

Все операции транзакции выполняются либо полностью, либо не выполняются вовсе. Это исключает ситуацию, когда часть данных обновлена, а часть нет.

2. **Consistency (Согласованность)**:

После завершения транзакции данные остаются в согласованном состоянии, соблюдаются все ограничения и правила целостности базы.

3. **Isolation (Изоляция)**:

Результаты выполнения транзакции не видны другим транзакциям до её завершения. Это предотвращает конфликты при конкурентном доступе.

4. **Durability (Долговечность)**:

После фиксации транзакции изменения сохраняются даже в случае сбоя системы.

**Необходимость транзакций**

1. **Гарантия целостности данных**:

Транзакции защищают данные от частично выполненных операций, обеспечивая их согласованность.

2. **Обеспечение согласованности многопользовательской работы**:

При одновременной работе нескольких пользователей транзакции предотвращают конфликты и сохраняют корректность данных.

3. **Управление ошибками**:

В случае ошибки транзакция позволяет откатить все изменения, что предотвращает повреждение данных.

4. **Поддержка сложных операций**:

Многие задачи (например, перевод средств между счетами) требуют выполнения нескольких связанных операций, которые должны быть выполнены либо полностью, либо не выполнены вовсе.

5. **Повышение надёжности системы**:

Транзакции обеспечивают защиту данных при сбоях, позволяя восстанавливать базу в согласованное состояние.

START TRANSACTION;

UPDATE accounts SET balance = balance - 100 WHERE id = 1;

UPDATE accounts SET balance = balance + 100 WHERE id = 2;

COMMIT;

Транзакции необходимы для обеспечения целостности, согласованности и надёжности данных в любой системе управления базами данных. Они особенно важны в многопользовательских и критически важных системах, таких как банковские или торговые приложения.

20. Алгоритмы конкурентного доступа (управления параллелизмом) в базах данных

призваны обеспечить целостность данных и корректное выполнение транзакций при одновременном доступе нескольких пользователей (или процессов) к одной и той же базе. Основная цель — избежать неконсистентных состояний и защититься от таких проблем, как «грязное» чтение, «потерянное» обновление, неповторяющееся чтение и пр.

Ниже рассмотрены основные механизмы и подходы, применяемые при организации конкурентного доступа.

**1. Базовые понятия**

1. **Транзакция**

Логическая единица работы с данными, которая либо выполняется целиком (commit), либо полностью откатывается (rollback), если в ходе выполнения произошла ошибка или конфликт.

2. **Изоляция транзакций**

Гарантирует, что транзакции не будут мешать друг другу и данные, которые использует каждая транзакция, являются корректными с точки зрения заданного уровня изоляции.

3. **Уровни изоляции** (по стандарту ANSI/ISO SQL):

• Read Uncommitted (самый «свободный» режим, допускающий «грязные» чтения),

• Read Committed (самый распространённый уровень, не даёт читать незакоммиченные данные),

• Repeatable Read (исключает «грязные» и «неповторяющиеся» чтения),

• Serializable (самый строгий уровень, делает транзакции по сути «последовательно» исполняемыми).

**2. Основные проблемы при параллельном доступе**

1. **«Грязное» чтение (Dirty Read)**

Происходит, если одна транзакция читает данные, которые ещё не были зафиксированы (commit) другой транзакцией. Если та транзакция откатится (rollback), получается, что первая «увидела» несуществующее состояние данных.

2. **Неповторяющееся чтение (Non-repeatable Read)**

Транзакция дважды читает одну и ту же строку и получает разные результаты из-за того, что другая транзакция успела обновить данные между двумя чтениями.

3. **Фантомное чтение (Phantom Read)**

Транзакция при повторном запросе видит новые строки, которых не было в предыдущем запросе (другая транзакция вставила новые записи).

4. **Потерянное обновление (Lost Update)**

Две транзакции одновременно читают и модифицируют одну и ту же запись, и «побеждает» только последняя операция, при этом результат изменений первой операции не учитывается.

**3. Подходы к управлению конкурентным доступом**

**3.1 Двухфазная блокировка (Two-Phase Locking, 2PL)**

1. **Суть**

• Каждый объект (запись или набор записей) блокируется при чтении или записи.

• Блокировка бывает двух типов: Shared (S) — совместная, для чтения, и Exclusive (X) — эксклюзивная, для записи.

• Транзакция не может менять тип блокировки «на лету»: сначала все блокировки выставляются (растущая фаза), а затем все блокировки освобождаются (убывающая фаза).

2. **Плюсы**

• Простой и понятный подход.

• Широко используется в традиционных реляционных СУБД.

3. **Минусы**

• Возможны «глобальные» блокировки (Table-level locking), которые снижают уровень параллелизма при большом количестве одновременно выполняемых транзакций.

• При неправильном проектировании схемы блокировок могут возникать взаимоблокировки (deadlocks).

4. **Улучшения и нюансы**

• Гранулярность блокировок: от уровня всей таблицы до отдельных строк или даже отдельных полей.

• Управление транзакциями может использовать эскалацию блокировок, когда при слишком большом количестве строковых блокировок система переходит на блокировку всего раздела или таблицы.

**3.2 Оптимистическое управление параллелизмом (Optimistic Concurrency Control, OCC)**

1. **Суть**

• Предполагается, что конфликты при одновременном доступе редки.

