**1. Распределённые и централизованные базы данных. Архитектура файл-сервер. Архитектура клиент-сервер.**

**Централизованные системы баз данных** – системы, в которых единственная логическая база данных размещалась в пределах одного узла и находилась под управлением одной СУБД. СУБД и централизация обработки информации позволили устранить такие недостатки традиционных файловых систем, как несвязанность, несогласованность и избыточность данных. По мере роста баз данных и особенно при их использовании в территориально разделенных организациях появляются другие проблемы. Так, для централизованной СУБД, находящейся в узле телекоммуникационной сети, с помощью которой различные подразделения организации получают доступ к данным, с ростом объема информации и количества транзакций возникают следующие трудности: большой поток обменов данными; низкая надежность; низкая общая производительность; большие затраты на разработку.

Для снижения остроты недостатков создают распределённые базы данных.

**Распределённые базы данных (РБД)** – совокупность логически взаимосвязанных баз данных, распределённых в компьютерной сети. РБД состоит из набора узлов, связанных коммуникационной сетью, в которой: каждый узел – это полноценная СУБД сама по себе; узлы взаимодействуют между собой таким образом, что пользователь любого из них может получить доступ к любым данным в сети так, как будто они находятся на его собственном узле. Каждый узел сам по себе является системой базы данных. Любой пользователь может выполнить операции над данными на своём локальном узле точно так же, как если бы этот узел вовсе не входил в распределённую систему.

**Архитектура файл-сервер.**

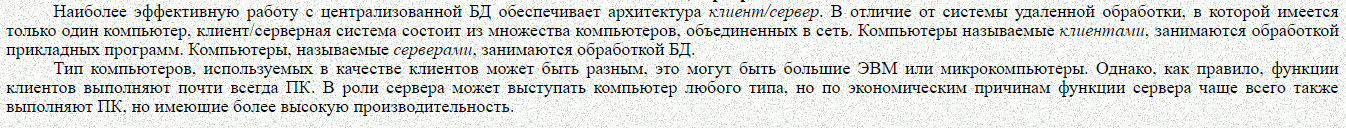
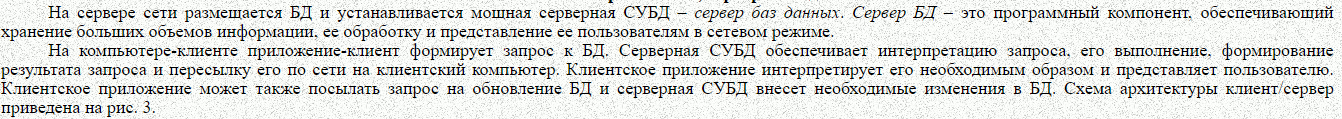
В архитектуре файл-сервер базы данных хранятся на сервере, клиент обращается к серверу с файловыми командами, а механизм управления всеми информационными ресурсами находится на компьютере клиента.

Файл-серверные базы данных могут быть доступны многим клиентам через сеть. Сама база данных хранится на сетевом файл-сервере в единственном экземпляре. Для каждого клиента во время работы создается локальная копия данных, с которой он манипулирует. При этом возникают проблемы, связанные с возможным одновременным доступом нескольких пользователей к одной и той же информации. Эти проблемы решаются разработчиками приложений баз данных

Архитектура файл-сервер обладает значительными недостатками. Одним из них является непроизводительная загрузка сети. При каждом запросе клиента данные в его локальной копии полностью обновляются из базы данных на сервере. Даже если запрос относится всего к одной записи, обновляются все записи базы данных. Если записей в базе данных много, то и при небольшом числе клиентов сеть будет загружена очень основательно, что серьезно скажется на скорости выполнения запросов.

Другой недостаток связан с тем, что забота о целостности данных при такой организации работы возлагается на программы клиентов. Одним из традиционных средств, на основе которых создаются файл-серверные системы, являются локальные СУБД. Такие системы, как правило, не отвечают требованиям обеспечения целостности данных, в частности, они не поддерживают обработки транзакций.

И еще один недостаток – управление базой данных осуществляется с разных компьютеров, поэтому в значительной степени затруднена организация контроля доступа, соблюдения конфиденциальности, что также усложняет поддержку целостности базы данных.

**Архитектура клиент-сервер.**

**2. Иерархическая и сетевая модели данных.**

**ИЕРАРХИЧЕСКАЯ МОДЕЛЬ ДАННЫХ.**

Иерархическая модель данных появилась впервые в результате обобщения структур данных языка Кобол. В иерархических моделях основная структура представления данных имеет форму дерева. На самом высшем уровне иерархии находится только одна вершина, которая называется корнем дерева. Эта вершина имеет связи с вершинами второго уровня, вершины второго уровня имеют связи с вершинами третьего уровня и т.д. Связи между вершинами одного уровня отсутствуют. Следовательно, данные в иерархической структуре не равноправны – одни жёстко подчинены другим. Доступ к информации возможен только по вертикальной схеме, начиная с корня, так как каждый элемент связан только с одним элементом на верхнем уровне и с одним или несколькими на низком.

Иерархическая модель является наиболее простой, поэтому исторически она появилась первой.

**Достоинства:** достаточно эффективное использование памяти и неплохие временные показатели выполнения операций над данными.

**Недостатки:** достаточно сложные логические связи и соответствующая громоздкость в обработке данных.

**Наиболее известные иерархические базы данных:** IMS, PC/Focus, Team-Up, Data Edge, Ока, ИНЭС, МИРИС.

**СЕТЕВАЯ МОДЕЛЬ ДАННЫХ.**

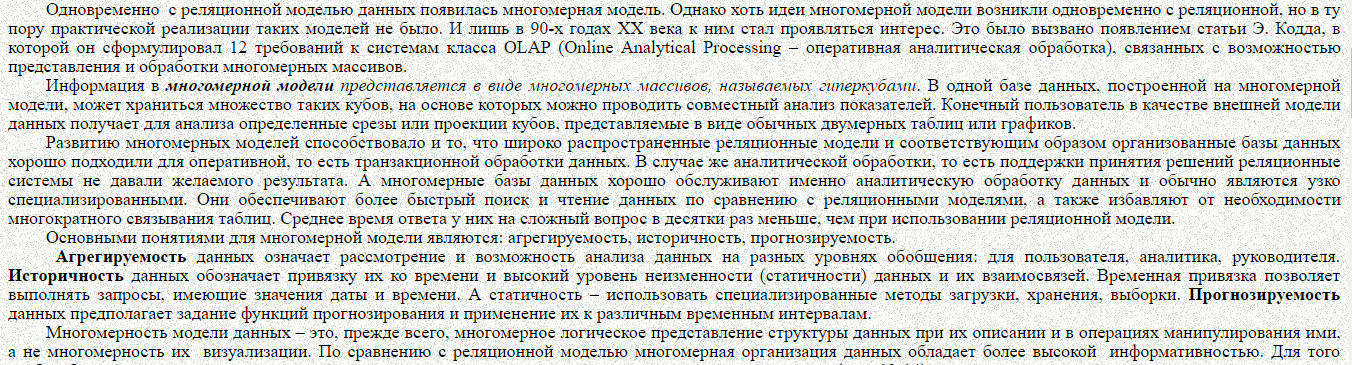
**Сетевая модель** – структура, у которой любой элемент может быть связан с любым другим элементом.

