МОДЕЛИ ДАННЫХ И СУБД

1. Общие представления о базах данных. Основные понятия теории баз данных.
2. Системы управления базами данных
3. Представление данных с помощью модели "сущность-связь".
4. Реляционная модель данных
5. Реляционная алгебра и реляционное исчисление
6. Теория нормальных форм.
7. Язык SQL
8. Вопросы практического программирования.
9. Транзакции, блокировки и многопользовательский доступ к данным.
10. Использования WWW - технологий для доступа к существующим базам данных

ОБЩИЕ ПРЕДСТАВЛЕНИЯ О БАЗАХ ДАННЫХ. ОСНОВНЫЕ ПОНЯТИЯ ТЕОРИИ БАЗ ДАННЫХ.

За время существования человечества было накоплено достаточно большое количество знаний (информации). Эти знания сохранялась в памяти людей, записях, книгах, рисунках, картинах и т.д., и, можно без преувеличения сказать, что именно этим знаниям стало возможным развитие цивилизации. Для того чтобы можно было воспользоваться необходимой информацией, необходимо было ее разыскать среди множества.

Рассмотрим, например, вопрос, связанный с поиском необходимой книги в библиотеке. Пусть нам известны автор и название книги. Если книги в библиотеке ни коим образом не упорядочены и их достаточно много, то разыскать необходимую книгу достаточно сложно - необходимо в предельном случае пересмотреть все книги для того, чтобы найти необходимую. Мы можем облегчить нашу задачу, если предварительно расположим книги по какому-нибудь правилу, - например, в алфавитном порядке по авторам книг. Тогда нашу книгу найти значительно проще.

Усложним несколько задачу - пусть теперь нам известно только название книги, а наша библиотека упорядочена по-прежнему только по авторам книг. В таком случае такое упорядоченное расположение книг нам не может помочь и мы по-прежнему, как и ранее будем вынуждены брать каждую книгу в руки пока не найдем необходимую.

Если переупорядочить нашу библиотеку, расположив книги в алфавитном порядке по названиям книг, мы сможем теперь решить нашу проблему, но останется неразрешенной задача поиска книги по автору.

Можно привести еще несколько примеров - например, у книги может быть несколько авторов; мы разыскиваем не книгу, а рассказ в книге (альманахе, сборнике рассказов различных авторов); нам неизвестно ни название, ни автор книги, а известна лишь тематика или издательство и т.д. Как решение этих, да и других, проблем была идея при поиске рассматривать не книгу, а карточку, на которой указан автор(ы) книги, название, издательство, аннотация, помимо этого указан УДК - тематический номер. В справочно-библиографическом

отделе библиотеке имеется несколько каталогов: алфавитный каталог по авторам, по названиям книг, по тематике и т.д.

На этом примере мы подошли к понятию

Информационная система - под этим термином понимается какая либо **система** (это слово выделено) хранения информации. Особенно важным в этом понятии является термин "система". Этот термин предполагает упорядоченное, систематизированное хранение информации, т.е. помимо самих сохраняемых данных существуют и специальные правила обработки и поиска этих данных. Нередко такие сохраняемые данные называют базами данных

Современные информационные системы, основаны на концепции интеграции данных, характеризуются огромными объемами хранимых данных, сложной организацией, необходимостью удовлетворять разнообразные требования многочисленных пользователей.

Информационная система – система, реализующая автоматизированный сбор, обработку и манипулирование данными и включающая технические средства обработки данных, программное обеспечение и соответствующий персонал.

Цель любой информационной системы – обработка данных об объектах реального мира. Основой информационной системы является база данных. В широком смысле слова база данных – это совокупность сведений о конкретных объектах реального мира в какой-нибудь предметной области. Под предметной областью принято понимать часть реального мира, подлежащего изучению для организации управления и в конечном счете автоматизации.

Создавая базу данных, пользователь стремится упорядочить информацию по различным признакам и быстро производить выборку с произвольным сочетанием признаков. Большое значение при этом приобретает структурирование данных.

Структурирование данных – это введение соглашений о способах представления данных.

Неструктурированными называют данные, записанные, например, в текстовом файле.

Пример неструктурированных данных, содержащих сведения о студентах (номер личного дела, фамилия, имя, отчество, год рождения):

Личное дело № 16493, Сергеев Петр Михайлович, дата рождения 1 января 1976г.;
Л/д № 16494. Петрова Анна Владимировна, дата рожд. 4 марта 1975г.; № личного дела 17665, д.р. 19.08.1977, Анохин Андрей Борисович.

Легко убедиться, что сложно организовать поиск необходимых данных, хранящихся в неструктурированном виде.

Структурированные данные:

№ личного дела	**Фамилия**	**Имя**	**Отчество**	**Дата рождения**
16493	Сергеев	Петр	Михайлович	1.01.76
16494	Петрова	Анна	Владимировна	4.03.75
17665	Анохин	Андрей	Борисович	19.08.77

Чтобы автоматизировать поиск и систематизировать эти данные, необходимо выработать определенные соглашения о способах представления данных, т.е. дату рождения нужно записывать одинаково для каждого студента, она должна иметь одинаковую длину и определенное место среди остальной информации. Эти же замечания справедливы и для остальных данных (номер личного дела, фамилия, имя, отчество).

Пользователями базы данных могут быть различные прикладные программы, программные комплексы, а так же специалисты предметной области, выступающие в роли потребителей или источников данных, называемые конечными пользователями.

В современной технологии баз данных предполагается, что создание базы данных, ее поддержка и обеспечение доступа пользователей к ней осуществляется централизированно с помощью специального программного инструментария – **системами управления базами данных.**

База данных (БД) – это поименованная совокупность данных, отражающая состояние объектов и их отношений в рассматриваемой предметной области.

Объектом называется элемент предметной области, информацию о котором мы сохраняем.

Объект может быть реальным (человек, изделие, город) или абстрактным (событие, счет покупателя, изучаемый курс). Так в области продажи автомобилей примерами объектов могут служить МОДЕЛЬ АВТОМОБИЛЯ, КЛИЕНТ, СЧЕТ.

Система управления базами данных (СУБД) – это комплекс программных и языковых средств, предназначенных для создания, ведения и совместного применения баз данных многими пользователями.

Централизованный характер управления данными в базе данных предполагает необходимость существования некоторого лица (группы лиц), на которое возлагаются функции администрирования данными, хранимыми в базе.

Эволюция и поколения информационных систем

Обращаясь к истории развития и совершенствования информационных систем и систем управления базами данных, можно условно выделить три основных этапа.

Начальный этап был связан с созданием первого поколения СУБД, опиравшихся на иерархическую и сетевую модели данных. По времени он совпал с периодом, когда на рынке вычислительной техники доминировали большие ЭВМ (mainframe), например, система IBM 360/370, которые в совокупности с СУБД первого поколения составили аппаратно-программную платформу больших информационных систем.

К сожалению, СУБД первого поколения были в подавляющем большинстве закрытыми системами: отсутствовал стандарт внешних интерфейсов, не обеспечивалась переносимость прикладных программ. Они не обладали средствами автоматизации программирования и имели массу других недостатков, оцениваемых с точки зрения сегодняшних требований к СУБД. Кроме того, они были очень дороги. В то же время, СУБД первого поколения оказались весьма долговечными: разработанное на их основе программное обеспечение используется и по нынешний день. Большие ЭВМ по-прежнему хранят огромные массивы актуальной информации.

С созданием реляционной модели данных был начат **второй этап** в эволюции СУБД. Простота и гибкость модели привлекли к ней внимание разработчиков и снискали ей множество сторонников. Несмотря на некоторые недостатки (у кого их нет?), реляционная модель данных стала доминирующей. Условно эту группу систем можно назвать "**вторым поколением СУБД**". Его характеризовали две основные особенности - реляционная модель данных и язык запросов SQL.

Представители второго поколения в настоящее время еще сохраняют определенную популярность среди производителей СУБД, в большинстве своем развившись в системы третьего поколения, к которому и относятся современные СУБД.

Для них характерны использование идей объектно-ориентированного подхода, управления распределенными базами данных, активного сервера БД, языков

программирования четвертого поколения, фрагментации и параллельной обработки запросов, технологии тиражирования данных, многопотоковой архитектуры и других революционных достижений в области обработки данных. СУБД третьего поколения - это сложные многофункциональные программные системы, функционирующие в открытой распределенной среде. Сегодня они уже доступны для использования в деловой сфере, то есть они выступают не просто в качестве технических и научных решений, но как завершенные продукты, предоставляющие разработчикам мощные средства управления данными и богатый инструментарий для создания прикладных программ и систем.

На сегодняшний день существует, по разным оценкам, порядка 200 - 800 различных СУБД. Часть из этих СУБД уже практически ушли в прошлое и более не используются, в ряде СУБД используются схожие языку программирования БД. Так, например СУБД dBASE, Cliper, FoxPro достаточно близки по используемым языкам.

Классификация баз данных

По технологии обработки данных базы данных подразделяются на централизованные и распределенные.

Централизованная база данных хранится в памяти одной вычислительной системы. Если эта вычислительная система является компонентом сети, возможен распределенный доступ к такой базе. Такой способ использования баз данных часто применяют в локальных сетях.

Распределенная база данных состоит из нескольких, возможно пересекающихся или даже дублирующих друг друга частей, хранимых в различных ЭВМ вычислительной сети. Работа с такой базой осуществляется с помощью системы управления распределенной базой данных (СУРБД).

По способу доступа к данным базы данных разделяются на базы данных с локальным доступом и базы данных с удаленным (сетевым) доступом.

Системы централизованных баз данных с сетевым доступом предполагают различные архитектуры подобных систем:

- файл-сервер
- клиент-сервер

Файл-сервер. Архитектура систем БД с сетевым доступом предполагает выделение одной из машин сети в качестве центральной (сервера файлов). На такой машине хранится совместно используемая централизованная БД. Все другие машины сети выполняют функции рабочих станций, с помощью которых поддерживается доступ пользователей системы к централизованной базе данных. Файлы базы данных в соответствии с пользовательскими запросами передаются на рабочие станции, где в основном производится обработка. При большой интенсивности доступа к одним и тем же данным производительность информационной системы падает. Пользователи могут создавать также на рабочих местах локальные базы данных, которые используются ими монопольно

Клиент-сервер. В этой концепции подразумевается, что помимо хранения централизованной базы данных центральная машина (сервер базы данных) должна обеспечивать выполнение основного объема обработки данных. Запрос на данные, выдаваемый клиентом (рабочей станцией), порождает поиск и извлечение данных на сервере. Извлеченные данные (но не файлы) транспортируются по сети от сервера к клиенту. Спецификой архитектуры клиент-сервер является использование языка запросов SQL.

Системы управления базами данных

Система управления базами данных (СУБД) является универсальным программным средством, предназначенным для создания и ведения (обслуживания) баз данных (БД) на внешних запоминающих устройствах, а также доступа к данным и их обработки.
СУБД поддерживают один из возможных типов *моделей данных* — ***сетевую, иерархическую*** или ***реляционную,*** которые являются одним из важнейших признаков классификации СУБД. СУБД обеспечивают многоцелевой характер использования базы данных, защиту и восстановление данных. Наличие развитых диалоговых средств и языка запросов высокого уровня делает СУБД удобным средством для конечного пользователя.

Основными средствами СУБД являются:
•средства задания (описания) структуры базы данных;
•средства конструирования экранных форм, предназначенных для ввода данных, просмотра н их обработки в диалоговом режиме;
•средства создания запросов для выборки данных при заданных условиях, а также выполнения операций по их обработке;
•средства создания отчетов из базы данных для вывода на печать результатов обработки в удобном для пользователя виде;
•языковые средства — макросы, встроенный алгоритмический язык (Dbase, Visual Basic или другой), язык запросов (QBE — Query By Example, SQL) и т. п., которые используются для реализации нестандартных алгоритмов обработки данных, а также процедур обработки событий в задачах пользователя;
•средства создания приложений пользователя (генераторы приложений, средства создания меню и панелей управления приложениями), позволяющие объединить различные операции работы с базой данных в единый технологический процесс.
База данных — это совокупность данных, организованных на машинном носителе средствами СУБД. В базе данных обеспечивается интеграция логически связанных данных при минимальном дублировании хранимых данных. БД включает дачные, отражающие некоторую логическую модель взаимосвязанных информационных объектов, представляющих конкретную предметную область. База данных организуется в соответствии с моделью н структурами данных, которые поддерживаются в СУБД.
СУБД в многопользовательских *системах.* База данных, как правило, содержит данные, необходимые многим пользователям. Получение одновременного доступа нескольких пользователей к общей базе данных возможно при установке СУБД в локальной сети персональных компьютеров и создании многопользовательской базы данных.
В сети СУБД следит за разграничением доступа **разных** пользователей к общей базе данных и обеспечивает защиту данных при одновременной работе пользователей с общими данными. Автоматически обеспечивается защита данных от одновременной их корректировки несколькими пользователями-клиентами. В сети с *файловым сервером* база данных может размещаться на сервере. При этом СУБД загружается и осуществляет обработку данных базы на рабочих станциях пользователей. Концепция файлового сервера в локальной сети обеспечивается рядом сетевых операционных систем. Наиболее популярными являются Microsoft Windows NT и NetWare Novell. В сети, поддерживающей концепцию «клиент-сервер», используется сервер баз данных, который располагается на мощной машине, выполняет обработку данных, размещенных на сервере, и отвечает за их целостность и сохранность. Для управления базой данных на сервере используется язык структурированных запросов SQL (Structured Queries Language). На рабочих станциях-клиентах работает СУБД-клиент. Пользователи могут взаимодействовать не только со своими локальными базами, но и с данными, расположенными на сервере. СУБД-клиент, в которой поддерживается SQL, в полном объеме может посылать на сервер запросы SQL, получать необходимые данные, а также посылать обновленные данные. При этом с общей базой данных могут работать СУБД

разного тина, установленные на рабочих станциях, если в них поддерживается SQL. Подключение из СУБД к серверам баз данных SQL может быть осуществлено с помощью драйверов ODBC. ODBC (Open Database Connectivity, открытый стандарт доступа к базам данных), поддерживает стандартный протокол для серверов баз данных SQL.

Тенденции развития и классификация СУБД

Исторический аспект развития СУБД для ПК

Наибольшую популярность среди настольных систем, функционирующих в среде DOS, завоевали реляционные СУБД Dbase (компания Ashton-Tate), Paradox (Borland), Rbase (Mierorim), FoxPro (Fox Software), Clipper 5.0 (Nantucket), db_VISTA (Raima) с сетевой моделью данных.

В течение продолжительного периода времени широко использовались СУБД, совместимые со стандартом Xbase. Однако доля Xbase на рынке настольных СУБД сокращается. СУБД Dbase, FoxBase, FoxPro являются представителями этого семейства. СУБД Dbase имеют простой командный язык манипулирования данными и пользовательский интерфейс типа меню, средства генерации отчетов и экранных форм. Эта СУБД отличается хорошим быстродействием при выполнении запросов в небольших базах данных. В большинстве реляционных СУБД этого поколения, работающих в среде DOS, программы на базовом языке выполняются в режиме интерпретации, то есть заранее не преобразовываются в машинный код, что снижает их производительность.

Система db_VISTA, с включающим универсальным языком С, поддерживает сетевую модель. Она пользовалась популярностью среди профессиональных программистов. Областью ее применения, в частности, являются банковские информационные системы. К сетевым СУБД относится также AdabasD, которая предназначена для создания больших баз данных и может работать на разных платформах (техническая и программная среда).

Реляционная СУБД Paradox (версии 3.5, 4.0, 5.0) появилась на рынке в 1985г. Она отличается от семейства Xbase-продуктов запросами по образцу (QBE), генератором приложений на основе объектного подхода, настраиваемым меню пользователя, диалоговыми средствами и автоматическим формированием макросов, в которых можно запомнить все отлаженные пользователем процессы. В Paradox используется, кроме языка запросов QBE, базовый язык программирования PAL (Paradox Application Language) — язык для разработки приложений. Paradox, как и семейство Xbase, хранит свои объекты (таблицы, формы, отчеты, макросы) в отдельных файлах и обладает достаточной гибкостью, что позволяет модифицировать базу данных без перезагрузки данных. Обеспечивается создание сложных форм для нормализованных таблиц, через которые можно однократно вводить данные с внемашинных документов. Создание форм, запросов, отчетов, макросов легко выполняет пользователь-непрограммист. Для выполнения запроса в Paradox достаточно заполнить бланки запроса, которые на экране отображаются структурой таблицы базы данных. К мощным реляционным СУБД профессионального класса относится PROGRESS (фирмы Progress Software Co, USA). Она имеет встроенный язык SQL и собственный язык 4GL, может работать на разнообразных программно-аппаратных платформах, поддерживает архитектуру клиент-сервер.