• Каждая транзакция работает со «своей» копией данных, не ставя жёстких блокировок на время чтения/записи.

• Перед фиксацией (commit) система проверяет, не изменились ли за это время данные, на которые опирается транзакция (версионирование или контроль по временным меткам). Если обнаруживается конфликт, транзакция откатывается и повторяется.

2. **Плюсы**

• Высокий уровень параллелизма при низкой вероятности конфликтов.

• Нет долгих блокировок, можно эффективно обслуживать большое количество транзакций.

3. **Минусы**

• При высокой вероятности конфликтов многие транзакции будут откатываться, что приводит к снижению производительности.

• Необходимо надёжное версионирование или механизм фиксации.

**3.3 Многоверсионная конкурирующая контроль (MVCC — Multiversion Concurrency Control)**

1. **Суть**

• При обновлении строки в таблице не затирается предыдущая версия данных, а создаётся новая версия.

• Читающие транзакции обращаются к «актуальной для себя» версии строки, в то время как записывающие транзакции могут создавать новые версии.

• Таким образом, чтение не блокируется записью и наоборот, что повышает параллелизм.

2. **Как работает**

• Каждой версии строки назначаются метки (timestamps) или номера транзакций (transaction ID), чтобы понять, какие транзакции «видят» эти данные.

• При фиксации (commit) происходит «переключение» на новую версию для будущих транзакций.

• Устаревшие версии со временем удаляются сборщиком мусора (vacuum).

3. **Плюсы**

• Чтения не блокируются, меньше конфликтов.

• Часто повышает производительность в системах, где преобладают операции чтения.

4. **Минусы**

• Усложняет реализацию СУБД.

• Требует управления большим количеством версий. При больших объёмах данных нужно регулярно «чистить» старые версии, чтобы избежать разрастания хранилища.

**4. Этапы работы типичного алгоритма (на примере MVCC)**

1. **Начало транзакции**

• Транзакции присваивается внутренний номер (Transaction ID).

• Фиксируется «момент времени» или «срез данных», видимых для транзакции.

2. **Чтение данных**

• Транзакция читает ту версию строки, которая была актуальна на момент её старта (или на момент чтения, в зависимости от настроек изоляции).

• Запрещены «грязные» чтения, т.к. невидимые версии (незакоммиченные) не будут отображаться.

3. **Модификация (запись) данных**

• Создаётся новая версия изменяемой строки с отметкой, что она принадлежит данной транзакции.

• Старая версия продолжает существовать для тех, кто начал транзакцию раньше.

4. **Проверка целостности и конфликтов** (может быть реализована по-разному в разных механизмах)

• Если СУБД обнаружит конфликт (например, другая транзакция уже изменила эти же данные и зафиксировала), текущая транзакция может быть откатана или ждать (зависит от уровня изоляции).

5. **Фиксация (commit)**

• Новые версии строк становятся «актуальными» для всех будущих транзакций.

• Старая версия помечается «устаревшей» и впоследствии будет удалена сборщиком мусора.

6. **Откат (rollback)**

• Все изменения этой транзакции признаются недействительными, соответствующие новые версии строк удаляются или помечаются как «не существующие».

**5. Взаимоблокировки (Deadlock) и способы их решения**

Даже при правильном использовании блокировок и MVCC могут возникать взаимоблокировки, когда две транзакции ждут освобождения ресурсов друг от друга бесконечно.

1. **Deadlock Detection** (обнаружение)

• СУБД периодически проверяет циклы во взятых блокировках: если цикл найден, одна из транзакций откатывается.

2. **Deadlock Prevention** (предотвращение)

• Оптимизировать порядок блокировок в коде приложения: например, блокировать ресурсы всегда в одном и том же порядке.

• Использовать более короткие транзакции, чтобы уменьшить время «пребывания» блокировок.

**6. Рекомендации по организации конкурентного доступа**

1. **Выбирать правильный уровень изоляции**

• Использовать Read Committed по умолчанию и повышать уровень только при реальной необходимости (Repeatable Read, Serializable).

2. **Избегать слишком длительных транзакций**

• Держать транзакции «как можно короче» по времени и объёму работы.

3. **Минимизировать риск взаимоблокировок**

• Придерживаться согласованного порядка обращений к ресурсам.

• Использовать короткие SELECT перед UPDATE/DELETE.

4. **Следить за «чистотой» MVCC**

• Периодически запускать процедуры «уборки» (vacuum) в PostgreSQL и других СУБД, поддерживающих MVCC.

• Контролировать рост «хвоста» неактуальных версий.

5. **Понимать специфику СУБД**

• В разных системах (Oracle, MySQL, PostgreSQL, MSSQL) механизмы блокировок и MVCC могут отличаться деталями реализации, что может сильно влиять на производительность.

**Итого**

Конкурентный доступ к базе данных — важнейшая часть архитектуры любой СУБД, обеспечивающая целостность данных при параллельных операциях чтения и записи. На практике чаще всего используются:

• **Двухфазная блокировка (2PL)**, как классический способ синхронизации;

• **Многоверсионный контроль (MVCC)**, который повышает параллелизм и уменьшает количество блокировок при чтении;

• **Оптимистические подходы** (OCC), эффективные при невысокой конфликтности.

Выбор конкретного алгоритма или комбинации механизмов зависит от требований к производительности, ожидаемой нагрузки, профиля запросов (преобладают ли чтения или записи) и допустимого уровня консистентности.