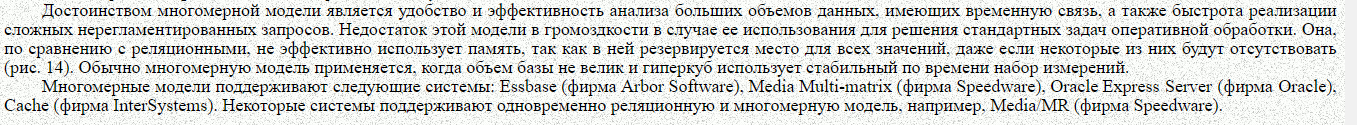
Сетевая база состоит из наборов записей, которые связаны между собой так, что записи могут содержать явные ссылки на другие наборы записей. Тем самым наборы записей образуют сеть. Связи между записями могут быть произвольными, и эти связи явно присутствуют и хранятся в базе данных.

**Достоинство:** высокая эффективность затрат памяти и оперативность.

**Недостаток:** сложность и жёсткость схемы базы, а также сложность понимания.

**К известным сетевым СУБД относятся:** DBMS, IDMS, TOTAL, VISTA, СЕТЬ, СЕТОР, КОМПАС.



**3. Многомерная модель данных.**

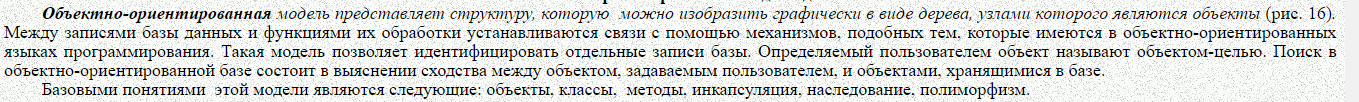
**4. Постреляционная модель данных.**

**Постреляционная модель** является расширением реляционной модели. Она снимает ограничение неделимости данных, допуская многозначные поля, значения которых состоят из подзначений, и набор значений воспринимается как самостоятельная таблица, встроенная в главную таблицу.

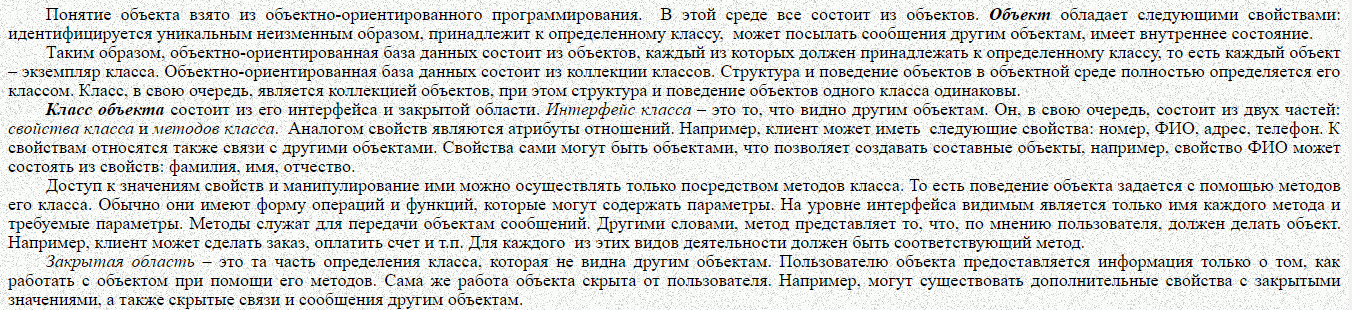
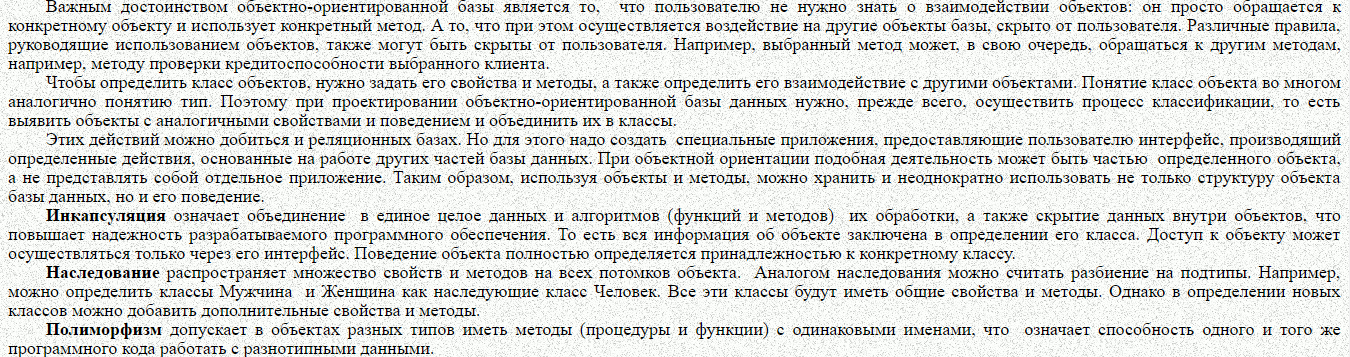
Спецификой постреляционной модели является то, что она поддерживает множественные группы, называемые ассоциированными множественными полями, а совокупность объединённых множественных полей называется ассоциацией. В постреляционной модели не накладываются требования на длину и количество полей в записях, что делает структуру таблиц более наглядной.

Достоинство: возможность представления совокупности связанных реляционных таблиц в виде одной постреляционной таблицы.

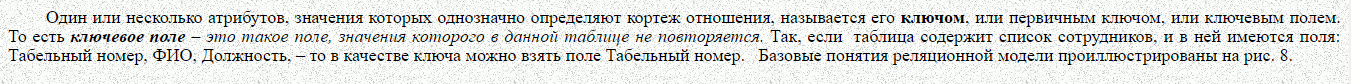
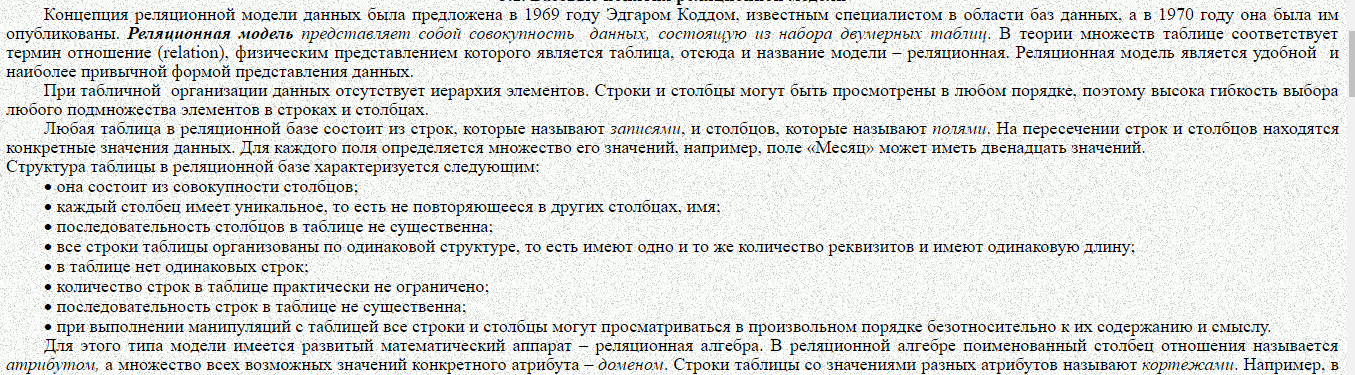
Недостаток: сложность обеспечения целостности и непротиворечивости данных, хранимых в ней.