Тенденции и перспективы развития СУБД

Перспективы развития архитектур СУБД связаны с развитием концепции обработки нетрадиционных данных и их интеграции, обмена данными из разных СУБД, многопользовательской технологии в локальных сетях. С 1996 г. операционная система Windows стала стандартом для настольных ПК. Для использования преимуществ этой операционной системы необходим переход к использованию 32-разрядных СУБД нижнего уровня. Наиболее известными и популярными СУБД такого типа являются: Access (Microsoft),

Paradox 7 for Windows 95 and Windows NT (Borland) и Approach for Windows 95 (Lotus). Относительно простой в изучении и использовании является считается Approach for Windows 95, которая ориентирована на разработку несложных приложений. Более совершенными, обладающими мощным языком разработки приложений пользователя являются две первые из названных СУБД — Paradox и Access. К общим свойствам СУБД Approach, Paradox и Access относятся:

• графический многооконный интерфейс, позволяющий пользователю в диалоговом режиме создавать таблицы, формы, запросы, отчеты и макросы;

• специальные средства, автоматизирующие работу, — многочисленные мастера (Wizards) в Access, ассистенты (Assistants) в Approach и эксперты (Experts) в Paradox;

• возможность работы в локальном режиме или в режиме клиента на рабочей
станции (Windows NT 3.51, Novell NetWare 4.1);

• использование объектной технологии OLE2 для внедрения в базу данных разной природы (текстов, электронных таблиц, изображений и т. п.);

• наличие собственного языка программирования. Особенности СУБД Approach, Paradox, Access:

• в Approach, в отличие от Paradox и Access, не обеспечивается полная поддержка языка запросов SQL, что ограничивает ее возможности в многопользовательских системах только просмотром данных;

• в Access предусмотрена автоматическая генерация кода SQL при создании запроса пользователем;

• в Approach язык для разработки приложений Lotus Script уступает по интеграционным возможностям и удобству работы объектно-ориентированным языкам (в Paradox — ObjectPAL, в Access — Visual Basic);

• Visual Basic в Access является наиболее мощным языком программирования, который обладает свойством автономности от СУБД н переносимости в другие приложения Microsoft Office, обеспечивая хорошую интеграцию данных;

• в Access имеется Мастер анализа таблиц, с помощью которого можно выполнить нормализацию таблицы.

Перспективы развития объектного подхода

Одной из важнейших тенденций развития СУБД является разработка «универсальных» **СУБД,** способных интегрировать в базе традиционные и нетрадиционные данные — тексты, рисунки, звук н видео, страницы HTML и др. Это особенно актуально для Web. Имеются два подхода к построению таких СУБД: *объектно-реляционный* — совершенствование существующих реляционных СУБД и *объектный.*

Следует отметить, что современные реляционные СУБД уже способны интегрировать данные, однако нетрадиционные данные не доступны для внутренней обработки. Универсальные СУБД должны выполнять такую обработку. В таких системах не нужны разнородные программы, которыми сложно управлять. По пути создания объектно-реляционных СУБД пошли такие фирмы, как IBM, Informix и Oracle. В IBM разработана объектно-реляционная СУБД DB2 для ОС AIX и OS/2. На начальном этапе фирма Oracle выпустила реляционный продукт Oracle Universal Server, интегрирующий СУБД Oracle 7.3 и специализированные серверы (Web, пространственных данных, текстов, видеосообщений), поддерживающие данные в разных хранилищах. В объекто-реляционной Oracle 8 должны быть интегрированы реляционные и нетрадиционные типы данных. Informix создала объектно-реляционную СУБД Universal Server.

Корпорация Microsoft сделала ставку на объектно-ориентированный интерфейс OLE DB, который обеспечивает доступ к данным Microsoft SQL Server (реляционная СУБД).

Фирма Sybase ориентирована на использование специализированных серверов, а интеграцию данных намеревается проводить другими средствами, то есть идет по пути создания объектно-реляционной СУБД (Adaptive-Server).

СУБД с параллельной обработкой данных

Информационные хранилища на базе СУБД с параллельной обработкой рассчитаны на многопроцессорные системы. Такие СУБД разделяются по типу архитектуры — без разделения ресурсов и с совместным использованием дискового пространства. В первом случае за каждым из процессоров закреплены выделенные области памяти и диски, что дает хорошую скорость обработки. Во втором случае все процессоры делят между собой как оперативную память, так н место на диске. Примерами СУБД без разделения ресурсов являются: DB2 (IBM), Informix Online Dynamic (Informix), Navigation Server (Sybase). СУБД с совместным использованием памяти является AdabasD версия 6.1 (Software AG). В СУБД Oracle 7.2 обеспечивается лучшая переносимость на различные платформы. Следует заметить, что выбор СУБД целесообразно осуществлять не только по типу архитектуры и качеству внешнего интерфейса, но прежде всего исходя из функциональных возможностей. Важными критериями выбора являются способность обработки сложных запросов (и скорость обработки), возможность переноса между платформами. Хорошей скоростью обработки сложных запросов отличается СУБД DB2 (IBM), а также DSA (Informix).

Классификация современных СУБД

К важным признакам классфикации современных СУБД относятся:

• среда функционирования — класс компьютеров и операционных систем (платформа), на которых работает СУБД, в том числе разрядность операционной системы, па которую ориентирована СУБД (16- или 32-разрядные);

• тип поддерживаемой в СУБД модели данных — сетевая, иерархическая и реляционная;

• возможности встроенного языка СУБД, его переносимость в другие приложения (SQL, Visual Basic, ObjectPAL и т. п.);

• наличие развитых диалоговых средств конструирования (таблиц, форм, запросов, отчетов, макросов) и средств работы с базой данных;

• возможность работы с нетрадиционными данными в корпоративных сетях (страницы HTML, сообщения электронной почты, изображения, звуковые файлы, видеоклипы и т. п.);

• используемая концепция работы с нетрадиционными данными — объектно-реляционные, объектные;

• уровень использования — локальная (для настольных систем), архитектура клиент-сервер, с параллельной обработкой данных (многопроцессорная);

• использование объектной технологии OLE 2.0;

• возможности интеграции данных из разных СУБД;

• степень поддержки языка SQL и возможности работы с сервером баз данных (SQL-сервером);

• наличие средств отчуждаемых приложений, позволяющих не проводить полной инсталляции СУБД для тиражируемых приложений пользователя.

ПРЕДСТАВЛЕНИЕ ДАННЫХ С ПОМОЩЬЮ МОДЕЛИ "СУЩНОСТЬ-СВЯЗЬ".

Назначение модели.

Прежде, чем приступать к созданию системы автоматизированной обработки информации, разработчик должен сформировать понятия о предметах, фактах и событиях, которыми будет оперировать данная система. Для того, чтобы привести эти понятия к той или иной модели данных, необходимо заменить их информационными представлениями. Одним из наиболее удобных инструментов унифицированного представления данных, независимого от реализующего его программного обеспечения, является модель "сущность-связь" (entity - relationship model, ER - model).

Модель "сущность-связь" основывается на некой важной семантической информации о реальном мире и предназначена для *логического* представления данных. Она определяет значения данных в контексте их взаимосвязи с другими данными. Важным для нас является тот факт, что из **модели "сущность-связь"** могут быть порождены все существующие **модели данных** (иерархическая, сетевая, реляционная, объектная), поэтому она является наиболее общей.

Модель "сущность-связь" была предложена в 1976 г. Питером Пин-Шэн Ченом, русский перевод его статьи 'Модель "сущность-связь" - шаг к единому представлению данных' опубликован в журнале "СУБД" N 3 за 1995 г.

Элементы модели.

Любой фрагмент предметной области может быть представлен как *множество сущностей*, между которыми существует некоторое *множество связей*. Дадим определения:

Сущность (entity) - это объект, который может быть идентифицирован неким способом, отличающим его от других объектов. Примеры: *конкретный человек, предприятие, событие и т.д.*

Набор сущностей (entity set) - множество сущностей одного типа (обладающих одинаковыми свойствами). Примеры: *все люди, предприятия, праздники и т.д.* Наборы сущностей не обязательно должны быть непересекающимися. Например, сущность, принадлежащая к набору МУЖЧИНЫ, также принадлежит набору ЛЮДИ.

Сущность фактически представляет из себя множество **атрибутов**, которые описывают свойства всех членов данного набора сущностей.

Пример:
рассмотрим множество работников некого предприятия. Каждого из них можно описать с помощью характеристик *табельный номер, имя, возраст*. Поэтому, сущность СОТРУДНИК имеет атрибуты ТАБЕЛЬНЫЙ_НОМЕР, ИМЯ, ВОЗРАСТ. Используя нотацию языка Pascal этот факт можно представить как:

```
type employe = record
      number : string[6];
      name   : string[50];
      age    : integer;
end;
```

В дальнейшем для определения сущности и ее атрибутов будем использовать обозначение вида

СОТРУДНИК (ТАБЕЛЬНЫЙ_НОМЕР, ИМЯ, ВОЗРАСТ).

Например отделы,на которые подразделяется предприятие, и в которых работают сотрудники, можно описать как ОТДЕЛ(НОМЕР_ОТДЕЛА, НАИМЕНОВАНИЕ).

Множество значений (область определения) атрибута называется **доменом**. Например, для атрибута ВОЗРАСТ домен (назовем его ЧИСЛО_ЛЕТ) задается интервалом целых чисел больших нуля, поскольку людей с отрицательным возрастом не бывает.

В упомянутой статье П.Чена атрибут определяется как *функция, отображающая набор сущностей в набор значений или в декартово произведение наборов значений*. Так атрибут ВОЗРАСТ производит отображение в набор значений (домен) ЧИСЛО_ЛЕТ. Атрибут ИМЯ производит отображение в декартово произведение наборов значений ИМЯ, ФАМИЛИЯ и ОТЧЕСТВО.

Отсюда определяется **ключ сущности** - группа атрибутов, такая, что отображение набора сущностей в соответствующую группу наборов значений является взаимнооднозначным отображением. Другими словами: ключ сущности - это один или более атрибутов уникально определяющих данную сущность. В нашем примере ключем сущности СОТРУДНИК является атрибут ТАБЕЛЬНЫЙ_НОМЕР (конечно, только в том случае, если все табельные номера на предприятии уникальны).

Связь (relationship) - это ассоциация, установленная между несколькими сущностями. Примеры:

- поскольку каждый сотрудник работает в каком-либо отделе, между сущностями СОТРУДНИК и ОТДЕЛ существует связь "работает в" или ОТДЕЛ-РАБОТНИК;
- так как один из работников отдела является его руководителем, то между сущностями СОТРУДНИК и ОТДЕЛ имеется связь "руководит" или ОТДЕЛ-РУКОВОДИТЕЛЬ;
- могут существовать и связи между сущностями одного типа, например связь РОДИТЕЛЬ - ПОТОМОК между двумя сущностями ЧЕЛОВЕК;

(В скобках здесь следует отметить, что в методике проектирования данных есть своеобразное правило хорошего тона, согласно которому сущности обозначаются с помощью имен существительных, а связи - глагольными формами. Данное правило, однако, не является обязательным)

К сожалению, не существует общих правил определения, что считать сущностью, а что связью. В рассмотренном выше примере мы положили, что "руководит" - это связь. Однако, можно рассматривать сущность "руководитель", которая имеет связи "руководит" с сущностью "отдел" и "является" с сущностью "сотрудник".

Связь также может иметь атрибуты. Например, для связи ОТДЕЛ-РАБОТНИК можно задать атрибут СТАЖ_РАБОТЫ_В_ОТДЕЛЕ.

Роль сущности в связи - функция, которую выполняет сущность в данной связи. Например, в связи РОДИТЕЛЬ-ПОТОМОК сущности ЧЕЛОВЕК могут иметь роли "родитель" и "потомок". Указание ролей в модели "сущность-связь" не является обязательным и служит для уточнения семантики связи.

Набор связей (relationship set) - это отношение между *n* (причем *n* не меньше 2) сущностями, каждая из которых относится к некоторому набору сущностей.

Пример:

```
сущности              наборы сущностей
```

```
----------                ---------------
    e1     принадлежит        E1
    e2     принадлежит        E2
             .   .   .
    en     принадлежит        En


тогда [e1,e2,...,en] - набор связей R
```

Хотя, сторого говоря, понятия "связь" и "набор связей" различны (первая является элементом второго), их, тем не менее, очень часто смешивают. Поэтому, мы, не претендуя на академическую строгость, в дальнейшем также будем часто пользоваться терминами "связь" имея в виду "набор связей" и "сущность" имея в виду "набор сущностей".

В случае *n=2*, т.е. когда связь объединяет две сущности, она называется бинарной. Доказано, что *n*-арный набор связей (*n>2*) всегда можно заменить множеством бинарных, однако первые лучше отображают семантику предметной области.

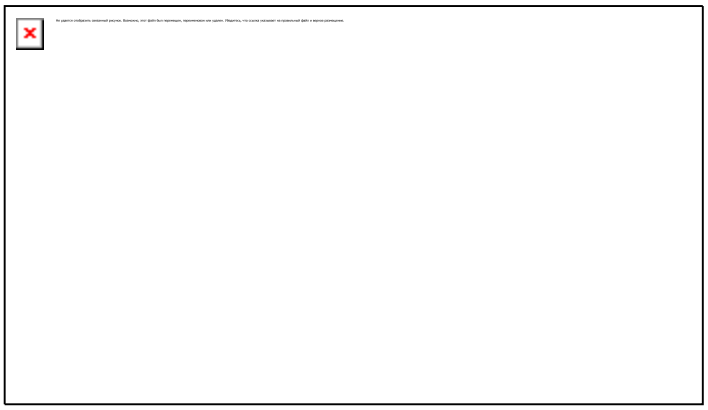
То число сущностей, которое может быть ассоциировано через набор связей с другой сущностью, называют **степенью связи**. Рассмотрение степеней особенно полезно для бинарных связей. Могут существовать следующие степени бинарных связей:

- один к одному (обозначается *1 : 1*). Это означает, что в такой связи сущности с одной ролью всегда соответствует не более одной сущности с другой ролью. В рассмотренном нами примере это связь "руководит", поскольку в каждом отделе может быть только один начальник, а сотрудник может руководить только в одном отделе. Данный факт представлен на следующем рисунке, где прямоугольники обозначают сущности, а ромб - связь. Так как степень связи для каждой сущности равна 1, то они соединяются одной линией.

  Другой важной характеристикой связи помимо ее степени является **класс принадлежности** входящих в нее сущностей или **кардинальность** связи. Так как в каждом отделе обязательно должен быть руководитель, то каждой сущности "ОТДЕЛ" непременно должна соответствовать сущность "СОТРУДНИК". Однако, не каждый сотрудник является руководителем отдела, следовательно в данной связи не каждая сущность "СОТРУДНИК" имеет ассоциированную с ней сущность "ОТДЕЛ".

  Таким образом, говорят, что сущность "СОТРУДНИК" имеет *обязательный класс принадлежности* (этот факт обозначается также указанием интервала числа возможных вхождений сущности в связь, в данном случае это 1,1), а сущность "ОТДЕЛ" имеет *необязательный класс принадлежности* (0,1). Теперь данную связь мы можем описать как *0,1:1,1*. В дальнейшем кардинальность бинарных связей степени 1 будем обозначать следующим образом:

- один ко многим (*1 : n*). В данном случае сущности с одной ролью может соответствовать любое число сущностей с другой ролью. Такова связь ОТДЕЛ-СОТРУДНИК. В каждом отделе может работать произвольное число сотрудников, но сотрудник может работать только в одном отделе. Графически степень связи *n* отображается "древообразной" линией, так это сделано на следующем рисунке.



Данный рисунок дополнительно иллюстрирует тот факт, что между двумя сущностями может быть определено несколько наборов связей.

Здесь также необходимо учитывать класс принадлежности сущностей. Каждый сотрудник должен работать в каком-либо отделе, но не каждый отдел (например, вновь сформированный) должен включать хотя бы одного сотрудника. Поэтому сущность "ОТДЕЛ" имеет обязательный, а сущность "СОТРУДНИК" необязательный классы принадлежности. Кардинальность бинарных связей степени *n* будем обозначать так:

- много к одному (*n : 1*). Эта связь аналогична отображению *1 : n*. Предположим, что рассматриваемое нами предприятие строит свою деятельность на основании контрактов, заключаемых с заказчиками. Этот факт отображается в модели "сущность-связь" с помощью связи КОНТРАКТ-ЗАКАЗЧИК, объединяющей сущности КОНТРАКТ(НОМЕР, СРОК_ИСПОЛНЕНИЯ, СУММА) и ЗАКАЗЧИК(НАИМЕНОВАНИЕ, АДРЕС). Так как с одним заказчиком может быть заключено более одного контракта, то связь КОНТРАКТ-ЗАКАЗЧИК между этими сущностями будет иметь степень *n : 1*.

В данном случае, по совершенно очевидным соображениям (каждый контракт заключен с конкретным заказчиком, а каждый заказчик имеет хотя бы один контракт, иначе он не был бы таковым), каждая сущность имеет обязательный класс принадлежности.