**Постреляционная модель данных реализована в СУБД** uniVers, Bubba и Dasdb. ****

**5) Объектно-ориентированная модель данных.**



**6. Реляционная модель данных. История развития. Основные понятия (тип данных, домен, отношение, кортеж, атрибут, ключ).**



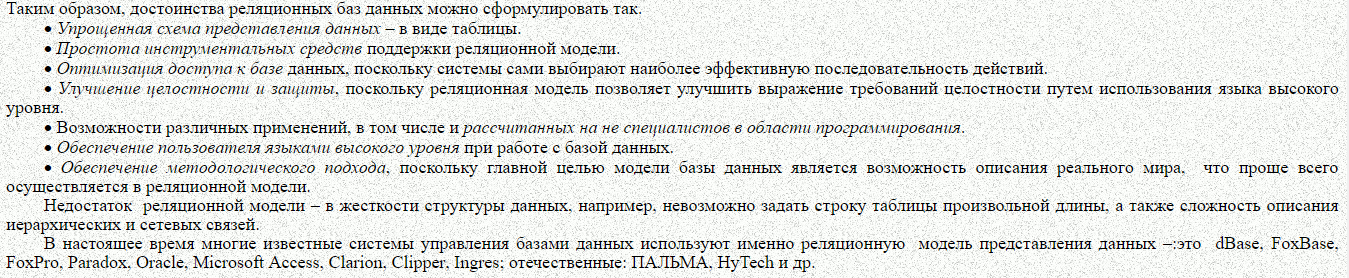
Тип данных определяется значениями, которые предполагается хранить в поле, и операциями, которые будут выполняться с этими значениями. Реляционная модель требует, чтобы типы используемых данных были простыми.

**Простые, или атомарные, типы данных** не обладают внутренней структурой. Данные такого типа **называют** **скалярами.** К простым типам данных относятся следующие типы:

* Логический
* Строковый
* Численный

Различные языки программирования могут расширять и уточнять этот список, добавляя такие типы как:

* Целый
* Вещественный
* Дата
* Время
* Денежный
* Перечислимый
* Интервальный
* И т. д.…



**7. Реляционная база данных.**

**Реляционная база данных** – это совокупность взаимосвязанных таблиц, каждая из которых содержит информацию об объектах определенного типа. Строка таблицы содержит данные об одном объекте (например, товаре, клиенте), а столбцы таблицы описывают различные характеристики этих объектов — атрибутов (например, наименование, код товара, сведения о клиенте). Записи, т. е. строки таблицы, имеют одинаковую структуру – они состоят из полей, хранящих атрибуты объекта. Каждое поле, т. е. столбец, описывает только одну характеристику объекта и имеет строго определенный тип данных. Все записи имеют одни и те же поля, только в них отображаются различные информационные свойства объекта.

В реляционной базе данных каждая таблица должна иметь первичный ключ – поле или комбинацию полей, которые единственным образом идентифицируют каждую строку таблицы. Если ключ состоит из нескольких полей, он называется составным. Ключ должен быть уникальным и однозначно определять запись. По значению ключа можно отыскать единственную запись. Ключи служат также для упорядочивания информации в БД.

Таблицы реляционной БД должны отвечать требованиям нормализации отношений.

Реляционные таблицы могут быть связаны друг с другом, следовательно, данные могут извлекаться одновременно из нескольких таблиц. Таблицы связываются между собой для того, чтобы в конечном счете уменьшить объем БД. Связь каждой пары таблиц обеспечивается при наличии в них одинаковых столбцов.

Существуют следующие типы информационных связей:

* один-к-одному;
* один-ко-многим;
* многие-ко-многим.

Связь один-к-одному предполагает, что одному атрибуту первой таблицы соответствует только один атрибут второй таблицы и наоборот.

Связь один-ко-многим предполагает, что одному атрибуту первой таблицы соответствует несколько атрибутов второй таблицы.

Связь многие-ко-многим предполагает, что одному атрибуту первой таблицы соответствует несколько атрибутов второй таблицы и наоборот.

**8. Функции системы управления базами данных (СУБД): управления данными во внешней памяти, управление буферами оперативной памяти, управление транзакциями.**

**Управления данными во внешней памяти.**

Эта функция предоставляет пользователю возможность выполнения основных операций с данными – хранение, извлечение и обновление информации. Она включает в себя обеспечение необходимых структур внешней памяти как для хранения данных, непосредственно входящих в БД, так и для служебных целей, например, для убыстрения доступа к данным. СУБД поддерживает собственную систему именования объектов БД.

**Управление буферами оперативной памяти.**

СУБД обычно работают с БД значительного размера; по крайней мере этот размер обычно существенно больше доступного объема оперативной памяти. Понятно, что если при обращении к любому элементу данных будет производиться обмен с внешней памятью, то вся система будет работать со скоростью устройства внешней памяти. Практически единственным способом реального увеличения этой скорости является буферизация данных в оперативной памяти. Однако этого недостаточно для целей СУБД. Поэтому в развитых СУБД поддерживается собственный набор буферов оперативной памяти.

**Управление транзакциями.**

Транзакция – это последовательность операций над БД, которые рассматриваются СУБД как единое целое и позволяют добавлять, удалять или обновлять сведения о некотором объекте в базе (по существу это некоторый программный код, написанный на одном из языков управления данными). Либо транзакция успешно выполняется, и СУБД фиксирует изменения БД, произведенные этой транзакцией, либо ни одно из этих изменений никак не отражается на состоянии БД. Например, если в результате транзакции произошел сбой компьютера, база данных попадает в противоречивое положение – некоторые изменения уже внесены, остальные нет.

Транзакция позволяет вернуть базу в первоначальное непротиворечивое состояние (отменить все выполненные изменения).

**9. Реляционная алгебра. Общая интерпретация реляционных операций.**

**Реляционная алгебра** – формальная система манипулирования отношениями в реляционной модели данных.

В реляционной модели данных выделяют раздел манипуляционная часть, согласно которому появилось утверждение о том, что доступ к реляционным данным осуществляется за счёт реляционной алгебры или реляционных исчислений.

Традиционно, вслед за Коддом, определяют восемь реляционных операторов, объединенных в две группы – теоретико-множественные операции и специальные реляционные операции.

**В состав теоретико-множественных операций входят операции:**

* объединения отношений;
* пересечения отношений;
* взятия разности отношений;
* декартово произведения отношений.

**Специальные реляционные операции включают:**

* ограничение отношения;
* проекцию отношения;
* соединение отношений;
* деление отношений.