- многие ко многим (*n : n*). В этом случае каждая из ассоциированных сущностей может быть представлена любым количеством экземпляров. Пусть на рассматриваемом нами предприятии для выполнения каждого контракта создается рабочая группа, в которую входят сотрудники разных отделов. Поскольку каждый сотрудник может входить в несколько (в том числе и ни в одну) рабочих групп, а каждая группа должна включать не менее одного сотрудника, то связь между сущностями СОТРУДНИК и РАБОЧАЯ_ГРУППА имеет степень *n : n*.

Если существование сущности *x* зависит от существования сущности *y*, то *x* называется **зависимой сущностью** (иногда сущность *x* называют "слабой", а "сущность" *y - сильной). В качестве примера рассмотрим связь между ранее описанными сущностями РАБОЧАЯ_ГРУППА и КОНТРАКТ. Рабочая группа создается только после того, как будет подписан контракт с заказчиком, и прекращает свое существование по выполнению контракта. Таким образом, сущность РАБОЧАЯ_ГРУППА является зависимой от сущности КОНТРАКТ. Зависимую сущность будем обозначать двойным прямоугольником, а ее связь с сильной сущностью линией со стрелкой:*

Заметим, что кардинальность связи для сильной сущности всегда будет (1,1). Класс принадлежности и степень связи для зависимой сущности могут быть любыми. Предположим, например, что рассматриваемое нами предприятие пользуется несколькими банковскими кредитами, которые представляются набором сущностей КРЕДИТ(НОМЕР_ДОГОВОРА,СУММА, СРОК_ПОГАШЕНИЯ, БАНК). По каждому кредиту должны осуществляться выплаты процентов и платежи в счет его погашения. Этот факт представляется набором сущностей ПЛАТЕЖ(ДАТА, СУММА) и набором связей "осуществляется по". В том случае, когда получение запланированного кредита отменяется, информация о нем должна быть удалена из базы даных. Соответственно, должны быть удалены и все сведения о плановых платежах по этому кредиту. Таким образом, сущность ПЛАТЕЖ зависит от сущности КРЕДИТ.

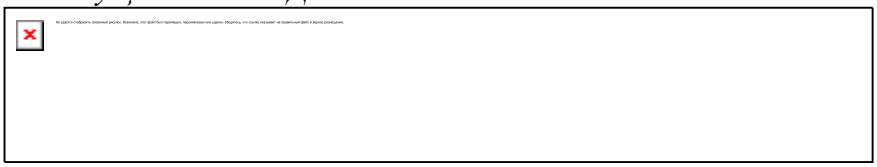
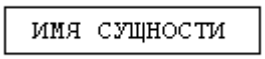
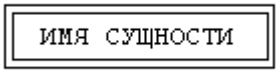
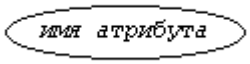
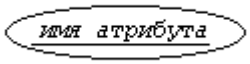


Диаграмма "сущность-связь".

Очень важным свойством модели "сущность-связь" является то, что она может быть представлена в виде графической схемы. Это значительно облегчает анализ предметной области. Существует несколько вариантов обозначения элементов диаграммы "сущность-связь", каждый из которых имеет свои положительные черты. Краткий обзор некоторых из этих нотаций будет сделан в параграфе 2.4. Здесь мы будем использовать некий гибрид нотаций Чена (обозначение сущностей, связей и атрибутов) и Мартина (обозначение степеней и кардинальностей связей). В таблице 2.1 приводится список используемых здесь обозначений.

Таблица 2.1

Обозначение	Значение
	Набор независимых сущностей
	Набор зависимых сущностей
	Атрибут
	Ключевой атрибут
	Набор связей

Атрибуты с сущностями и сущности со связями соединяются прямыми линиями. При этом для указания кардинальностей связей используются обозначения, введенные в предыдущем параграфе.

В процессе построения диаграммы можно выделить несколько очевидных этапов:

1. Идентификация представляющих интерес сущностей и связей.
2. Идентификация семантической информации в наборах связей (например, является ли некоторый набор связей отображением *1:n*).
3. Определение кардинальностей связей.
4. Определение атрибутов и наборов их значений (доменов).
5. Организация данных в виде отношений "сущность-связь".

В качестве примера построим диаграмму, отображающую связь данных для подсистемы учета персонала предприятия.

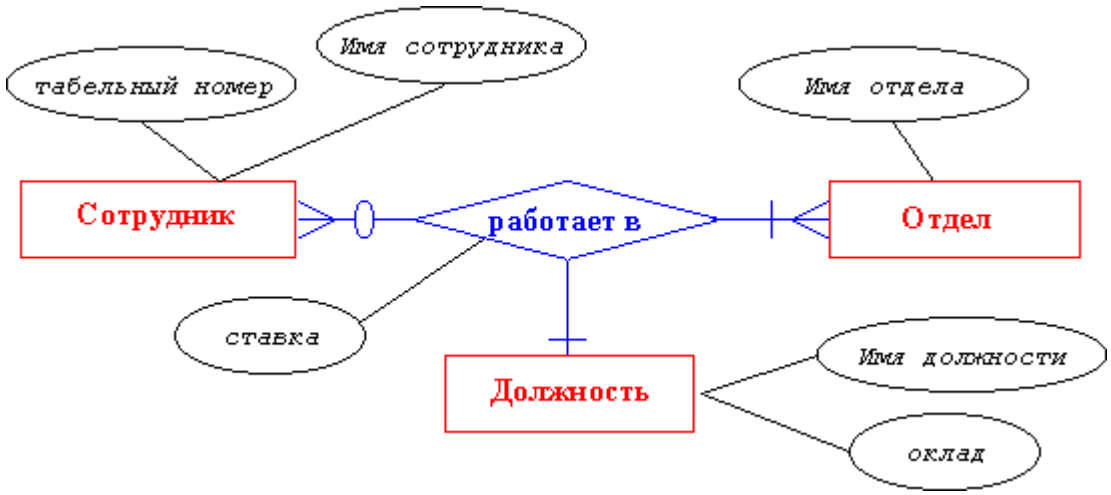
Выделим интересующие нас сущности и связи:

1. Прежде всего преприятие состоит из отделов, в которых работают сотрудники. Оклад каждого сотрудника зависит от занимаемой им должности (инженер, ведущий инженер, бухгалтер, уборщик и т.д.). Далее предположим, что на нашем предприятии допускается совместительство должностей, т.е. каждый сотрудник может иметь более чем одну должность (и работать более чем в одном отделе), причем может занимать неполную

ставку. В то же время, одну и ту же должность могут занимать одновременно несколько сотрудников. В результате этих рассуждений мы должны ввести наборы сущностей

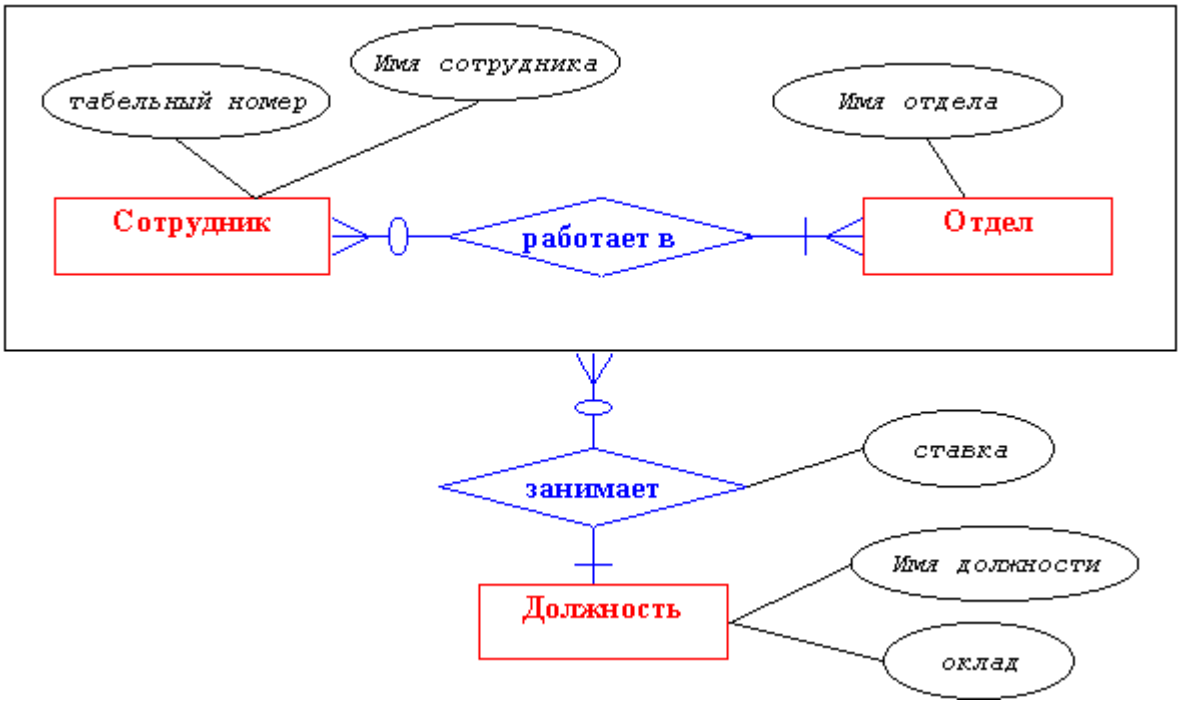
- ОТДЕЛ(ИМЯ_ОТДЕЛА),
- СОТРУДНИК(ТАБЕЛЬНЫЙ_НОМЕР, ИМЯ),
- ДОЛЖНОСТЬ(ИМЯ_ДОЛЖНОСТИ, ОКЛАД),

и и набор связей РАБОТАЕТ_В с атрибутом *ставка* между ними. Атрибут *ставка* может принимать значения из интервала `]0,1]` (больше нуля, но меньше или равен единице), он определяет какую часть должностного оклада получает данный сотрудник.



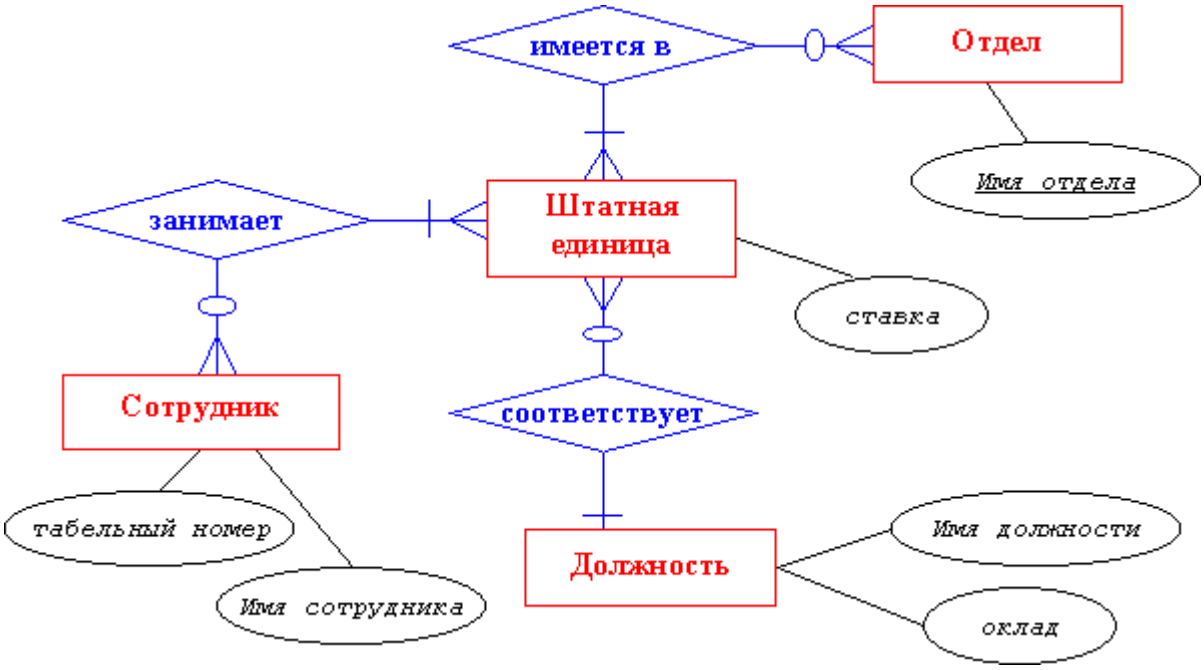
Как уже отмечалось выше, каждый *n*-арный набор связей можно заменить несколькими бинарными наборами. Сейчас как раз представляется удобный случай, чтобы оценить преимущества каждого из этих способов представления связей.

- Тренарная связь, показанная здесь, безусловно несет более полную информацию о предметной области. Действительно, она однозначно отображает тот факт, что оклад сотрудника зависит от его должности, отдела, где он работает, и ставки. Однако, в этом случае возникают некоторые проблемы с определением степени связи. Хотя, как было сказано, каждый работник может занимать несколько должностей, а в штате каждого отдела существуют вакансии с различными должностями, тем не менее класс принадлежности сущности ДОЛЖНОСТЬ на приведенном рисунке установлен в (1,1). Это объясняется тем, что ДОЛЖНОСТЬ ассоциируется фактически не с сущностями СОТРУДНИК и ОТДЕЛ, а со связью между ними. Обозначать этот факт предлагается так, как это показано на следующей диаграмме:



Здесь сущности СОТРУДНИК, ОТДЕЛ и связь РАБОТАЕТ_В аггрегируются в некую новую абстрактную сущность, которая ассоциируется с сущностью ДОЛЖНОСТЬ с помощью связи степени n:1. (Это обозначение заимствовано из книги Silberschatz, Korth and Sudarshan Database System Concepts, 1997).

- Попытаемя отобразить ассоциации сотрудников, отделов и должностей с помощью бинарных связей.



В этом случае для адекватного описания семантики предметной области

необходимо ввести еще одну сущность ШТАТНАЯ_ЕДИНИЦА, которая фактически заменяет собой связь РАБОТАЕТ_В в абстрактной сущности и поэтому имеет атрибут *ставка*.

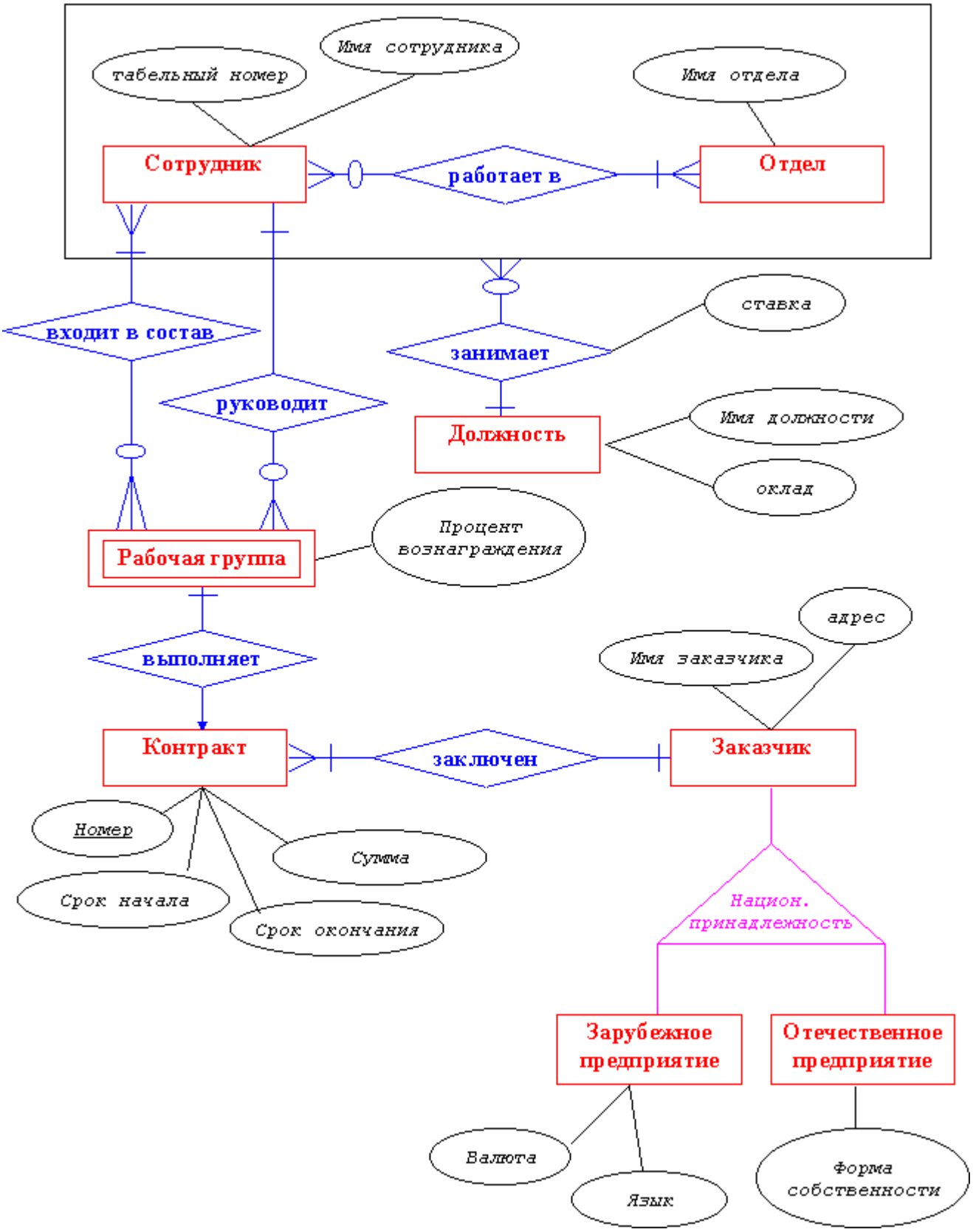
Переход от *n*-арной связи через аггрегацию сущностей к набору бинарных связей можно рассматривать как последовательные этапы одного процесса, который приводит к однозначному порождению реляционной модели данных. При построении диаграммы "сущность- связь" можно использовать любой из этих трех способов представления данных.