Кроме того, в состав алгебры включается операция присваивания, позволяющая сохранить в базе данных результаты вычисления алгебраических выражений, и операция переименования атрибутов, дающая возможность корректно сформировать заголовок (схему) результирующего отношения.

**Общая интерпретация реляционных операций**

* При выполнении операции объединения двух отношений производится отношение, включающее все кортежи, входящие хотя бы в одно из отношений-операндов.
* Операция пересечения двух отношений производит отношение, включающее все кортежи, входящие в оба отношения-операнда.
* Отношение, являющееся разностью двух отношений включает все кортежи, входящие в отношение - первый операнд, такие, что ни один из них не входит в отношение, являющееся вторым операндом.
* При выполнении прямого произведения двух отношений производится отношение, кортежи которого являются конкатенацией (сцеплением) кортежей первого и второго операндов.
* Результатом ограничения отношения по некоторому условию является отношение, включающее кортежи отношения-операнда, удовлетворяющее этому условию.
* При выполнении проекции отношения на заданный набор его атрибутов производится отношение, кортежи которого производятся путем взятия соответствующих значений из кортежей отношения-операнда.
* При соединении двух отношений по некоторому условию образуется результирующее отношение, кортежи которого являются конкатенацией кортежей первого и второго отношений и удовлетворяют этому условию.
* У операции реляционного деления два операнда - бинарное и унарное отношения. Результирующее отношение состоит из одноатрибутных кортежей, включающих значения первого атрибута кортежей первого операнда таких, что множество значений второго атрибута (при фиксированном значении первого атрибута) совпадает со множеством значений второго операнда.
* Операция переименования производит отношение, тело которого совпадает с телом операнда, но имена атрибутов изменены.
* Операция присваивания позволяет сохранить результат вычисления реляционного выражения в существующем отношении БД.

**10. Особенности теоретико-множественных операций реляционной алгебры.**

Хотя в основе теоретико-множественной части реляционной алгебры лежит классическая теория множеств, соответствующие операции реляционной алгебры обладают некоторыми особенностями.

Начнем с операции объединения. Смысл операции объединения в реляционной алгебре в целом остается теоретико-множественным. Но если в теории множеств операция объединения осмысленна для любых двух множеств-операндов, то в случае реляционной алгебры результатом операции объединения должно являться отношение. Если допустить в реляционной алгебре возможность теоретико-множественного объединения произвольных двух отношений (с разными схемами), то, конечно, результатом операции будет множество, но множество разнотипных кортежей, т.е. не отношение. Если исходить из требования замкнутости реляционной алгебры относительно понятия отношения, то такая операция объединения является бессмысленной.

Все эти соображения приводят к появлению понятия совместимости отношений по объединению: два отношения совместимы по объединению в том и только в том случае, когда обладают одинаковыми заголовками. Более точно, это означает, что в заголовках обоих отношений содержится один и тот же набор имен атрибутов, и одноименные атрибуты определены на одном и том же домене.

Если два отношения совместимы по объединению, то при обычном выполнении над ними операций объединения, пересечения и взятия разности результатом операции является отношение с корректно определенным заголовком, совпадающим с заголовком каждого из отношений-операндов. Напомним, что если два отношения "почти" совместимы по объединению, т.е. совместимы во всем, кроме имен атрибутов, то до выполнения операции типа соединения эти отношения можно сделать полностью совместимыми по объединению путем применения операции переименования.

Другие проблемы связаны с операцией взятия прямого произведения двух отношений. В теории множеств прямое произведение может быть получено для любых двух множеств, и элементами результирующего множества являются пары, составленные из элементов первого и второго множеств. Поскольку отношения являются множествами, то и для любых двух отношений возможно получение прямого произведения. Но результат не будет отношением! Элементами результата будут являться не кортежи, а пары кортежей.

По поводу теоретико-множественных операций реляционной алгебры следует еще заметить, что все четыре операции являются ассоциативными. Т. е., если обозначить через OP любую из четырех операций, то (A OP B) OP C = A (B OP C), и следовательно, без введения двусмысленности можно писать A OP B OP C (A, B и C - отношения, обладающие свойствами, требуемыми для корректного выполнения соответствующей операции). Все операции, кроме взятия разности, являются коммутативными, т.е. A OP B = B OP A.

**11. Специальные реляционные операции.**

**Операция ограничения.**

В результате выполнения операции ограничения (WHERE) производится отношение, заголовок которого совпадает с заголовком отношения-операнда, а в тело входят те кортежи отношения-операнда, для которых значением условия ограничения является true.

**Операция взятия проекции.**

Результатом проекции (PROJECT) отношения A по списку атрибутов a1, a2, ..., an является отношение, с заголовком, определяемым множеством атрибутов a1, a2, ..., an, и с телом, состоящим из кортежей вида <a1:v1, a2:v2, ..., an:vn> таких, что в отношении A имеется кортеж, атрибут a1 которого имеет значение v1, атрибут a2 имеет значение v2, ..., атрибут an имеет значение vn. Тем самым, при выполнении операции проекции выделяется "вертикальная" вырезка отношения-операнда с естественным уничтожением потенциально возникающих кортежей-дубликатов.

**Операция соединения отношений.**

Соединением отношения A и B по условию c называется отношение (A TIMES B) WHERE c, где c представляет собой логическое выражение, в которое могут входить атрибуты отношений A и B и (или) скалярные выражения (число).

Имеется важный частный случай соединения - эквисоединение и простое, но важное расширение операции эквисоединения - естественное соединение. Операция соединения называется операцией эквисоединения, если условие соединения имеет вид (a = b), где a и b - атрибуты разных операндов соединения. Этот случай важен потому, что (a) он часто встречается на практике, и (b) для него существуют эффективные алгоритмы реализации.

Операция естественного соединения применяется к паре отношений A и B, обладающих (возможно составным) общим атрибутом c (т.е. атрибутом с одним и тем же именем и определенным на одном и том же домене). Пусть ab обозначает объединение заголовков отношений A и B. Тогда естественное соединение A и B - это спроектированный на ab результат эквисоединения A и B по A/c и BBC. Если вспомнить введенное нами в конце предыдущей главы определение внешнего ключа отношения, то должно стать понятно, что основной смысл операции естественного соединения - возможность восстановления сложной сущности, декомпозированной по причине требования первой нормальной формы. Операция естественного соединения не включается прямо в состав набора операций реляционной алгебры, но она имеет очень важное практическое значение.

**Операция деления отношений.**

Эта операция наименее очевидна из всех операций реляционной алгебры и поэтому нуждается в более подробном объяснении. Пусть заданы два отношения - A с заголовком {a1, a2, ..., an, b1, b2, ..., bm} и B с заголовком {b1, b2, ..., bm}. Будем считать, что атрибут bi отношения A и атрибут bi отношения B не только обладают одним и тем же именем, но и определены на одном и том же домене. Назовем множество атрибутов {aj} составным атрибутом a, а множество атрибутов {bj} - составным атрибутом b. После этого будем говорить о реляционном делении бинарного отношения A(a,b) на унарное отношение B(b).

Результатом деления A на B является унарное отношение C(a), состоящее из кортежей v таких, что в отношении A имеются кортежи <v, w> такие, что множество значений {w} включает множество значений атрибута b в отношении B.