2. Перечисли ряд объектов, описанных в предыдущем праграфе, которые будут полезны при моделировании данных рассматриваемого предприятия. Им соответствуют следующие сущности:
   - ЗАКАЗЧИК(ИМЯ_ЗАКАЗЧИКА,АДРЕС)
   - КОНТРАКТ(НОМЕР,СРОК_НАЧАЛА,СРОК_ОКОНЧАНИЯ,СУММА)
   - РАБОЧАЯ ГРУППА(ПРОЦЕНТ_ВОЗНАГРАЖДЕНИЯ)

   Атрибут *"процент_вознаграждения"* отражает ту долю стоимости контракта, которая предназначена для оплаты труда членов соответствующей рабочей группы. Смысл остальных атрибутов понятен без дополнительных пояснений. Связи между перечисленными сущностями также описаны в предыдущем параграфе.
   Как правило, один из членов рабочей группы является руководителем по отношению к другим сотрудникам, входящим в ее состав. Для отражения этого факта мы должны ввести связь "руководит" с кардинальностью *1,1:0,n* между сущностями СОТРУДНИК и РАБОЧАЯ_ГРУППА (сотрудник может руководить в произвольном числе рабочих групп, но каждая рабочая группа имеет одного и только одного руководителя).

3. Рассмотрим теперь более внимательно информационный объект "заказчик". На практике очень часто возникает необходимость различать национальную прнадлежность юридических лиц, с которыми предприятие вступает в договорные отношения. Это свзано с тем, что для зарубежных фирм необходимо хранить, например, сведения о валюте, в которой осуществляются расчеты, языке, на котором подписан котракт и т.д. В свою очередь, для отечественных компаний необходимо иметь сведения о их форме собственности (частная или государственная), поскольку от этого может зависеть порядок налогообложения средств, полученных за выполнение работ по контракту. Таким образом, мы приходим к выводу, что необходимо ввести в рассмотрение еще два непересекающихся множества ЗАРУБЕЖНОЕ_ПРЕДПРИЯТИЕ(ВАЛЮТА, ЯЗЫК) и ОТЕЧЕСТВЕННОЕ_ПРЕДПРИЯТИЕ(ФОРМА_СОБСТВЕННОСТИ), объединение которых составляет полное множество ЗАКАЗЧИК. Ассоциацию между этими объектами называют **отношением наследования** или **иерархической связью**, так как сущности ЗАРУБЕЖНОЕ_ПРЕДПРИЯТИЕ и ОТЕЧЕСТВЕННОЕ_ПРЕДПРИЯТИЕ наследуют атрибуты сущности ЗАКАЗЧИК(ИМЯ_ЗАКАЗЧИКА, АДРЕС). Для того, чтобы определить к какому подмножеству относится конкретная сущность из набора ЗАКАЗЧИК (и, соответственно, какой набор атрибутов она имеет) необходимо ввести атрибут *"национальная принадлежность", называемый* ***дискриминантом***. *Этот тип связи предлагается отображать на диаграмме следующим образом:*

Обобщая все проведенные выше рассуждения, получим диаграму "сущность-связь", показанную на слудющем рисунке.


Имя сотрудника
табельный номер
Имя отдела
Сотрудник
работает в
Отдел
входит в состав
ставка
занимает
руководит
Должность
Имя должности
оклад
Процент вознаграждения
Рабочая группа
выполняет
адрес
Имя заказчика
Контракт
заключен
Заказчик
Номер
Сумма
Срок начала
Срок окончания
Национ. принадлежность
Зарубежное предприятие
Отечественное предприятие
Валюта
Язык
Форма собственности

РЕЛЯЦИОННАЯ МОДЕЛЬ ДАННЫХ

Реляционная модель предложена сотрудником компании IBM Е.Ф.Коддом в 1970 г. (русский перевод статьи, в которой она впервые описана опубликован в журнале "СУБД" N 1 за 1995 г.). В настоящее время эта модель является фактическим стандартом, на который ориентируются практически все современные коммерческие СУБД.

4.1.1.Структура данных

В реляционной модели достигается гораздо более высокий уровень абстракции данных, чем в иерархической или сетевой. В упомянутой статье Е.Ф.Кодда утверждается, что "*реляционная модель предоставляет средства описания данных на основе только их естественной структуры, т.е. без потребности введения какой-либо дополнительной структуры для целей машинного представления*". Другими словами, представление данных не зависит от способа их физической организации. Это обеспечивается за счет использования математической теории отношений (само название "реляционная" происходит от английского relation - "отношение"). Перейдем к рассмотрению структурной части реляционной модели данных. Прежде всего необходимо дать несколько определений.

Определения: Декартово произведение: Для заданных конечных множеств (не обязательно различных) декартовым произведением называется множество произведений вида: , где

Пример: *если даны два множества A (a1,a2,a3) и B (b1,b2), их декартово произведение будет иметь вид C=A*B (a1*b1, a2*b1, a3*b1, a1*b2, a2*b2, a3*b2)*

Отношение: Отношением *R*, определенным на множествах называется подмножество декартова произведения . При этом: множества называются **доменами** отношения элементы декартова произведения называются **кортежами**

- число *n* определяет **степень отношения** (*n=1* - унарное, *n=2* - бинарное, ..., *n*-арное)
- количество кортежей называется **мощностью отношения**

Пример: *на множестве C из предыдущего примера могут быть определены отношения R1 (a1*b1, a3*b2) или R2 (a1*b1, a2*b1, a1*b2)*

Отношения удобно представлять в виде таблиц. На рис. 4.1 представлена таблица (отношение степени 5), содержащая некоторые сведения о работниках гипотетического предприятия. Строки таблицы соответствуют кортежам. Каждая строка фактически представляет собой описание одного объекта реального мира (в данном случае работника), характеристики которого содержатся в столбцах. Можно провести аналогию между элементами реляционной модели данных и элементами модели "сущность-связь". Реляционные отношения соответствуют наборам сущностей, а кортежи - сущностям. Поэтому, также как и в модели "сущность-связь" столбцы в таблице, представляющей реляционное отношение, называют **атрибутами.**

Рис.4.1 Основные компоненты реляционного отношения.

Каждый атрибут определен на домене, поэтому домен можно рассматривать как множество допустимых значений данного атрибута.

Несколько атрибутов одного отношения и даже атрибуты разных отношений могут быть определены на одном и том же домене. В примере, показанном на рис.4.1 атрибуты "Оклад" и "Премия" определены на домене "Деньги". Поэтому, понятие домена имеет семантическую нагрузку: данные можно считать сравнимыми только тогда, когда они относятся к одному домену. Таким образом, в рассматриваемом нами примере сравнение атрибутов "Табельный номер" и "Оклад" является семантически некорректным, хотя они и содержат данные одного типа.

Именнованное множество пар "имя атрибута - имя домена" называется **схемой отношения.** Мощность этого множества - называют **степенью** или *"арностью"* отношения. Набор именованных схем отношений представляет из себя **схему базы данных.**

Атрибут, значение которого однозначно идентифицирует кортежи, называется **ключевым** (или просто **ключом).** В нашем случае ключом является атрибут "Табельный номер", поскольку его значение уникально для каждого работника предприятия. Если кортежи идентифицируются только сцеплением значений нескольких атрибутов, то говорят, что отношение имеет составной ключ.

Отношение может содержать несколько ключей. Всегда один из ключей объявляется **первичным**, его значения не могут обновляться. Все остальные ключи отношения называются *возможными ключами.*

В отличие от иерархической и сетевой моделей данных в реляционной отсутствует понятие группового отношения. Для отражения ассоциаций между кортежами разных отношений используется дублирование их ключей. Рассмотренный в параграфах 3.1 и 3.2 пример базы данных, содержащей сведения о подразделениях предприятия и работающих в них сотрудниках, применительно к реляционной модели будет иметь вид:

Рис.4.2. База данных о подразделениях и сотрудниках предприятия.

Например, связь между отношениями ОТДЕЛ и СОТРУДНИК создается путем копирования первичного ключа *"Номер_отдела"* из первого отношения во второе. Таким образом:
для того, чтобы получить список работников данного подразделения, необходимо

1. из таблицы ОТДЕЛ установить значение атрибута *"Номер_отдела"*, соответствующее данному "Наименованию_отдела"
2. выбрать из таблицы СОТРУДНИК все записи, значение атрибута *"Номер_отдела"* которых равно полученному на предыдущем шаге.

для того, чтобы узнать в каком отделе работает сотрудник, нужно выполнить обратную операцию:

1. определяем *"Номер_отдела"* из таблицы СОТРУДНИК
2. по полученному значению находим запись в таблице ОТДЕЛ.

Атрибуты, представляющие собой копии ключей других отношений, называются **внешними ключами**.

4.1.2.Свойства отношений.

1. Отсутствие кортежей-дубликатов. *Из этого свойства вытекает наличие у каждого кортежа первичного ключа. Для каждого отношения, по крайней мере, полный набор его атрибутов является первичным ключом. Однако, при определении первичного ключа должно соблюдаться требование "минимальности", т.е. в него не должны входить те атрибуты, которые можно отбросить без ущерба для основного свойства первичного ключа - однозначно определять кортеж.*
2. Отсутствие упорядоченности кортежей.
3. Отсутствие упордоченности атрибутов. Для ссылки на значение атрибута всегда используется имя атрибута.
4. Атомарность значений атрибутов, т.е. среди значений домена не могут содержаться множества значений (отношения).

Теория нормальных форм.

Функциональные зависимости.

Реляционная база данных содержит как структурную, так и семантическую информацию. Структура базы данных определяется числом и видом включенных в нее отношений, и связями типа "один ко многим", существующими между кортежами этих отношений. Семантическая часть описывает множество функциональных зависимостей, существующих между атрибутами этих отношений. Дадим определение функциональной зависимости.

Определение:

Если даны два атрибута *X* и *Y* некоторого отношения, то говорят, что *Y* функционально зависит от *X*, если в любой момент времени каждому значению *X* соответствует ровно одно значение *Y*.

Функциональная зависимость обозначается *X -> Y*. Отметим, что *X* и *Y* могут представлять собой не только единичные атрибуты, но и группы, составленные из нескольких атрибутов одного отношения.

Можно сказать, что функциональные зависимости представляют собой связи типа "один ко многим", существующие внутри отношения.

Некоторые функциональные зависимости могут быть нежелательны.

Определение:

Избыточная функциональная зависимость - зависимость, заключающая в себе такую информацию, которая может быть получена на основе других зависимостей, имеющихся в базе данных.

Корректной считается такая схема базы данных, в которой отсутствуют избыточные функциональные зависимости. В противном случае приходится прибегать к процедуре декомпозиции (разложения) имеющегося множества отношений. При этом порождаемое множество содержит большее число отношений, которые являются проекциями отношений исходного множества. (Операция проекции описана в разделе, посвященном реляционной алгебре). *Обратимый* пошаговый процесс замены данной совокупности отношений другой схемой с устранением избыточных функциональных зависимостей называется **нормализацией.**

Условие обратимости требует, чтобы декомпозиция сохраняла эквивалентность схем при замене одной схемы на другую, т.е. в результирующих отношениях:

- не должны появляться ранее отсутствовавшие кортежи;
- на отношениях новой схемы должно выполняться исходное множество функциональных зависимостей.

1NF - первая нормальная форма.

Для обсуждения первой нормальной формы необходимо дать два определения:

Простой атрибут - атрибут, значения которого атомарны (неделимы).

Сложный атрибут - получается соединением нескольких атомарных атрибутов, которые могут быть определены на одном или разных доменах. (его также называют вектор или агрегат данных).

Теперь можно дать

Определение первой нормальной формы:

отношение находится в 1NF если значения всех его атрибутов атомарны.

Рассмотрим пример, заимствованный из уже упоминавшейся статьи Е.Ф.Кодда:

В базе данных отдела кадров предприятия необходимо хранить сведения о служащих, которые можно попытаться представить в отношении

СЛУЖАЩИЙ(НОМЕР_СЛУЖАЩЕГО, ИМЯ, ДАТА_РОЖДЕНИЯ, ИСТОРИЯ_РАБОТЫ, ДЕТИ).

Из внимательного рассмотрения этого отношения следует, что атрибуты *"история_работы"* и *"дети"* являются сложными, более того, атрибут *"история_работы"* включает еще один сложный атрибут *"история_зарплаты"*. Данные агрегаты выглядят следующим образом: ИСТОРИЯ_РАБОТЫ (ДАТА_ПРИЕМА, НАЗВАНИЕ, ИСТОРИЯ_ЗАРПЛАТЫ), ИСТОРИЯ_ЗАРПЛАТЫ (ДАТА_НАЗНАЧЕНИЯ, ЗАРПЛАТА), ДЕТИ (ИМЯ_РЕБЕНКА, ГОД_РОЖДЕНИЯ).

Их связь представлена на рис. 4.3.

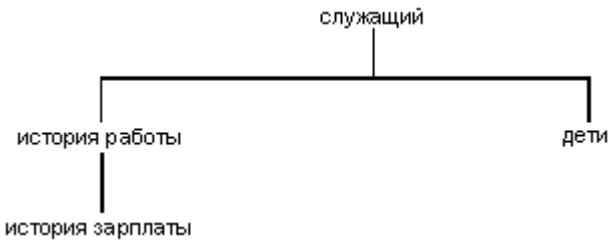


Рис.4.3. Исходное отношение.

Для приведения исходного отношения СЛУЖАЩИЙ к первой нормальной форме необходимо декомпозировать его на четыре отношения, так как это показано на следующем рисунке:

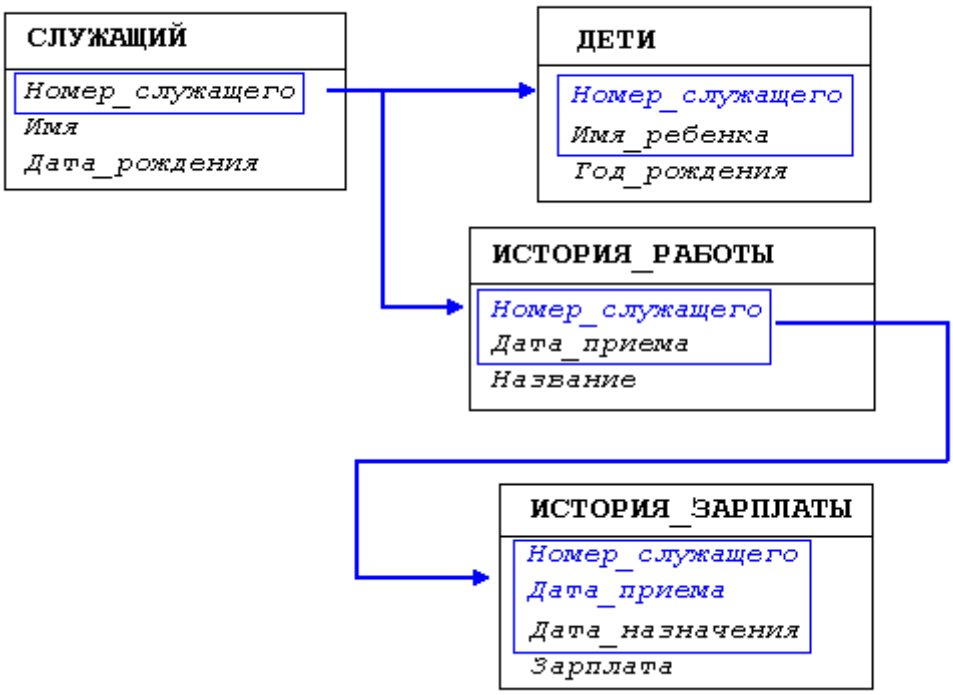


Рис.4.4. Нормализованное множество отношений.

Здесь первичный ключ каждого отношения выделен синей рамкой, названия внешних ключей набраны шрифтом синего цвета. Напомним, что именно внешние ключи служат для представления функциональных зависимостей, существующих в исходном отношении. Эти функциональные зависимости обозначены линиями со стрелками.