**12. Нормализация отношений.**

**Нормализация** – процесс, направленный на уменьшение избыточности информации в БД (разбиение информации на более детальные составляющие).

**Нормальная форма** – показатель уровня и глубины нормализации БД.

**13. Этапы проектирования баз данных. Инфологическое проектирование.**

**Проектирование базы данных (БД)** – одна из наиболее сложных и ответственных задач, связанных с созданием информационной системы (ИС).

Процесс проектирования включает в себя следующие этапы:

1. Анализ и описание предметной области.
2. Инфологическое проектирование.
3. Логическое проектирование БД.
4. Выбор системы управления базой данных (СУБД) и других инструментальных программных средств.
5. Физическое проектирование БД.

**Инфологическое проектирование** – описание предметной области, выполненное с использованием специальных языковых средств, не зависящих от используемых в дальнейшем программных и технических средств.

На данном этапе в рамках предметной области необходимо выделить **сущности** и присвоить им соответствующие **атрибуты** (свойства).

**Сущность** – «множество реально-существующих или абстрактных индивидуальных объектов, которые обладают одинаковым набором свойств».

**Атрибутом** считается некоторое свойство объекта реального мира.

Инфологическая модель представляет собой наличие ER-диаграмм.   
ER-диаграмма представляет из себя наличие объектов, их свойств, а так же связей между ними. Проектируя инфологическую модель предметной области, необходимо построить одну общую ER-модель.

**14. 1, 2, 3 и 4 нормальные формы. Приведение базы данных к нормализованному виду.**

**Первая нормальная форма** (1 НФ) ⎯ отношение находится в 1НФ, если все его атрибуты являются атомарными (простыми, неделимыми).

**Вторая нормальная форма** (2 НФ) ⎯ отношение находится во второй нормальной форме тогда и только тогда, когда оно находится в 1НФ, и каждый не первичный атрибут функционально полно зависит от первичного ключа (помещение в отдельную табл. данных, которые частично зависят от первичного ключа).

**Третья нормальная форма** (3 НФ) ⎯ отношение находится в 3 НФ тогда и только тогда, когда оно находится во 2 НФ, и каждый не ключевой атрибут транзитивно зависит от первичного ключа ( устранение из табл. данных, которые не зависят от первичного ключа).

**Четвертая нормальная форма** (4 НФ) ⎯ отношение находится в 4 НФ, тогда и только тогда, когда в случае существования многозначной зависимости атрибута B от атрибута A все остальные атрибуты этого отношения функционально полно зависят от A (отсутствуют нетривиальные многозначные зависимости).

Для поддержания БД в устойчивом состоянии используется ряд механизмов, которые получили обобщенное название средств **поддержки целостности**. Приведение структуры БД в соответствие этим ограничениям - это и есть **нормализация**.

**Нормализация** – процесс, направленный на уменьшение избыточности информации в БД (разбиение информации на более детальные составляющие).

**15. Целостность данных.**

**Целостность данных** – правила, которые СУБД автоматически соблюдает при вводе и удалении значений в связанных полях таблиц. Обеспечение целостности данных можно включить либо отключить при создании связи в схеме данных. Правила целостности:

1. первичный ключ должен содержать уникальные непустые значения,

2. внешний ключ должен содержать только те значения, которые уже имеются среди значений первичного ключа.

ЦД – в каждый момент времени данные должны быть не противоречивы.

**16) История развития SQL. Язык SQL. Функции и основные возможности**

*SQL (Structured Query Language) — Структурированный Язык Запросов —*стандартный язык запросов по работе с реляционными БД. Язык SQL появился после реляционной алгебры, и его прототип был разработан в конце 70-х годов в компании IBM Research. Он был реализован в первом прототипе реляционной СУБД фирмы IBM System R. В дальнейшем этот язык применялся во многих коммерческих СУБД и в силу своего широкого распространения постепенно стал стандартом «де-факто» для языков манипулирования данными в реляционных СУБД.

Первый международный стандарт языка SQL был принят в 1989 г. (далее мы будем называть его SQL/89 или SQL1). Иногда стандарт SQL1 также называют стандартом ANSI/ISO, и подавляющее большинство доступных на рынке СУБД поддерживают этот стандарт полностью. Однако развитие информационных технологий, связанных с базами данных, и необходимость реализации переносимых приложений потребовали в скором времени доработки и расширения первого стандарта SQL.

В конце 1992 г. был принят новый международный стандарт языка SQL, который в дальнейшим будем называть SQL/92 или SQL2. И он не лишен недостатков, но в то же время является существенно более точным и полным, чем SQL/89. В настоящий момент большинство производителей СУБД внесли изменения в свои продукты так, чтобы они в большей степени удовлетворяли стандарту SQL2.

В 1999 году появился новый стандарт, названный SQL3. Если отличия между стандартами SQL1 и SQL2 во многом были количественными, то стандарт SQL3 соответствует качественным серьезным преобразованиям. В SQL3 введены новые типы данных, при этом предполагается возможность задания сложных структурированных типов данных, которые в большей степени соответствуют объектной ориентации. Наконец, добавлен раздел, который вводит стандарты на события и триггеры, которые ранее не затрагивались в стандартах, хотя давно уже широко использовались в коммерческих СУБД. В стандарте определены возможности четкой спецификации триггеров как совокупности события и действия. В качестве действия могут выступать не только последовательность операторов SQL, но и операторы управления ходом выполнения программы. В рамках управления транзакциями произошел возврат к старой модели транзакций, допускающей *точки сохранения (savepoints).*Возможность указания в операторе отката ROOLBACK точек возврата позволит откатывать транзакцию не в начало, а в промежуточную ранее сохраненную точку. Такое решение повышает гибкость реализации сложных алгоритмов обработки информации.

**История версий SQL**

|  |  |  |  |
| --- | --- | --- | --- |
| Год | Название | Иное название | Изменения |
| 1986 | SQL-86 | SQL-87 | Первый вариант стандарта, принятый институтом ANSI и одобренный ISO в 1987 году. |
| 1989 | SQL-89 | FIPS 127-1 | Немного доработанный вариант предыдущего стандарта. |
| 1992 | SQL-92 | SQL2, FIPS 127-2 | Значительные изменения (ISO 9075); уровень Entry Level стандарта SQL-92 был принят как стандарт FIPS 127-2. |
| 1999 | SQL:1999 | SQL3 | Добавлена поддержка регулярных выражений, рекурсивных запросов, поддержка триггеров, базовые процедурные расширения, нескалярные типы данных и некоторые объектно-ориентированные возможности. |
| 2003 | SQL:2003 |  | Введены расширения для работы с XML-данными, оконные функции (применяемые для работы с OLAP-базами данных), генераторы последовательностей и основанные на них типы данных. |
| 2006 | SQL:2006 |  | Функциональность работы с XML-данными значительно расширена. Появилась возможность совместно использовать в запросах SQL и XQuery. |
| 2008 | SQL:2008 |  | Улучшены возможности оконных функций, устранены некоторые неоднозначности стандарта SQL:2003 |

Язык SQL. Функции и основные возможности

1.Возможность в запросе группировать кортежи отношений результата по указанным полям в соответствии с условиями выборки на всю группу целиком.