Алгоритм нормализации описан Е.Ф.Коддом следующим образом:

- Начиная с отношения, находящегося на верху дерева (рис. 4.3.), берется его первичный ключ, и каждое непосредственно подчиненное отношение расширяется путем вставки домена или комбинации доменов этого первичного ключа.
- Первичный Ключ каждого расширенного таким образом отношения состоит из Первичного Ключа, который был у этого отношения до расширения и добавленного Первичного Ключа родительского отношения.
- После этого из родительского отношения вычеркиваются все непростые домены, удаляется верхний узел дерева, и эта же процедура повторяется для каждого из оставшихся поддеревьев.

2NF - вторая нормальная форма.

Очень часто первичный ключ отношения включает несколько атрибутов (в таком случае его называют **составным**) - см., например, отношение ДЕТИ, показанное на рис. 4.4. При этом вводится понятие **полной функциональной зависимости.**

Определение:

неключевой атрибут функционально полно зависит от составного ключа если он функционально зависит от всего ключа в целом, но не находится в функциональной зависимости от какого-либо из входящих в него атрибутов.

Пример:

Пусть имеется отношение ПОСТАВКИ (N_ПОСТАВЩИКА, ТОВАР, ЦЕНА). Поставщик может поставлять различные товары, а один и тот же товар может поставляться разными поставщиками. Тогда ключ отношения - *"N_поставщика + товар"*. Пусть все поставщики поставляют товар по одной и той же цене. Тогда имеем следующие функциональные зависимости:

- *N_поставщика, товар -> цена*
- *товар -> цена*

Неполная функциональная зависимость атрибута "цена" от ключа приводит к следующей аномалии: при изменении цены товара необходим полный просмотр отношения для того, чтобы изменить все записи о его поставщиках. Данная аномалия является следствием того факта, что в одной структуре данных объединены два семантических факта. Следующее разложение дает отношения во 2НФ:

- ПОСТАВКИ (N_ПОСТАВЩИКА, ТОВАР)
- ЦЕНА_ТОВАРА (ТОВАР, ЦЕНА)

Таким образом, можно дать

Определение второй нормальной формы:

Отношение находится во 2НФ, если оно находится в 1НФ и каждый неключевой атрибут функционально полно зависит от ключа.

3NF - третья нормальная форма.

Перед обсуждением третьей нормальной формы необходимо ввести понятие **транзитивной функциональной зависимости.**

Определение:

Пусть X, Y, Z - три атрибута некоторого отношения. При этом X --> Y и Y --> Z, но обратное соответствие отсутствует, т.е. Z -/-> Y и Y -/-> X. Тогда Z транзитивно зависит от X.

Пусть имеется отношение ХРАНЕНИЕ (ФИРМА, СКЛАД, ОБЪЕМ), которое содержит информацию о фирмах, получающих товары со складов, и объемах этих складов. Ключевой атрибут - *"фирма"*. Если каждая фирма может получать товар только с одного склада, то в данном отношении имеются следующие функциональные зависимости:

- *фирма -> склад*
- *склад -> объем*

При этом возникают аномалии:

- если в данный момент ни одна фирма не получает товар со склада, то в базу данных нельзя ввести данные о его объеме (т.к. не определен ключевой атрибут)
- если объем склада изменяется, необходим просмотр всего отношения и изменение кортежей для всех фирм, связанных с данным складом.

Для устранения этих аномалий необходимо декомпозировать исходное отношение на два:

- ХРАНЕНИЕ (ФИРМА, СКЛАД)
- ОБЪЕМ_СКЛАДА (СКЛАД, ОБЪЕМ)

Определение третьей нормальной формы:

Отношение находится в 3НФ, если оно находится во 2НФ и каждый неключевой атрибут нетранзитивно зависит от первичного ключа.

BCNF - нормальная форма Бойса-Кодда.

Эта нормальная форма вводит дополнительное ограничение по сравнению с 3НФ.

Определение нормальной формы Бойса-Кодда:

Отношение находится в BCNF, если оно находится во 3НФ и в ней отсутствуют зависимости атрибутов первичного ключа от неключевых атрибутов.

Ситуация, когда отношение будет находится в 3NF, но не в BCNF, возникает при условии, что отношение имеет два (или более) возможных ключа, которые являются составными и имеют общий атрибут. Заметим, что на практике такая ситуация встречается достаточно редко, для всех прочих отношений 3NF и BCNF эквивалентны.

Многозначные зависимости и четвертая нормальная форма (4NF).

Четвертая нормальная форма касается отношений, в которых имеются повторяющиеся наборы данных. Декомпозиция, основанная на функциональных зависимостях, не приводит к исключению такой избыточности. В этом случае используют декомпозицию, основанную на *многозначных зависимостях*.

Многозначная зависимость является обобщением функциональной зависимости и рассматривает соответствия между *множествами* значений атрибутов.

В качестве примера рассмотрим отношение ПРЕПОДАВАТЕЛЬ (ИМЯ, КУРС, УЧЕБНОЕ_ПОСОБИЕ), хранящее сведения о курсах, читаемых преодавателем, и написанных им учебниках. Пусть профессор N читает курсы "Теория упругости" и "Теория колебаний" и имеет соответствующие учебные пособия, а профессор K читает курс "Теория удара" и является автором учебников "Теория удара" и "Теоретическая механика". Тогда наше отношение будет иметь вид:

ИМЯ	КУРС	УЧЕБНОЕ_ПОСОБИЕ
N	Теория упругости	Теория упругости
N	Теория колебаний	Теория упругости
N	Теория упругости	Теория колебаний
N	Теория колебаний	Теория колебаний
K	Теория удара	Теория удара
K	Теория удара	Теоретическая механика

добавляем:

K	Теория упругости	Теория удара
K	Теория упругости	Теоретическая механика

Это отношение имеет значительную избыточность и его использование приводит к возникновению *аномалии обновления*. Например, добавление информации о том, что профессор K будет также читать лекции по курсу "Теория упругости" приводит к необходимости добавить два кортежа (по одному для каждого написанного им учебника) вместо одного. Тем не менее, отношение ПРЕПОДАВАТЕЛЬ находится в NFBC (ключевой атрибут - ИМЯ).

Заметим, что указанные аномалии исчезают при замене отношения ПРЕПОДАВАТЕЛЬ его проекциями:

ИМЯ	КУРС		ИМЯ	УЧЕБНОЕ_ПОСОБИЕ
N	Теория упругости		N	Теория упругости
N	Теория колебаний		N	Теория колебаний
K	Теория удара		K	Теоретическая механика
K	Теория упругости		K	Теория удара

Аномалия обновления возникает в данном случае потому, что в отношении ПРЕПОДАВАТЕЛЬ имеются:

1. зависимость *множества* значений атрибута КУРС от *множества* значений атрибута ИМЯ
2. зависимость *множества значений атрибута УЧЕБНОЕ_ПОСОБИЕ от множества значений атрибута ИМЯ.*

Такие зависимости и называются многозначными и обозначаются как

ИМЯ ->> КУРС *ИМЯ ->> УЧЕБНОЕ_ПОСОБИЕ*

Нетрудно показать, что многозначные зависимости всегда образуют связанные пары, поэтому их часто обозначают

ИМЯ ->> КУРС | УЧЕБНОЕ_ПОСОБИЕ

Очевидно, что каждая функциональная зависимость является многозначной, но не каждая многозначная зависимость является функциональной.

Определение четвертой нормальной формы:

Отношение находится в 4NF если оно находится в BCNF и в нем отстутсвуют многозначные зависимости, не являющиеся функциональными зависимостями.

4.2.7. Зависимости по соединению и пятая нормальная форма (5NF).

До сих пор мы предполагали, что единственной операцией, необходимой для устранения избыточности в отношении, была декомпозиция его на две проекции. Однако, существуют отношения, для которых нельзя выполнить декомпозицию без потерь на две проекции, но которые можно подвергнуть декомпозиции без потерь на три (или более) проекций. Этот факт получил название ***зависимости по соединению****, а такие отношения называют* ***3-декомпозируемые отношения*** *(ясно, что любое отношение можно назвать "n-декомпозируемым", где n >= 2).*

Детально этот вопрос здесь мы не обсуждаем (полное изложение см. в книге К.Дейта), заметим лишь, что зависимость по соединению является обощением многозначной зависимости. Отношения, в которых имеются зависимости по соединению, не являющиеся одновременно ни многозначными, ни функциональными, также характеризуются аномалиями обновления. Поэтому, вводится понятие пятой нормальной формы.

Определение пятой нормальной формы:

Отношение находится в 5НФ тогда и только тогда, когда любая зависимость по соединению в нем определяется только его возможными ключами.

Другими словами, каждая проекция такого отношения содержит не менее одного возможного ключа и не менее одного неключевого атрибута.

Ограничения целостности

Целостность данных - это механизм поддержания соответствия базы данных предметной области. В реляционной модели данных определены два базовых требования обеспечения целостности:

- целостность ссылок
- целостность сущностей.

Целостность сущностей.

Объект реального мира представляется в реляционной базе данных как кортеж некоторого отношения. Требование целостности сущностей заключается в следующем:

каждый кортеж любого отношения должен отличатся от любого другого кортежа этого отношения (т.е. любое отношение должно обладать первичным ключом).

Вполне очевидно, что если данное требование не соблюдается (т.е. кортежи в рамках одного отношения не уникальны), то в базе данных может хранится противоречивая информация об одном и том же объекте. Поддержание целостности сущностей обеспечивается средствами системы управления базой данных (СУБД). Это осуществляется с помощью двух ограничений:

- при добавлении записей в таблицу проверяется уникальность их первичных ключей
- не позволяется изменение значений атрибутов, входящих в первичный ключ.

Целостность ссылок

Сложные объекты реального мира представляются в реляционной базе данных в виде кортежей нескольких нормализованных отношений, связанных между собой. При этом:

1. Связи между данными отношениями описываются в терминах функциональных зависимостей.
2. Для отражения функциональных зависимостей между кортежами разных отношений используется дублирование первичного ключа одного отношения (родительского) в другое (дочернее). Атрибуты, представляющие собой копии ключей родительских отношений, называются внешними ключами.

Требование целостности по ссылкам состоит в следующем:

для каждого значения внешнего ключа, появляющегося в дочернем отношении, в родительском отношении должен найтись кортеж с таким же значением первичного ключа.

Пусть, например, даны отношения ОТДЕЛ (N_ОТДЕЛА, ИМЯ_ОТДЕЛА) и СОТРУДНИК (N_СОТРУДНИКА, N_ОТДЕЛА, ИМЯ_СОТРУДНИКА), в которых хранятся сведения о работниках предприятия и подразделениях, где они работают. Отношение ОТДЕЛ в данной паре является родительским, поэтому его первичный ключ *"N_отдела"* присутствует в дочернем отношении СОТРУДНИК. Требование целостности по ссылкам означает здесь, что в таблице СОТРУДНИК не может присутствовать кортеж со значением атрибута *"N_отдела"*, которое не встречается в таблице ОТДЕЛ. Если такое значение в отношении ОТДЕЛ отсутствует, значение внешнего ключа в отношении СОТРУДНИК считается неопределенным.

Как правило, поддержание целостности ссылок также возлагается на систему управления базой данных. Например, она может не позволить пользователю добавить запись, содержащую внешний ключ с несуществующим (неопределенным) значением.

В заключение этого раздела отметим, что часто вместо выражения "целостность по ссылкам" употребляют его синонимы "ссылочная целостность", "целостность связей" или "требование внешнего ключа".

ОПЕРАЦИИ НАД ДАННЫМИ (РЕЛЯЦИОННАЯ АЛГЕБРА).

Операции обработки кортежей.

Эти операции связаны с изменением состава кортежей в каком-либо отношении.

- ДОБАВИТЬ - необходимо задать имя отношения и ключ кортежа.
- УДАЛИТЬ - необходимо указать имя отношения, а также идентифицировать кортеж или группу кортежей, подлежащих удалению.
- ИЗМЕНИТЬ - выполняется для названного отношения и может корректировать как один, так и несколько кортежей.

4.4.2.Операции обработки отношений.

На входе каждой такой операции используется одно или несколько отношений, результатом выполения операции всегда является новое отношение.

В рассмотренных ниже примерах (которые заимствованы из книги Э.Озкарахан "Машины баз данных и управление базами данных" -М: "Мир", 1989) используются следующие отношения:

```
P(D1,D2,D3)   Q(D4,D5)    R(M,P,Q,T)      S(A,B)

 1 11 x          x 1       x 101 5 a        5 a
 2 11 y          x 2       y 105 3 a       10 b
 3 11 z          y 1       z 500 9 a       15 c
 4 12 x                    w  50 1 b        2 d
                           w  10 2 b        6 a
                           w 300 4 b        1 b
```

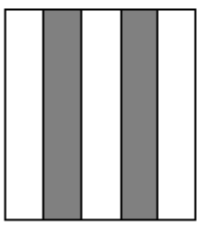
В реляционной алгебре определены следующие операций обработки отношений:

- ПРОЕКЦИЯ (ВЕРТИКАЛЬНОЕ ПОДМНОЖЕСТВО).
  Операция проекции представляет из себя выборку из каждого кортежа отношения значений атрибутов, входящих в список A, и удаление из полученного отношения повторяющихся строк.

Проекция / PROJECT /

Обозначение	Определение	LEAP
$R[A]$	$\{r[A] : r \in R\}$	r = project (R) $(A_1, A_2, \ldots, A_n)$

Пример:



$$R[M,T] = \begin{bmatrix} x & a \\ y & a \\ z & a \\ w & b \\ w & b \\ w & b \end{bmatrix} = \begin{bmatrix} x & a \\ y & a \\ z & a \\ w & b \end{bmatrix}$$

- ВЫБОРКА (ОГРАНИЧЕНИЕ, ГОРИЗОНТАЛЬНОЕ ПОДМНОЖЕСТВО).
  На входе используется одно отношение, результат - новое отношение, построенное по той же схеме, содержащее подмножество кортежей исходного отношения, удовлетворяющих условию выборки.

Выборка / SELECT /

Обозначение	Определение	LEAP
$R[A\ \theta v]$ $R[A_1\ \theta A_2]$	$\{r : r \in R \wedge (r[A]\ \theta v)\}$ $\{r : r \in R \wedge (r[A_1]\ \theta r[A_2])\}$	r = select (R) ((cond) bool (cond))

Пример:

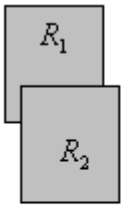
$$P[D_2 = 11] = \begin{bmatrix} 1 & 11 & x \\ 2 & 11 & y \\ 3 & 11 & z \end{bmatrix}$$

- ОБЪЕДИНЕНИЕ.
  Отношения-операнды в этом случае должны быть определены по одной схеме. Результирующее отношение содержит все строки операндов за исключением повторяющихся.

Объединение / UNION /

Обозначение	Определение	LEAP
$R_1 \cup R_2$	$\{r : r \in R_1 \vee r \in R_2\}$	r = (R1) union (R2)

Пример:



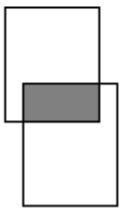
$$R[Q,T] \cup S = \begin{bmatrix} 5 & a \\ 3 & a \\ 9 & a \\ 1 & b \\ 2 & b \\ 4 & b \end{bmatrix} \cup \begin{bmatrix} ~~5~~ & ~~a~~ \\ 10 & b \\ 15 & c \\ 2 & d \\ 6 & a \\ ~~1~~ & ~~b~~ \end{bmatrix} = \begin{bmatrix} 5 & a \\ 3 & a \\ 9 & a \\ 1 & b \\ 2 & b \\ 4 & b \\ 10 & b \\ 15 & c \\ 2 & d \\ 6 & a \end{bmatrix}$$

- ПЕРЕСЕЧЕНИЕ.
  На входе операции два отношения, определенные по одной схеме. На выходе - отношение, содержащие кортежи, которые присутствуют в обоих исходных отношениях.

Пересечение / INTERSECT /

Обозначение	Определение	LEAP
$R_1 \cap R_2$	$\{r : r \in R_1 \wedge r \in R_2\}$	r = (R1) intersect (R2)

Пример:



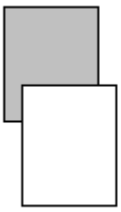
$$R[Q,T] \cap S = \begin{bmatrix} 5 & a \\ 3 & a \\ 9 & a \\ 1 & b \\ 2 & b \\ 4 & b \end{bmatrix} \cap \begin{bmatrix} 5 & a \\ 10 & b \\ 15 & c \\ 2 & d \\ 6 & a \\ 1 & b \end{bmatrix} = \begin{bmatrix} 5 & a \\ 1 & b \end{bmatrix}$$

- РАЗНОСТЬ.
  Операция во многом похожая на ПЕРЕСЕЧЕНИЕ, за исключением того, что в результирующем отношении содержатся кортежи, присутствующие в первом и отсутствующие во втором исходных отношениях.