2.В SQL не обязательно удалять кортежи-дубликатыкак в результате, так и в промежуточном кортеже. Результатом выполнения запроса SQL является мультимножественный кортеж. В СУБД System R были реализованы все теоретикомножественные операторы.

**17) Выражения в sql**

**cтроковые операции**

**||** - соединение строк, в некоторых СУБД операнды автоматически преобразуются в строковый тип. В MS Access используется &.

select 'hello'||' world'

select 'hello'||' world' from dual -- для Oracle

**алгебраические операции**

* **+** - сложение;
* **-** - вычитание;
* **\*** - умножение;
* **/** - деление;
* **mod** - остаток от деления. Oracle: mod(6,2). MySql: 6 mod 2.

Операции + и - также используются при работе со временем и интервалами. В Oracle и PostgreSQL возможна разница между датами. Результат возвращается в виде интервала в днях. Ниже приведен пример добавления к дате интервала.

-- для PostgreSQL

select date '2009-01-01'+INTERVAL '3 7:07:05'

-- для Oracle

select date '2009-01-01' +

INTERVAL '3 7:07:05' day to second

from dual;

-- для MySQL

select date '2009-01-01'+

INTERVAL '3 7:07:05' day\_second

Ко времени можно прибавлять целое число, но результат зависит от конкретной СУБД.

-- для Oracle, 1 интерпретируется как день

select date '2009-01-01'+1 from dual

-- для PostgreSQL, 1 интерпретируется как день

select date '2009-01-01'+1

-- для MySQL, 1 интерпретируется как год

select date '2009-01-01'+1

**операции отношения**

* **<** - меньше;
* **<=** - меньше либо равно;
* **>** - больше;
* **>=** - больше либо равно;
* **=** - равно;
* **<>**,**!=** - не равно;

**логические операции и предикаты**

* **and** - логическое и;
* **or** - логическое или;
* **nor** - отрицание;
* **between** - определяет, находится ли значение в указанном диапазоне:   
  выражение BETWEEN значение\_с AND значение\_по
* **exists** - определяет есть ли в указанной выборке хотя бы одна запись   
  EXISTS (select ...)   
  Для скорости в подзапросе обычно выбирают константу, а не поля записей, так как в данном случае нам важны не данные, а факт существования записей;
* **in** - определяет, входит ли указанное значение в указанное множество:   
  выражение IN (значение1,...,значениеn)   
  В качестве множества значений может служить корректная выборка   
  выражение IN (select ...)
* **is null** - является ли указанное выражение NULL значением:  
  выражение IS NULL
* **like** - определяет, удовлетворяет ли строка указанному шаблону:  
  строковое\_выражение LIKE шаблон [ESCAPE еск\_символ]  
  Знак % в шаблоне интерпретируется как строка любой длины, знак \_ как любой символ. В конструкции ESCAPE еск\_символ указывается символ ESCAPE последовательности, который отменит обычную интерпретацию символов '\_' и '%'. В последних стандартах включены предикаты SIMILAR и LIKE\_REGEX расширяющие возможности LIKE, используя в качестве шаблона регулярные выражения.

**условные выражения**

* ***case*** - условный оператор, имеющий следующий синтаксис:
* CASE WHEN условие THEN результат
* [WHEN условиеn THEN результатn]
* [ELSE результат\_по\_умолчанию]

END

* ***decode***(expr,s1,r1[,sn,rn][,defr]) - сравнивает выражение expr с каждым выражением si из списка. Если выражения равны то возвращается значение равное ri. Если ни одно из выражений в списке не равно expr, то возвращается defr или NULL, если defr не было указано. Эта функция доступна только в Oracle и в большинстве случае заменяет оператор CASE;
* ***coalesce***(arg1,...,argn) - возвращает первый аргумент в списке не равный null. Для двух аргументов в Oracle можно воспользоваться функцией nvl;
* ***greatest***(arg1,...,argn) - возвращает наибольший аргумент в списке;
* ***least***(arg1,...,argn) - возвращает наименьший аргумент в списке;
* ***nullif***((arg1,arg2) - возвращает null если два аргумента равны, иначе первый аргумент.

Ниже приведен пример использования выражения в запросе выбора данных.

-- для MySQL, PostresSQL

-- в скобках наше выражение

select ('молоко' LIKE '%оло%') as result;

-- эмулировать логический тип в запросах данных

-- для Oracle можно с помощью CASE

select case

-- в скобках наше условие

when (2 BETWEEN 0 AND 3 )

then 1

else 0

end as result from dual;

**прочие операции**

В каждой СУБД свой набор операций, выше были приведены наиболее употребительные. Например, в PosgreSQL можно использовать и такие операции:

* **^** - возведение в степень;
* **|/** - квадратный корень;
* **||/** - кубический корень;
* **!** - постфиксный факториал;
* **!!** - префиксный факториал;
* **@** - абсолютное значение.

**18) СУБД в архитектуре клиент-сервер. Открытые системы.**

***Системная архитектура клиент-сервер***

В общем случае, чтобы прикладная программа, выполняющаяся на рабочей станции, могла запросить услугу некоторого сервера, требуется, как минимум, некоторый интерфейсный программный слой, поддерживающий такого рода взаимодействия. Система разбивается на две части, которые могут выполняться в разных узлах сети, т.е. клиентскую и серверную части. Прикладная программа и конечный пользователь взаимодействуют с клиентской частью системы, которую в простейшем случае обеспечивает просто надсетевой интерфейс.

Клиентская часть системы по мере необходимости обращается по сети к серверной части. В развитых системах сетевое обращение серверной части может и не понадобиться при условии, что система может предугадывать потребности пользователя и в клиентской части содержаться данные, которые необходимы для выполнения запроса пользователя. Интерфейс серверной части определен и фиксирован, поэтому возможно создание новых клиентских частей существующей системы. Основной проблемой систем, основанных на архитектуре клиент-сервер, является то, что в соответствии с «Концепцией открытых систем» от них требуется мобильность в как можно более широком классе аппаратно-программных решений. Попытки создания систем, поддерживающих все возможные протоколы, приводят к перегрузке систем сетевыми деталями в ущерб их функциональности. Еще более сложный аспект этой проблемы связан с возможностью использования разных представлений данных в разных узлах неоднородной локальной сети. На разных компьютерах может существовать различная адресация, представление чисел, кодировка символов и т.д. Это особенно существенно для серверов высокого уровня, в том числе и серверов баз данных.

Общим решение проблемы мобильности систем, основанным на архитектуре клиент-сервер, является опора на программные пакеты, реализующие удаленный вызов процедур (Remote Procedure Call).

При использовании таких средств обращение к сервису в удаленном узле выглядит как обычный вызов процедуры. Средства RPC, в которых содержится вся информация от специфики аппаратуры локальной сети и сетевых протоколов, переводят вызов процедур в последовательность сетевых взаимодействий, тем самым специфики сетевой среды и протоколов от программиста, занимающегося разработкой прикладных программ.