Разность / SET DIFFERENCE /

Обозначение	Определение	LEAP
$R_1 - R_2$	$\{r : r \in R_1 \wedge r \notin R_2\}$	r = (R1) difference (R2) r = (R1) minus (R2)

Пример:



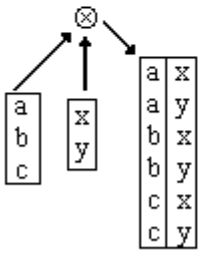
$$R[Q,T] - S = \begin{bmatrix} 5 & a \\ 3 & a \\ 9 & a \\ 1 & b \\ 2 & b \\ 4 & b \end{bmatrix} \cap \begin{bmatrix} 5 & a \\ 10 & b \\ 15 & c \\ 2 & d \\ 6 & a \\ 1 & b \end{bmatrix} = \begin{bmatrix} 3 & a \\ 9 & a \\ 2 & b \\ 4 & b \end{bmatrix}$$

- ДЕКАРТОВО ПРОИЗВЕДЕНИЕ
  Входные отношения могут быть определены по разным схемам. Схема результирующего отношения включает все атрибуты исходных. Кроме того:
  - степень результирующего отношения равна сумме степеней исходных отношений
  - мощность результирующего отношения равна произведению мощностей исходных отношений.

Декартово произведение / CARTESIAN PRODUCT /

Обозначение	Определение	LEAP
$R_1 \otimes R_2$	$\{(r_1 \| r_2) : r_1 \in R_1 \wedge r_2 \in R_2\}$	r = (R1) product (R2)

Пример:



$$R[M,T] \otimes (R[Q,T] \cap S) = \begin{bmatrix} x & a \\ y & a \\ z & a \\ w & b \end{bmatrix} \otimes \begin{bmatrix} 5 & a \\ 1 & b \end{bmatrix} = \begin{bmatrix} x & a & 5 & a \\ x & a & 1 & b \\ y & a & 5 & a \\ y & a & 1 & b \\ z & a & 5 & a \\ z & a & 1 & b \\ w & b & 5 & a \\ w & b & 1 & b \end{bmatrix}$$

- СОЕДИНЕНИЕ
  Данная операция имеет сходство с ДЕКАРТОВЫМ ПРОИЗВЕДЕНИЕМ. Однако, здесь добавлено условие, согласно которому вместо полного произведения всех строк в результирующее отношение включаются только строки, удовлетворяющие определенному соотношению между атрибутами соединения ***(A_1,A_2)*** соответствующих отношений.

Соединение / JOIN /

Обозначение	Определение
$R_1 [A_1 \ \theta A_2] R_2$	$\{(r_1 \| r_2) : r_1 \in R_1 \wedge r_2 \in R_2 \wedge (r_1[A_1] \ \theta r_2[A_2])\}$

LEAP : r = join (R1) (R2) ((cond) bool (cond))

Пример:

$$P[D_3 = D_4]Q = \begin{bmatrix} 1 & 11 & x & x & 1 \\ 1 & 11 & x & x & 2 \\ 2 & 11 & y & y & 1 \\ 4 & 12 & x & x & 1 \\ 4 & 12 & x & x & 2 \end{bmatrix} = \begin{bmatrix} 1 & 11 & x & 1 \\ 1 & 11 & x & 2 \\ 2 & 11 & y & 1 \\ 4 & 12 & x & 1 \\ 4 & 12 & x & 2 \end{bmatrix}$$

- ДЕЛЕНИЕ
  Пусть отношение **R** , называемое делимым, содержит атрибуты ***(A_1,A_2,...,A_n)***. Отношение **S** - делитель содержит подмножество атрибутов **A**: ***(A_1,A_2,...,A_k) (k<n)***. Результирующее отношение **C** определено на атрибутах отношения **R**, которых нет в **S**, т.е. ***A_{k+1},A_{k+2},...,A_n***. Кортежи включаются в результирующее отношение **C** только в том случае, если его декартово произведение с отношением **S** содержится в делимом **R**.

Деление / DIVISION /

Обозначение	Определение	LEAP :
$\overline{R_1\left[A_1 \div A_2\right]R_2}$	$\left\{r\left[A_1\right] : r \in R_1 \wedge R_2\left[A_2\right] \subseteq g_R\left(r\left[A_1\right]\right)\right\}$	не поддерживается

Пример:

пусть

$$R_1\left(A_1, A_2\right) = \begin{bmatrix} a & x \\ a & y \\ a & z \\ b & x \\ c & y \end{bmatrix} \quad R_2\left(A_3, A_4\right) = \begin{bmatrix} 1 & x \\ 2 & x \\ 1 & y \end{bmatrix}$$

тогда

$$R_1\left[A_2 \div A_4\right] = \begin{bmatrix} a & x \\ a & y \\ a & z \\ b & x \\ c & y \end{bmatrix} \div \begin{bmatrix} x \\ y \end{bmatrix} = \left[a\right]$$

Реляционное исчисление.

В реляционной модели определяются два базовых механизма манипулирования данными:

- основанная на теории множеств реляционная алгебра
- основанное на математической логике реляционное исчисление.

Также как и выражения реляционной алгебры формулы реляционного исчисления определяются над отношениями реляционных баз данных, и результатом вычисления также является отношение.

Эти механизмы манипулирования данными различаются уровнем процедурности:

- запрос, представленный на языке релационной алгебры, может быть вычислен на основе вычисления элементарных алгебраичесских операций с учетом их старшинства и возможных скобок
- формула реляционного исчисления только устанавливает условия, которым должны удовлетворять кортежи результирующего отношения. Поэтому языки реляционного исчисления являются более непроцедурными или декларативными.

Пример: Пусть даны два отношения:

СОТРУДНИКИ (СОТР_НОМЕР, СОТР_ИМЯ, СОТР_ЗАРПЛ, ОТД_НОМЕР)
ОТДЕЛЫ(ОТД_НОМЕР, ОТД_КОЛ, ОТД_НАЧ)

Мы хотим узнать имена и номера сотрудников, являющихся начальниками отделов с количеством работников более 10. Выполнение этого запроса средствами реляционной алгебры распадается на четко определенную последовательность шагов:

(1).выполнить соединение отношений СОТРУДНИКИ и ОТДЕЛЫ по условию СОТР_НОМ = ОТДЕЛ_НАЧ.

C1 = СОТРУДНИКИ [СОТР_НОМ = ОТД_НАЧ] ОТДЕЛЫ

(2).из полученного отношения произвести выборку по условию ОТД_КОЛ > 10

C2 = C1 [ОТД_КОЛ > 10].

(3).спроецировать результаты предыдущей операции на атрибуты СОТР_ИМЯ, СОТР_НОМЕР

C3 = C2 [СОТР_ИМЯ, СОТР_НОМЕР]

Заметим, что порядок выполнения шагов может повлиять на эффективность выполнения запроса. Так, время выполнения приведенного выше запроса можно сократить, если поменять местами этапы (1) и (2). В этом случае сначала из отношения СОТРУДНИКИ будет сделана выборка всех кортежей со значением атрибута ОТДЕЛ_КОЛ > 10, а затем выполнено соединение результирующего отношения с отношением ОТДЕЛЫ. Машинное время экономится за счет того, что в операции соединения участвуют меньшие отношения.

На языке реляционного исчисления данный запрос может быть записан как:

```
Выдать СОТР_ИМЯ и СОТР_НОМ для СОТРУДНИКИ таких, что
существует ОТДЕЛ с таким же, что и СОТР_НОМ значением ОТД_НАЧ
и значением ОТД_КОЛ большим 50.
```

Здесь мы указываем лишь характеристики результирующего отношения, но не говорим о способе его формирования. СУБД сама должна решить какие операции и в каком порядке надо выполнить над отношениями СОТРУДНИКИ и ОТДЕЛЫ. Задача оптимизации выполнения запроса в этом случае также ложится на СУБД.

Язык SQL

В предыдущих разделах мы рассмотрели "штатные" средства манипулирования данными, поддерживаемые реляционной моделью - реляционная алгебра и реляционное исчисление. Однако, на практике крайне редко одно из этих средств принимается в качестве полной основы какого-либо языка базы данных. Так и SQL (Structured Query Language - структурированный язык запросов) основывается на некоторой смеси алгебраических и логических конструкций.

Язык SQL (эта аббревиатура должна произноситься как "сикуель", однако все чаще говорят "эс-ку-эль") в настоящее время является промышленным стандартом, который в большей или меньшей степени поддерживает любая СУБД, претендующая на звание "реляционной". В то же время SQL подвергается суровой критике как раз за недостаточное соответствие реляционным принципам *(см. например, статью Х. Дарвина и К.Дейта Третий манифест, опубликованную в журнале СУБД N 1 за 1996 год)*.

Из истории SQL:

В начале 70-х годов в компании IBM была разработана экспериментальная СУБД System R на основе языка SEQUEL (Structured English Qeury Language - структурированный английский язык запросов), который можно считать непосредственным предшественником SQL. Целью разработки было создание простого непроцедурного языка, которым мог воспользоваться любой пользователь, даже не имеющий навыков программирования. В 1981 году IBM объявила о своем первом, основанном на SQL программном продукте, SQL/DS. Чуть позже к ней присоединились Oracle и другие производители. Первый стандарт языка SQL был принят

Американским национальным институтом стандартизации (ANSI) в 1987 (так называемый SQL level /уровень/ 1) и несколько уточнен в 1989 году (SQL level 2). Дальнейшее развитие языка поставщиками СУБД потребовало принятия в 1992 нового расширенного стандарта (ANSI SQL-92 или просто SQL-2). В настоящее время ведется работа по подготовке третьего стандарта SQL, который должен включать элементы объекто-ориентрованного доступа к данным.

Необходимо сказать, что хотя SQL и задумывался как средство работы конечного пользователя, в конце концов он стал настолько сложным, что превратился в инструмент программиста. Вопросы создания приложений обработки данных с использованием SQL рассматриваются в конце данной главы.

В SQL определены два подмножества языка:

- **SQL-DDL** (Data Definition Language) - язык определения структур и ограничений целостности баз данных. Сюда относятся команды создания и удаления баз данных; создания, изменения и удаления таблиц; управления пользователями и т.д.
- **SQL-DML** (Data Manipulation Language) - язык манипулирования данными: добавление, изменение, удаление и извлечение данных, управления транзакциями

Здесь не дается строгое описание всех возможностей SQL-92. Во-первых, ни одна СУБД не поддерживает их в полной мере, а во-вторых, производители СУБД часто предлагают собственные расширения SQL, несовместимые друг с другом. Поэтому мы рассматриваем некое подмножество языка, которое дает общее представление о его специфике и возможностях. В то же время, этого подмножества достаточно, чтобы начать самостоятельную работу с любой СУБД. Более формальный (и более полный) обзор стандартов SQL сделан в статье С. Д. Кузнецова "Стандарты языка реляционных баз данных SQL: краткий обзор",журнал СУБД N 2, 1996 г. Ознакомится с русским переводом стандарта SQL можно на сервере Центра информационных технологий, сравнительное описание различных версий языка (для СУБД Sybase SQL Server, Sybase SQL Anywhere, Microsoft SQL Server, Informix, Oracle Server) приводится в книге Дж.Боуман, С.Эмерсон, М.Дарновски "Практическое руководство по SQL", Киев, Диалектика, 1997.

Следует также отметить, что в отличие от "теретической" терминологии, используемой при описании реляционной модели (*отношение, атрибут, кортеж*), в литературе при описании SQL часто используется терминология "практическая" (соответственно - *таблица, столбец, строка*). Здесь мы следуем этой традиции.

Все примеры построены применительно к базе данных **publications**, содержащей сведения о публикациях (как печатных, так и электронных), относящихся к теме данного курса. Структуру этой базы данных можно посмотреть здесь, ее проектирование описано в разделе 5.4, доступ к ней для практических занятий можно получить через Internet посредством СУБД Leap (реляционная алгебра) или СУБД PostgreSQL. (язык SQL).

4.6.1.Типы данных SQL.

- Символьные типы данных - содержат буквы, цифры и специальные символы.
    - CHAR или CHAR(n) -символьные строки фиксированной длины. Длина строки определяется параметром n. CHAR без параметра соответсвует CHAR(1). Для хранения таких данных всегда отводится n байт вне зависимости от реальной длины строки.

    - VARCHAR(n) - символьная строка переменной длины. Для хранения данных этого типа отводится число байт, соответствующее реальной длине строки.
- Целые типы данных - поддерживают только целые числа (дробные части и десятичные точки не допускаются). Над этими типами разрешается выполнять арифметические операции и применять к ним агрегирующие функции (определение максимального, минимального, среднего и суммарного значения столбца реляционной таблицы).
    - INTEGER или INT- целое, для хранения которого отводится, как правило, 4 байта. (Замечание: число байт, отводимое для хранения того или иного числового типа данных зависит от используемой СУБД и аппаратной платформы, здесь приводятся наиболее "типичные" значения) Интервал значений от - 2147483647 до + 2147483648
    - SMALLINT - короткое целое (2 байта), интервал значений от - 32767 до +32768
- Вещественные типы данных - описывают числа с дробной частью.
    - FLOAT и SMALLFLOAT - числа с плавающей точкой (для хранения отводится обычно 8 и 4 байта соответсвенно).
    - DECIMAL(p) - тип данных аналогичный FLOAT с числом значащих цифр p.
    - DECIMAL(p,n) - аналогично предыдущему, p - общее количество десятичных цифр, n - количество цифр после десятичной запятой.
- Денежные типы данных - описывают, естественно, денежные величины. Если в ваша система такого типа данных не поддерживает, то используйте DECIMAL(p,n).
    - MONEY(p,n) - все аналогично типу DECIMAL(p,n). Вводится только потому, что некоторые СУБД предусматривают для него специальные методы форматирования.
- Дата и время - используются для хранения даты, времени и их комбинаций. Большинство СУБД умеет определять интервал между двумя датами, а также уменьшать или увеличивать дату на определенное количество времени.
    - DATE - тип данных для хранения даты.
    - TIME - тип данных для хранения времени.
    - INTERVAL - тип данных для хранения верменного интервала.
    - DATETIME - тип данных для хранения моментов времени (год + месяц + день + часы + минуты + секунды + доли секунд).
- Двоичные типы данных - позволяют хранить данные любого объема в двоичном коде (оцифрованные изображения, исполняемые файлы и т.д.). Определения этих типов наиболее сильно различаются от системы к системе, часто используются ключевые слова:
    - BINARY
    - BYTE
    - BLOB
- Последовательные типы данных - используются для представления возрастающих числовых последовательностей.
    - SERIAL - тип данных на основе INTEGER, позволяющий сформировать уникальное значение (например, для первичного ключа). При добавлении записи СУБД автоматически присваивает полю данного типа значение, получаемое из возрастающей последовательности целых чисел.

В заключение следует сказать, что для всех типов данных имеется общее значение NULL - "не определено". Это значение имеет каждый элемент столбца до тех пор, пока в него не будут введены данные. При создании таблицы можно явно указать СУБД могут ли элементы того или иного столбца иметь значения NULL (это не допустимо, например, для столбца, являющего первичным ключом).

Операторы создания схемы базы данных.

При описании команд предполагается, что:

- текст, набранный строчными буквами (например, **CREATE TABLE**) является обязательным
- текст, набранный прописными буквами и заключенный в угловые скобки (например, **<имя_базы_данных>**) обозначает переменную, вводимую пользователем
- в квадратные скобки (например, **[NOT NULL]**) заключается необязательная часть команды
- взаимоисключающие элементы команды разделяются вертикальной чертой (например, **[UNIQUE | PRIMARY KEY]**).

Операторы базы данных

Команда	Описание
CREATE DATABASE <имя_базы_данных>	Создание базы данных.
DROP DATABASE <имя_базы_данных>	Удаление базы данных.

Создание и удаление таблиц

Создание таблицы:

```
   CREATE TABLE <имя_таблицы>
        (<имя_столбца> <тип_столбца>
                          [NOT NULL]
                          [UNIQUE | PRIMARY KEY]
                          [REFERENCES <имя_мастер_таблицы>
[<имя_столбца>]]
             , ...)
```

Пользователь обязан указать имя таблицы и список столбцов. Для каждого столбца обязательно указываются его имя и тип (см. таблицу в предыдущем разделе), а также опционально могут быть указаны параметры

- **NOT NULL** - в этом случае элементы столбца всегда должны иметь определенное значение (не NULL)
- один из взаимоисключающих параметров **UNIQUE** - значение каждого элемента столбца должно быть уникальным или **PRIMARY KEY** - столбец является первичным ключом.
- **REFERNECES <имя_мастер_таблицы> [<имя_столбца>]** - эта конструкция определяет, что данный столбец является внешним ключом и указывает на ключ какой мастер_таблицы он ссылается.

Контроль за выполнением указанных условий осуществляет СУБД.