При вызове удаленной процедуры средства RPC производят преобразования форматов данных клиента в промежуточные машинонезависимые форматы, а затем в форматы данных серверов. При передаче конкретных параметров производятся конкретные преобразования. Если система реализована на основе стандартного пакета RPC,она может быть перенесена в любую открытую среду.

***Открытая система***

Реальное распространение архитектуры клиент-сервер стало возможным благодаря развитию и широкому внедрению в практику «Концепции открытых систем». Основным смыслом подхода открытых систем является упрощение комплексирования вычислительных систем за счет международной и национальной стандартизаций аппаратных и программных интерфейсов. Главной побудительной причиной развития «Концепции открытых систем» явился повсеместный переход к использованию локальных вычислительных сетей и те проблемы комплексирования аппаратно-программных средств, которые этот переход вызвали. В связи с бурным развитием технологий глобальных коммуникаций открытые системы приобрели еще больший масштаб.

Основой открытых систем, дающей существенные преимущества пользователям, является независимость от конкретного поставщика. Ориентируясь на продукцию компаний, придерживающихся стандартов открытых систем, потребитель, который приобретает продукт компании, не попадает в зависимость от ее продуктов, он может продолжить наращивание мощности своей системы за счет приобретения продуктов любой другой компании, соблюдающей стандарты открытых систем, причем это касается как аппаратных, так и программных средств. Технологии стандарта открытых систем обеспечивают реальную и проверенную практикой возможность производства системных и прикладных программных средств со свойствами мобильности и интероперабильности. Свойство мобильности означает сравнительную простоту переноса программной системы в широком спектре аппаратно-программных средств, соответствующих стандартам открытых систем. Интероперабильность означает упрощение комплексирования новых программных систем на основе использования готовых компонентов со стандартными интерфейсами.

Использование подхода открытых систем выгодно и производителям программного обеспечения, и пользователям. Открытые системы обеспечивают естественное решение проблемы смены поколений аппаратных и программных средств. Производителям аппаратных и программных средств нет необходимости решать все проблемы заново, они могут, по крайней мере, некоторое время продолжать комплексировать системы, используя существующие компоненты. При этом возникает новый уровень конкуренции: все производители аппаратного и программного обеспечения обязаны обеспечить некоторую стандартную среду и при этом вынуждены добиваться как можно лучшей ее реализации. Преимуществом для пользователей является то, что они могут постепенно заменять компоненты системы на более совершенные, и при этом система не утрачивает своей работоспособности. В этом кроется решение проблемы постепенного наращивания вычислительных, информационных и других мощностей компьютерной системы.

**19) Команда SELECT**

Все запросы в SQL состоят из одиночной команды. Структура этой команды обманчиво проста, потому что вы должны расширять ее так, чтобы выполнить высоко сложные оценки и обработки данных. Эта команда называется —SELECT(ВЫБОР).

В самой простой форме, команда SELECT просто инструктирует базу данных, чтобы извлечь информацию из таблицы. Например, вы могли бы вывести таблицу Продавцов, напечатав следующее:

**SELECT snum, sname, city, comm FROM Salespeople;**

Вывод для этого запроса показывается в Рисунке 3.1.

=============== SQL Execution Log ============ | SELECT snum, sname, city, comm | | FROM Salespeople; | | ==============================================| | snum sname city comm | | ------ ---------- ----------- ------- | | 1001 Peel London 0.12 | | 1002 Serres San Jose 0.13 | | 1004 Motika London 0.11 | | 1007 Rifkin Barcelona 0.15 | | 1003 Axelrod New York 0.10 | =============================================== Рисунок 3.1. команда SELECT

Другими словами, эта команда просто выводит все данные из таблицы. Большинство программ будут также давать заголовки столбца как выше, а некоторые позволяют детальное форматирование вывода, но это уже вне стандартной спецификации.

**20) Команда SELECT. Раздел FROM.**

В простейшем случае применение команды *SELECT* выглядит так:

SELECT \*

FROM Table\_Name;

Звездочка указывает, что нужно показать все поля. Вместо звездочки можно указать конкретное *поле* или поля, разделяя их запятыми.

*Команда* ***FROM*** определяет имена таблиц, из которых осуществляется *выборка данных*. Если таблиц несколько, их имена разделяются запятыми. Иногда таблицы имеют длинные имена. В этом случае бывает выгодно использовать *псевдонимы* (alias) имен таблиц, указывая их через *пробел* после имени таблицы:

SELECT Field1, f.Field2, s.Field2

FROM Table1 f, Table2 s;

**Оператор выборки** — это отдельный оператор **языка SQL**, позволяющий получить результат запроса в прикладной программе без привлечения курсора. Поэтому оператор выборки имеет синтаксис, отличающийся от синтаксиса спецификации курсора, и при его выполнении возникают ограничения на результат табличного выражения. Фактически, и то, и другое диктуется спецификой оператора выборки как одиночного оператора **SQL.**

**21) Команда SELECT. Разделы WHERE, GROUP BY, HAVING, ORDER BY.**

### Команда WHERE

*Команда* ***WHERE*** позволяет использовать условие, которые может быть верным или нет для каждой записи *БД*. Если условие верное, то *запись* добавляется в набор данных, иначе отвергается. Давайте рассмотрим пример. Загрузите *SQL* -монитор из прошлой лекции. Предположим, нам нужно получить следующие данные на каждого сотрудника: Фамилия, Имя, Отдел, Должность. Пишем соответственный *SQL* -*запрос*:

SELECT Фамилия, Имя, Отдел, Должность

FROM LichData, Doljnost;

Выполнив этот *запрос*, вы получите нечто непонятное. В полученном наборе данных всем сотрудникам подряд присваивается вначале первая должность, затем вторая, и так до конца. Другими словами, если у вас 10 сотрудников и 10 должностей, то вместо ожидаемых десяти записей вы получите 10 \* 10 = 100 записей! Полученные данные называют *недостоверными*. Чтобы избежать этого, существует *команда* *WHERE*, которая позволяет задать условие выборки данных:

SELECT Фамилия, Имя, Отдел, Должность

FROM LichData, Doljnost

WHERE Ключ = Сотрудник;

Теперь все в порядке, отдел и должность соответствуют каждому сотруднику. Мы указали, что нам нужны лишь те записи, значения которых в *поле* "*Ключ*" одной таблицы соответствуют значениям в *поле* "Сотрудник" другой таблицы. Полный *синтаксис* требует указания таблицы вместе с полем:

WHERE LichData.Ключ = Doljnost.Сотрудник;

Однако поскольку у нас в этих таблицах нет полей с похожими именами, мы можем воспользоваться упрощенным вариантом. И в том, и в другом случае мы получим одинаковый набор данных.

Как и в любом условии, здесь можно применять различные *операторы* сравнения:

|  |  |
| --- | --- |
| Таблица 8.2 . Операторы сравнения | |
| **Оператор** | **Описание** |
| = | Равно |
| > | Больше |
| < | Меньше |
| >= | Больше или равно |
| <= | Меньше или равно |
| <> | Не равно |

Кроме того, мы можем использовать логические *операторы* *AND*, *OR* и *NOT*, формируя более сложные запросы:

SELECT Фамилия, Имя, Отдел, Должность

FROM LichData, Doljnost

WHERE (LichData.Ключ = Doljnost.Сотрудник)

AND (Должность = "Бухгалтер");

Логические *операторы* имеют более высокий приоритет, поэтому в приведенном примере можно обойтись и без скобок. Данный *запрос* выдаст нам данные только на бухгалтеров. Как вы могли заметить, в отличие от Delphi, строка в *SQL*заключается не в одинарные, а в двойные кавычки! Однако *SQL* более демократичен, одинарные кавычки тоже принимаются. Обычно их используют, если внутри строки требуется указать кавычки, например, 'Строка "в кавычках" будет отображена'.

Еще следует заметить, что подобное *связывание* таблиц не требует наличия индексных полей. Но если такие поля есть, *механизм доступа к данным* будет их использовать для более эффективной выборки данных.

### Команда ORDER BY

*Команда* ***ORDER BY*** позволяет сортировать записи по определенному полю как в возрастающем, так и в убывающем порядке. Воспользуемся предыдущим примером, и отсортируем записи по полю "Фамилия":

SELECT Фамилия, Имя, Отдел, Должность

FROM LichData, Doljnost

WHERE Ключ = Сотрудник

ORDER BY Фамилия;

Как уже говорилось, мы можем сортировать данные как по возрастанию ( ***ASC*** ), так и по убыванию ( ***DESC*** ) значений. *Сортировка* по возрастанию установлена "по умолчанию", а вот чтобы сортировать записи по убыванию, после имени поля следует поставить служебное *слово* *DESC*:

ORDER BY Фамилия DESC;

Опять заметим, что для сортировки записей наличие индексных полей необязательно. Часто бывает, что нужно вывести данные из двух таблиц, имеющих *связь* *один-ко-многим*. При этом нужно сортировать данные не по одному, а по двум полям:

SELECT Фамилия, Имя, Телефон, Примечание

FROM LichData, Telephones

WHERE Ключ = Сотрудник

ORDER BY Фамилия, Телефон;

В этом случае мы получим набор данных, в котором записи отсортированы вначале по фамилии сотрудника, затем по его номеру телефона:

Двойная сортировка данных

**Рис. 8.1 .**Двойная сортировка данных

### Команда HAVING

*Команда* ***HAVING*** позволяет определить условия, чтобы удалить определенные группы из полученного набора данных, точно так же, как *команда* *WHERE* делает это для отдельных записей. Предположим, нам нужно получить максимальный стаж работы по каждой должности, как это мы делали выше, но при этом указать, что этот максимальный стаж должен быть более 7 лет. Следовательно, если на какой-то должности работают молодые сотрудники, имеющие меньший стаж работы, эта должность не будет приниматься. Для задания условия мы обычно используем команду *WHERE*, но в этой команде нельзя использовать *агрегатные функции*, формирующие *значение* из *группы записей*. Другими словами, *запрос*, подобный этому:

SELECT Отдел, Должность, MAX(Стаж)

FROM LichData, Doljnost

WHERE Ключ = Сотрудник AND (MAX(Стаж) > 7)

GROUP BY Отдел, Должность;

вызовет ошибку. Использование в запросе команды *HAVING* решает эту проблему:

SELECT Отдел, Должность, MAX(Стаж)

FROM LichData, Doljnost

WHERE Ключ = Сотрудник

GROUP BY Отдел, Должность

HAVING MAX(Стаж) > 7;

Мы изучили все основные команды *SQL* -запросов. На основе этих команд можно создавать запросы любой сложности. Однако еще раз заметим, что хорошие знания языка *SQL*позволят вам создавать более гибкие и мощные приложения, так что не ленитесь, изучайте *SQL*. Ведь вам наверняка придется создавать клиент-серверные *базы данных*, а в этой архитектуре вся работа с данными осуществляется только посредством *SQL*.

### Команда GROUP BY

*Команда* ***GROUP BY*** позволяет группировать записи по какому-то определенному значению, и применяется совместно с агрегатными функциями. Предположим, нам требуется не просто получить средний стаж всех сотрудников, а еще и разбить эти данные по отделам. Вдруг директору приспичит узнать, в каком отделе у него работает больше всего молодых или старых специалистов! В нашу задачу входит выявить средний стаж сотрудников для каждого имеющегося отдела. Это мы можем выявить таким запросом:

SELECT AVG(Стаж), Отдел

FROM LichData, Doljnost

WHERE Ключ = Сотрудник

GROUP BY Отдел;

Как видите, *команда* *GROUP BY* используется после команды *WHERE* и группирует записи по значению поля "Отдел". В результате получим таблицу из двух полей. Первое *поле* будет сформировано агрегатной функцией, второе *поле* "Отдел" из таблицы Doljnost. Для каждого отдела будет рассчитано среднее *значение* поля "Стаж".

*Команда* ***GROUP BY*** позволяет группировать записи не только по одному, но и по множеству полей. Предположим, одну и ту же должность могут иметь несколько человек (например, пять бухгалтеров в бухгалтерии). Нам требуется найти самого старого сотрудника не только по отделу, но и по занимаемой должности. Нас выручит следующий *запрос*:

SELECT Отдел, Должность, MAX(Стаж)

FROM LichData, Doljnost

WHERE Ключ = Сотрудник

GROUP BY Отдел, Должность;

В результате мы получим набор данных, сгруппированный не только по отделам, но и по должностям. Если должность занимает только один человек, то мы получим его стаж. Если несколько - то наибольший из них.

**22) Системы управления базами данных следующего поколения.**

В этом разделе очень кратко рассматриваются основные направления исследований и разработок в области так называемых постреляционных систем, т.е. систем, относящихся к следующему поколению (хотя термин "next-generation DBMS" зарезервирован для некоторого подкласса современных систем).

Хотя отнесение СУБД к тому или иному классу в настоящее время может быть выполнено только условно (например, иногда объектно-ориентированную СУБД O2 относят к системам следующего поколения), можно отметить три направления в области СУБД следующего поколения. Чтобы не изобретать названий, будем обозначать их именами наиболее характерных СУБД.

1. *Направление Postgres*. Основная характеристика: максимальное следование (насколько это возможно с учетом новых требований) известным принципам организации СУБД (если не считать коренной переделки системы управления внешней памятью).
2. *Направление Exodus/Genesis*. Основная характеристика: создание собственно не системы, а генератора систем, наиболее полно соответствующих потребностям приложений. Решение достигается путем создания наборов модулей со стандартизованными интерфейсами, причем идея распространяется вплоть до самых базисовых слоев системы.
3. *Направление Starburst*. Основная характеристика: достижение расширяемости системы и ее приспосабливаемости к нуждам конкретных приложений путем использования стандартного механизма управления правилами. По сути дела, система представляет собой некоторый интерпретатор системы правил и набор модулей-действий, вызываемых в соответствии с этими правилами. Можно изменять наборы правил (существует специальный язык задания правил) или изменять действия, подставляя другие модули с тем же интерфейсом.

В целом можно сказать, что СУБД следующего поколения - это прямые наследники реляционных систем. Тем не менее, различные направления систем третьего поколения стоит рассмотреть отдельно, поскольку они обладают некоторыми разными характеристиками.