Пример: создание базы данных ***publications:***

```
   CREATE DATABASE publications;

   CREATE TABLE authors (au_id INT PRIMARY KEY,
                         author VARCHAR(25) NOT NULL);
   CREATE TABLE publishers (pub_id INT PRIMARY KEY,
                            publisher VARCHAR(255) NOT NULL,url
VARCHAR(255));
   CREATE TABLE titles (title_id INT PRIMARY KEY,
                        title VARCHAR(255) NOT NULL,
                        yearpub INT,
```

```
                          pub_id INT REFERENCES publishers(pub_id));
   CREATE TABLE titleautors (au_id INT REFERENCES authors(au_id),
                             title_id INT REFERENCES
titles(title_id));
   CREATE TABLE wwwsites (site_id INT PRIMARY KEY,
                          site VARCHAR(255) NOT NULL,
                          url VARCHAR(255));
   CREATE TABLE wwwsiteauthors (au_id INT REFERENCES authors(au_id),
                                site_id INT REFERENCES
wwwsites(site_id));
```

Удаление таблицы:

```
DROP TABLE <имя_таблицы>
```

Модификация таблицы:

Добавить столбцы	**ALTER TABLE <имя_таблицы> ADD (<имя_столбца> <тип_столбца> [NOT NULL] [UNIQUE | PRIMARY KEY] [REFERENCES <имя_мастер_таблицы> [<имя_столбца>]] ,...)**
Удалить столбцы	**ALTER TABLE <имя_таблицы> DROP (<имя_столбца>,...)**
Модификация типа столбцов	**ALTER TABLE <имя_таблицы> MODIFY (<имя_столбца> <тип_столбца> [NOT NULL] [UNIQUE | PRIMARY KEY] [REFERENCES <имя_мастер_таблицы> <имя_столбца>]] ,...)**

4.6.3.DDL: Операторы создания индексов.

Создание индекса:

```
   CREATE [UNIQUE] INDEX <имя_индекса> ON <имя_таблицы>
(<имя_столбца>,...)
```

Эта команда создает индекс с заданным именем для таблицы <имя_таблицы> по столбцам, входящим в список, указанный в скобках. Индекс часто представляет из себя структуру типа B-дерева (см. параграф 1.2), но могут использоваться и другие структуры. Создание индексов значительно ускоряет работу с таблицами. В случае указания необязательного параметра **UNIQUE** СУБД будет проверять каждое значение индекса на уникальность.

Очень часто встает вопрос, какие поля необходимо индексировать. Обязательно надо строить индексы для первичных ключей, поскольку по их значениям осуществляется доступ к данным при операциях соединения двух и более таблиц. Также в ответе на этот вопрос поможет анализ наиболее частых запросов к базе данных. Например, для БД **publications** можно ожидать, что

одним из наиболее частых запросов будет выборка всех публикаций данного автора. Для минимизации времени этого запроса необходимо посроить индекс для таблицы **authors** по именам авторов:

```
CREATE INDEX au_names ON authors (author);
```

Создание индексов для первичных ключей:

```
CREATE INDEX au_index ON authors (au_id);
CREATE INDEX title_index ON titles (title_id);
CREATE INDEX pub_index ON publishers (pub_id);
CREATE INDEX site_index ON wwwsites (site_id);
```

Первоначальное определение структуры индексов производится разработчиком на стадии создания прикладной системы. В дальнейшем она уточняется администратором системы по результатам анализа ее работы, учета наиболее часто выполняющихся запросов и т.д.

Удаление индекса:

```
DROP INDEX <имя_индекса>
```

4.6.4.DDL: Операторы управления правами доступа.

По соображениям безопасности не каждому пользователю прикладной системы может быть разрешено получать информацию из какой-либо таблицы, а тем более изменять в ней данные. Для определения прав пользователей относительно объектов базы данных (таблицы, представления, индексы) в **SQL** определена пара команд `GRANT` и `REVOKE`. Синтаксис операции передачи прав на таблицу:

```
GRANT <тип_права_на_таблицу>
  ON  <имя_таблицы> [<список_столбцов>]
  TO  <имя_пользователя>
```

Права пользователя на уровне таблицы определяются следующими ключевыми словами (как мы увидим чуть позже эти ключевые слова совпадают с командами выборки и изменения данных):

- `SELECT` - получение информации из таблицы
- `UPDATE` - изменение информации в таблице
- `INSERT` - добавление записей в таблицу
- `DELETE` - удаление записей из таблицы
- `INDEX` - индексирование таблицы
- `ALTER` - изменение схемы определения таблицы
- `ALL` - все права

В поле <тип_права_на_таблицу> может быть указано либо ключевое слово `ALL` или любая комбинация других ключевых слов. Например, предоставим все права на таблицу **publishers** пользователю **andy**:

```
GRANT ALL ON publishers TO andy;
```

Пользователю **peter** предоставим права на извлечение и дбавление записей на эту же таблицу:

```
GRANT SELECT INSERT ON publishers TO peter;
```

В том случае, когда одинаковые права надо предоставить сразу всем пользователям, вместо выполнения команды GRANT для каждого из них, можно вместо имени пользователя указать ключевое слово PUBLIC:

```
GRANT SELECT ON publishers TO PUBLIC;
```

Отмена прав осуществляется командой REVOKE:

```
REVOKE <тип_права_на_таблицу>
   ON  <имя_таблицы> [<список_столбцов>]
   FROM  <имя_пользователя>
```

Все ключевые слова данной команды эквивалентны оператору GRANT.

Большинство систем поддерживают также команду GRANT для назначения привилегий на базу данных в целом. В этом случае формат команды:

```
GRANT <тип_права_на_базу_данных>
  ON  <имя_базы данных>
  TO  <имя_пользователя>
```

К сожалению, способы задания прав на базу данных различны для разных СУБД, и точную их формулировку нужно уточнять в документации. В качестве примера приведем список прав на базу данных, поддерживаемых СУБД *Informix*:

- CONNECT - права на доступ к данным и их модификацию, если это разрешено на уровне таблицы;
- RESOURCE - права на управление ресурсами. Все перечисленное выше плюс права на создание новых объектов (таблиц, индексов и т.д.) и удаление и изменение тех объектов, которыми данный пользователь владеет;
- DBA - права на администрирование. Все права на управление ресурсами плюс права на удаление базы данных, удаление любых объектов, назначение и отмена прав других пользователей.

Отмена прав на базу данных осуществляется командой:

```
REVOKE <тип_права_на_базу_данных> FROM  <имя_пользователя>
```

Команды модификации данных.

К этой группе относятся операторы добавления, изменения и удаления записей.

Добавить новую запись в таблицу:

```
INSERT INTO <имя_таблицы> [ (<имя_столбца>,<имя_столбца>,...) ]
                                  VALUES (<значение>,<значение>,..)
```

Список столбцов в данной команде не является обязательным параметром. В этом случае должны быть указаны значения для всех полей таблицы в том порядке, как эти столбцы были перечислены в команде CREATE TABLE, например:

```
     INSERT INTO publishers VALUES (16,"Microsoft
Press","http://www.microsoft.com");
```

Пример с указанием списка столбцов:

```
     INSERT INTO publishers (publisher,pub_id)
            VALUES ("Super Computer Publishing",17);
```

Модификация записей:

```
     UPDATE <имя_таблицы> SET <имя_столбца>=<значение>,...
         [WHERE <условие>]
```

Если задано ключевое слово WHERE и условие, то команда UPDATE применяется только к тем записям, для которых оно выполняется. Если условие не задано, UPDATE применяется ко всем записям. Пример:

```
UPDATE publishers SET url="http://www.superpub.com" WHERE
pub_id=17;
```

В качестве условия используются логические выражения над константами и полями. В условиях допускаются:

- операции сравнения: > , < , >= , <= , = , <> , != . В SQL эти операции могут применяться не только к числовым значениям, но и к строкам ("<" означает раньше, а ">" позже в алфавитном порядке) и датам ("<" раньше и ">" позже в хронологическом порядке).
- оперции проверки поля на значение NULL: IS NULL, IS NOT NULL
- операции проверки на вхождение в диапазон: BETWEEN и NOT BETWEEN.
- операции проверки на вхождение в список: IN и NOT IN
- операции проверки на вхождение подстроки: LIKE и NOT LIKE
- отдельные операции соединяются связями AND, OR, NOT и группируются с помощью скобок.

Подробно все эти ключевые слова будут описаны и проиллюстрированы в параграфе, посвященном оператору SELECT. Здесь мы ограничимся приведением несложного примера:

```
UPDATE publishers SET url="url not defined" WHERE url IS NULL;
```

Эта команда находит в таблице **publishers** все неопределенные значения столбца **url** и заменяет их строкой "url not defined".

Удаление записей

```
DELETE FROM <имя_таблицы> [ WHERE <условие> ]
```

Удаляются все записи, удовлетворяющие указанному условию. Если ключевое слово WHERE и условие отстутствуют, из таблицы удаляются все записи. Пример:

```
DELETE FROM publishers WHERE publisher = "Super Computer
Publishing";
```

Эта команда удаляет запись об издательстве Super Computer Publishing.

4.6.6.DML: Выборка данных.

Для извлечения записей из таблиц в SQL определен оператор **SELECT**. С помощью этой команды осуществляется не только операция реляционной алгебры "выборка" (горизонтальное подмножество), но и предварительное соединение (join) двух и более таблиц. Это наиболее сложное и мощное средство SQL, полный синтаксис оператора **SELECT** имеет вид:

```
SELECT [ALL | DISTINCT] <список_выбора>
     FROM <имя_таблицы>, ...
     [ WHERE <условие> ]
     [ GROUP BY <имя_столбца>,... ]
        [ HAVING <условие> ]
     [ORDER BY <имя_столбца> [ASC | DESC],... ]
```

Порядок предложений в операторе **SELECT** должен строго соблюдаться (например, GROUP BY должно всегда предшествовать ORDER BY), иначе это приведет к появлению ошибок.

Мы начнем рассмотрение **SELECT** с наиболее простых его форм. Все примеры, приведенные ниже, касающиеся базы данных publications, можно выполнить самостоятельно, зайдя на эту страничку, поэтому результаты запросов здесь не приводятся.

Этот оператор всегда начинается с ключевого слова SELECT. В кострукции <список_выбора> определяется столбец или столбцы, включаемые в результат. Он может состоять из имен одного или нескольких столбцов, или из одного символа * (звездочка), определяющего все столбцы. Элементы списка разделяются запятыми.

Пример: получить список всех авторов

```
SELECT author FROM authors;
```

получить список всех полей таблицы **authors**:

```
SELECT * FROM authors;
```

В том случае, когда нас интересуют не все записи, а только те, котрые удовлетворяют некому условию, это условие можно указать после ключевого слова WHERE. Например, найдем все книги, опубликованные после 1996 года:

```
SELECT title FROM titles WHERE yearpub > 1996;
```

Допустим теперь, что нам надо найти все публикации за интервал 1995 - 1997 гг. Это условие можно записать в виде:

```
SELECT title FROM titles WHERE yearpub>=1995 AND
yearpub<=1997;
```

Другой вариант этой команды можно получить с использованием логической операции проверки на вхождение в интервал:

```
SELECT title FROM titles WHERE yearpub BETWEEN 1995 AND
1997;
```

При использовании конструкции NOT BETWEEN находятся все строки, не входящие в указанный диапазон.

Еще один вариант этой команды можно построить с помощью логической операции проверки на вхождение в список:

```
SELECT title FROM titles WHERE yearpub IN (1995,1996,1997);
```

Здесь мы задали в явном виде список интересующих нас значений. Конструкция NOT IN позволяет найти строки, не удовлетворяющие условиям, перечисленным в списке.

Наиболее полно преимущества ключевого слова IN проявляются во вложенных запросах, также называемых подзапросами. Предположим, нам нужно найти все издания, выпущенные компанией "Oracle Press". Наименования издательских компаний содержатся в таблице **publishers**, названия книг в таблице **titles**. Ключевое слово NOT IN позволяет объединить обе таблицы (без получения общего отношения) и извлечь при этом нужную информацию:

```
SELECT title FROM titles WHERE pub_id IN
   (SELECT pub_id FROM publishers WHERE publisher='Oracle Press');
```

При выполнении этой команды СУБД вначале обрабатывает вложенный запрос по таблице **publishers**, а затем его результат передает на вход основного запроса по таблице **titles.**

Некоторые задачи нельзя решить с использованием только операторов сравнения. Например, мы хоти найти web-site издательтва "Wiley", но не знаем его точного наименования. Для решения этой задачи предназначено ключевое слово LIKE, его синтаксис имеет вид:

```
WHERE <имя_столбца> LIKE <образец> [ ESCAPE <ключевой_символ> ]
```

Образец заключается в кавычки и должен содержать шаблон подстроки для поиска. Обычно в шаблонах используются два символа:

- % (знак процента) - заменяет любое количество символов
- _ (подчеркивание) - заменяет одиночный символ.

Попробуем найти искомый web-site:

```
SELECT publiser, url FROM publishers WHERE publisher LIKE '%Wiley%';
```

В соотвествии с шаблоном СУБД найдет все строки включающие в себя подстроку "Wiley".
Другой пример: найти все книги, название которых начинается со слова "SQL":

```
SELECT title FROM titles WHERE title LIKE 'SQL%';
```

В том случае, когда надо найти значение, которое само содержит один из символов шаблона, используют ключевое слово ESCAPE и <ключевой_символ>. Литерал, следующий в шаблоне после ключевого символа, рассматривается как обычный символ, все последующие символы имеют обычное значение. Например, нам надо найти ссылку на web-страницу, о которой известно, что в ее url содержится подстрока "my_works":

```
SELECT site, url FROM wwwsites WHERE url LIKE '%my@_works%'
ESCAPE '@';
```

В заключение заметим, что при выполнении оператора SELECT результирующее отношение может иметь несколько записей с одинаковыми значениями всех полей. Чтобы исключить повторяющиеся записи из выборки используется ключевое слово DISTINCT. Ключевое слово ALL указывает, что в результат необходимо включать все строки.

4.6.7.DML: Выборка из нескольких таблиц.

Очень часто возникает ситуация, когда выборку данных надо производить из отношения, которое является результатом слияния (join) двух других отношений. Например, нам нужно получить из базы данных **publications** информацию о всех печатных изданиях в виде следующей таблицы:

```
-------------------------------------------------
|название_книги | год_выпуска | издательство   |
-------------------------------------------------
|                |             |                |
|                |             |                |
```

Для этого СУБД предварительно должна выполнить слияние таблиц **titles** и **publishers**, а только затем произвести выборку из полученного отношения.

Для выполнения операции такого рода в операторе SELECT после ключевого слова FROM указывается список таблиц, по которым произвоится поиск данных. После ключевого слова WHERE указывается условие, по которому производится слияние. Для того, чтобы выполнить данный запрос, нужно дать команду:

```
SELECT titles.title,titles.yearpub,publishers.publisher
    FROM titles,publishers
    WHERE titles.pub_id=publishers.pub_id;
```

А вот пример, где одновременно задаются условия и слияния, и выборки (результат предыдущего запроса ограничивается изданиями после 1996 года):

```
SELECT titles.title,titles.yearpub,publishers.publisher
    FROM titles,publishers
    WHERE titles.pub_id=publishers.pub_id AND
          titles.yearpub>1996;
```

Следует обратить внимание на то, что когда в разных таблицах присутствуют одноименные поля, то для устранения неоднозначности перед именем поля указывается имя таблицы и знак "." (точка). (Хорошее правило: имя таблицы указывать всегда!)

Естественно, имеется возможность производить слияние и более чем двух таблиц. Например, чтобы дополнить описанную выше выборку именами авторов книг необходимо составить оператор следующего вида:

```
 SELECT
authors.author,titles.title,titles.yearpub,publishers.publisher
   FROM titles,publishers,titleauthors
   WHERE titleauthors.au_id=authors.au_id AND
         titleauthors.title_id=titles.title_id AND
         titles.pub_id=publishers.pub_id AND
         titles.yearpub > 1996;
```

4.6.8.DML: Вычисления внутри SELECT.

SQL позволяет выполнять различные арифметические операции над столбцами результирующего отношения. В конструкции <список_выбора> можно использовать константы, функции и их комбинации с арифметическими операциями и скобками. Например, чтобы узнать сколько лет прошло с 1992 года (год принятия стандарта SQL-92) до публикации той или иной книги можно выполнить команду:

```
SELECT title, yearpub-1992 FROM titles WHERE yearpub > 1992;
```

В арифметических вражения допускаются операции сложения (+), вычитания (-), деления (/), умножения (*), а также различные функции (COS, SIN, ABS - абсолютное значение и т.д.). Также в запрос можно добавить строковую константу:

```
SELECT 'the title of the book is', title, yearpub-1992
       FROM titles WHERE yearpub > 1992;
```

В SQL также определены так называемые агрегатные функции, которые совершают действия над совокупностью одинаковых полей в группе записей. Среди них:

- **AVG(<имя поля>)** - среднее по всем значениям данного поля
- **COUNT(<имя поля>)** или **COUNT (*)** - число записей
- **MAX(<имя поля>)** - максимальное из всех значений данного поля
- **MIN(<имя поля>)** - минимальное из всех значений данного поля
- **SUM(<имя поля>)** - сумма всех значений данного поля

Следует учитывать, что каждая агрегирующая функция возвращает единственное значение. Примеры: определить дату публикации самой "древней" книги в нашей базе данных

```
SELECT MIN(yearpub) FROM titles;
```

подсчитать количество книг в нашей базе данных:

```
SELECT COUNT(*) FROM titles;
```

Область действия данных функции можно ограничить с помощью логического условия. Например, количество книг, в названии которых есть слово "SQL":

```
SELECT COUNT(*) FROM titles WHERE title LIKE '%SQL%';
```

4.6.9.DML: Групировка данных.

Группировка данных в операторе SELECT осуществляется с помощью ключевого слова GROUP BY и ключевого слова HAVING, с помощью которого задаются условия разбиения записей на группы.

GROUP BY неразрывно связано с агрегирующими функциями, без них оно практически не используется. GROUP BY разделяет таблицу на группы, а агрегирующая функция вычисляет для каждой из них итоговое значение. Определим для примера количество книг каждого издательства в нашей базе данных:

```
SELECT publishers.publisher, count(titles.title)
    FROM titles,publishers
    WHERE titles.pub_id=publishers.pub_id
    GROUP BY publisher;
```

Ключевое слово HAVING работает следующим образом: сначала GROUP BY разбивает строки на группы, затем на полученные наборы накладываются условия HAVING. Например, устраним из предыдущего запроса те издательства, которые имеют только одну книгу:

```
SELECT publishers.publisher, count(titles.title)
   FROM titles,publishers
   WHERE titles.pub_id=publishers.pub_id
   GROUP BY publisher
     HAVING COUNT(*)>1;
```

Другой вариант использования HAVING - включить в результат только те издательства, название которых оканчивается на подстроку "Press":

```
SELECT publishers.publisher, count(titles.title)
     FROM titles,publishers
     WHERE titles.pub_id=publishers.pub_id
     GROUP BY publisher
        HAVING publisher LIKE '%Press';
```

В чем различие между двумя этими вариантами использования HAVING? Во втором варианте условие отбора записей мы могли поместить в раздел ключевого слова WHERE, в первом же варианте этого сделать не удастся, поскольку WHERE не допускает использования агрегирующих функций.

4.6.10.DML: Сортировка данных.

Для сортировки данных, получаемых при помощи оператора SELECT служит ключевое слово ORDER BY. С его помощью можно сортировать результаты по любому столбцу или выражению, указанному в <списке_выбора>. Данные могут быть упорядочены как по возрастанию, так и по убыванию. Пример: сортировать список авторов по алфавиту:

```
SELECT author FROM authors ORDER BY author;
```

Более сложный пример: получить список авторов, отсортированный по алфавиту, и список их публикаций, причем для каждого автора список книг сортируется по времени издания в обратном порядке (т.е. сначала более "свежие" книги, затем все более "древние"):

```
 SELECT
authors.author,titles.title,titles.yearpub,publishers.publisher
       FROM authors,titles,publishers,titleauthors
       WHERE titleauthors.au_id=authors.au_id AND
             titleauthors.title_id=titles.title_id AND
              titles.pub_id=publishers.pub_id
       ORDER BY authors.author ASC, titles.yearpub DESC;
```

Ключевое слово DESC задает здесь обратный порядок сортировки по полю **yearpub**, ключевое слов ASC (его можно опускать) - прямой порядок сортировки по полю **author.**

4.6.11.DML: Операция объединения.

В SQL предусмотрена возможность выполнения операции реляционной алгебры "ОБЪЕДИНЕНИЕ" (UNION) над отношениями, являющимися результатами оператора SELECT. Естественно, эти отношения должны быть определены по одной схеме.Пример: получить все Интеренет-ссылки, хранимые в базе данных **publications**. Эти ссылки хранятся в таблицах **publishers** и **wwwsites**. Для того, чтобы получить их в одной таблице, мы должны построить следующие запрос:

```
SELECT publisher,url FROM publishers
UNION
SELECT site,url FROM wwwsites;
```

4.6.12.Использование представлений.

До сих пор мы говорили о таблицах, которые *реально* хранятся в базе данных. Это, так называемые, базовые таблицы (base tables). Существует другой вид таблиц, получивший название "представления" (иногда их называют"представляемые таблицы").

Определение:
Представление (view) - это таблица, содержимое которой берется из других таблиц посредством запроса. При этом новые копии данных не создаются

Когда содержимое базовых таблиц меняется, СУБД автоматически перевыполняет запросы, создающие view, что приводит к соответствующи изменениям в представлениях.

Представление определяется с помощью команды

```
CREATE VIEW <имя_представления> [<имя_столбца>,...]
       AS <запрос>
```

При этом должны соблюдаться следующие ограничения:

- представление должно базироваться на единственном запросе (`UNION` не допустимо)
- выходные данные запроса, формирующего представление, должны быть не упорядочены (`ORDER BY` не допустимо)

Создадим представление, хранящее информацию об авторах, их книгах и издателях этих книг:

```
 CREATE VIEW books AS
 SELECT
authors.author,titles.title,titles.yearpub,publishers.publisher
   FROM authors,titles,publishers,titleauthors
   WHERE titleauthors.au_id=authors.au_id AND
         titleauthors.title_id=titles.title_id AND
         titles.pub_id=publishers.pub_id
```

Теперь любой пользователь, чьих прав на доступ к данному представлению достаточно, может осуществлять выборку данных из **books**. Например:

```
SELECT titles FROM books WHERE author LIKE '%Date'

SELECT author,count(title) FROM books GROUP BY author
```

(Права пользователей на доступ в представлениям назначаются также с помощью команд GRANT / REVOKE.)

Из приведенного выше примера достаточно ясен смысл использования представлений. Если запросы типа "выбрать все книги данного автора с указанием издательств" выполняются достаточно часто, то создание представляемой таблицы **books** значительно сократит накладные расходы на выполнение соединеия четырех базовых таблиц **authors**, **titles**, **publishers** и **titleauthors**. Кроме того, в представлении может быть представлена информация, явно не хранимая ни в одной из базовых таблиц. Например, один из столбцов представления может быть вычисляемым:

```
CREATE VIEW amount (publisher, books_count) AS
SELECT publishers.publisher, count(titles.title)
    FROM titles,publishers
    WHERE titles.pub_id=publishers.pub_id
    GROUP BY publisher;
```

Здесь использована еще одна, ранее не описанная, возможность SQL - присвоение новых имен столбцам представления. В приведенном примере число изданий, осуществленных каждым издатетлем, будет хранится в столбце с именем **books_count**. Заметим, что если мы хотим присвоить новые имена столбцам представления, нужно указывать имена для *всех* столбцов. Тип данных столбца представления и его нулевой статус всегда зависят от того, как он был определен в базовой таблице (таблицах).

Запрос на выборку данных к представлению выглядит абсолютно аналогично запросу к любой другой таблице. Однако на изменение данных в представлении накладываются ограничения. Кратко о них можно сказать следующее:

- Если представление основано на одной таблице, изменения данных в нем допускаются. При этом изменяются данные в связанной с ним таблице.
- Если представление основано более чем на одной таблице, то изменения данных в нем не допускаются, т.к. в большинстве случаев СУБД не может правильно восстановить схему базовых таблиц из схемы представления.

Удаление представления производится с помощью оператора:

```
DROP VIEW <имя_представления>
```

4.6.13.Другие возможности SQL.

Описываемые ниже возможности пока не стандартизованы, но представлены в той или иной мере практически во всех современных СУБД.

- **Хранимые процедуры.** Практический опыт создания приложений обработки данных показывает, что ряд операций над данными, реализующих общую для всех пользователей логику и не связанных с пользовательским интерфейсом, целесообразно вынести на сервер. Однако, для написания процедур, реализующих эти операции стандартных возможностей SQL не достаточно, поскольку здесь необходимы операторы обработки ветвлений, циклов и т.д. Поэтому многие поставщики СУБД предлагают собственные **процедурные** расширения SQL (PL/SQL компании Oracle и т.д.). Эти расширения содержат логические операторы (IF ... THEN ... ELSE), операторы перехода по условию (SWITCH ... CASE ...), операторы циклов (FOR, WHILE, UNTIL) и

операторы предачи управления в процедуры (CALL, RETURN). С помощью этих средств создаются функциональные модули, которые хранятся на сервере вместе с базой данных. Обычно такие модули называют *хранимые процедуры*. Они могут быть вызваны с передачей параметров любым пользователем, имеющим на то соотвествующие права. В некоторых системах хранимые процедуры могут быть реализованы и в виде внешних по отношению к СУБД модулей на языках общего назначения, таких как *C* или *Pascal*. Пример для СУБД PostgreSQL:

```
CREATE FUNCTION <имя_функции>
([<тип_параметра1>,...<тип_параметра2>])
RETURNS <возвращаемые_типы>
AS [ <SQL_оператор> | <имя_объектного_модуля> ]
LANGUAGE 'SQL' | 'C' | 'internal'
```

Вызов созданной функции осуществялется из оператора SELECT (также, как вызываются функции агрегирования). Более подробно о хранимых процедурах см. статью Э.Айзенберга Новый стандарт хранимых процедур в языке SQL, СУБД N 5-6, 1996 г.

- **Триггеры**. Для каждой таблицы может быть назначена хранимая процедура без параметров, которая вызывается при выполнении оператора модификации этой таблицы (INSERT, UPDATE, DELETE). Такие хранимые процедуры получили название триггеров. Триггеры выполняются автоматически, независимо от того, что именно является причиной модификации данных - действия человека оператора или прикладной программы. "Усредненный" синтаксис оператора создания триггера:

```
CREATE TRIGGER <имя_триггера>
ON <имя_таблицы>
FOR { INSERT | UPDATE | DELETE }
[, INSERT | UPDATE | DELETE ] ...
AS <SQL_оператор>
```

Ключевое слово `ON` задает имя таблицы, для которой определяется триггер, ключевое слово `FOR` указывает какая команда (команды) модификации данных активирует триггер. Операторы `SQL` после ключевого слова `AS` описывают действия, которые выполняет триггер и условия выполнения этих действий. Здесь может быть перечислено любое число операторов `SQL`, вызовов хранимых процедур и т.д. Использование триггеров очень удобно для выполнения операций контроля ограничений целостности (см. главу 4.3).

- **Мониторы событий**. Ряд СУБД допускает создание таких хранимых процедур, которые непрерывно сканируют одну или несколько таблиц на предмет обнаружения тех или иных событий (например, среднее значение какого-либо столбца достигает заданного предела). В случае наступления события может быть инициирован запуск триггера, хранимой процедуры, внешнего модуля и т.п. Пример: пусть наша база данных является частью автоматизированной системы управления технологическим процессом. В поле одной из таблиц заносятся показания датчика температуры, установленного на резце токарного станка. Когда это значение превышает заданный предел, запускается внешняя программа, изменяющая параметры работы станка.

Вопросы практического программирования.

В этой главе рассматриваются некоторые способы создания приложений, работающих с базой данных при помощи языка SQL. Как правило, любой поставщик СУБД предоставляет вместе со своей системой внешнюю утилиту, которая позволяет вводить операторы SQL в режиме командной строки и выдает на консоль результаты их выполнения (так, как это сделано на этой страничке, предоставляющей интерактивный доступ к БД publications). Недостатки такого режима работы очевидны: необходимо знать SQL, необходимо помнить схему БД, отсутствует возможность удобного просмотра результатов выполнения запросов. Поэтому, подобные утилиты стали инструментами администраторов баз данных, а для создания пользовательских приложений используются универсальные и специализированные языки программирования. Приложения, написанные таким образом, позволяют пользователю сосредоточиться на решении собственных задач, а не на структурах данных.

Почти все способы организации взаимодействия пользователя с базой данных, рассматриваемые ниже, основаны на модели "клиент-сервер". Т.е. предполагается, что каждое приложение обработки данных разбито, как минимум, на две части:

- клиента, который отвечает за организацию пользовательского интерфейса
- сервер, который собственно хранит данные, обрабатывает запросы и посылает их результаты клиенту для отображения

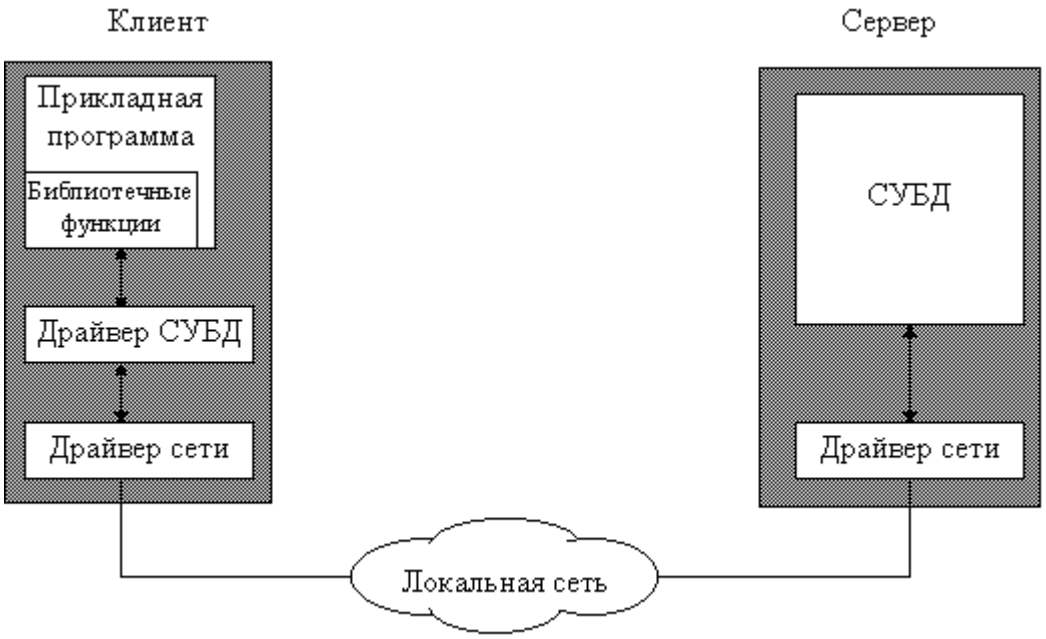
При этом предполагется, что каждая часть приложения функционирует на отдельном компьютере, т.е. к выделенному серверу БД с помощью локальной сети подключены персональные компьютеры пользователей (клиенты). Это наиболее популярная сегодня схема организации вычислительной среды. Более подробно архитектура "клиент-сервер" и различные способы ее реализации будут обсуждаться в главе 7.

Язык SQL позволяет только манипулировать данными, но в нем отсутствуют средства создания экранного интерфейса, что необходимо для пользовательских приложений. Для создания этого интерфейса служат универсальные языки третьего поколения (C, C++, Pascal) или проблемно-ориентированные языки четвертого поколения (xBase, Informix 4Gl, Progress, Jam,...). Эти языки содержат необходимые операторы ввода / вывода на экран, а также операторы структурного программирования (цикла, ветвтеления и т.д.). Также эти языки допускают определение структур, соответствующих записям таблиц обрабатываемой базы данных. В исходный текст программы включаются операторы языка SQL, которые во время исполнения передаются серверу БД, который собственно и производит манипулирование данными. Отношения, полученные в результате выполнения сервером SQL-запросов, возвращаются прикладной программе, которая заполняет строками этих отношений заранее определенные структуры. Дальнейшая работа клиентской программы (отображение, корректировка записей) ведется с этими структурами.

Рассмотрим различные способы орагнизации доступа прикладной программы к серверу базы данных.

4.7.1.Использование специализированных библиотек и встраиваемого SQL.

Каждая СУБД помимо интерактивной SQL-утилиты обязательно имеет библиотеку доступа и набор драйверов для различных операционных систем. Схема взаимодействия клиентского приложения с сервером базы данных в этом случае выглядит так:



Библиотека доступа - это, как правило, объектный файл, исходный код которого создан на универсальном языке типа *C*. Эта библиотека содержит набор функций, позволяющих пользовательскому приложению соединятся с базой данных, передавать запросы серверу и получать ответные данные. Типичный набор функций такой библиотеки (имена функций зависят от используемой библиотеки):

- **DB_connect(char *имя_базы_данных, char *имя_пользователя, char *пароль)** - устанавливает соединение с базой данной, возвращает указатель на структуру **db**, описывающую характеристики этого соединения
- **DB_exec(db, char *запрос)** - выполнить запрос к базе данных, определяемой структурой **db**. Применяется для любых запросов кроме SELECT. Возвращает код выполнения запроса (0 - удачно, либо код ошибки)
- **DB_select(db, char *запрос)** - выполнить запрос на извлечение данных (SELECT). Возвращает структуру **result**, содержащую результаты выполнения запроса (реляционное отношение).
- **DB_fetch(result)** - извлечь следующую запись из структуры **result**.
- **DB_close(db)** - закрыть соединение с базой данных.

Разумеется это минимальный набор функций для работы с базой данных. Обычно в библиотеке присутствуют также функции, позволяющие определить характеристики структуры **result** (число, порядок и имена столбцов, число строк, номер текущей строки), передвигаться по этой структуре не только вперед, но и назад (**DB_next**, **DB_prev**) и т.д. Пример программы, использующей библиотеку связи с базой данных:

```
    #include <dblib.h>        /* Файл, содержащий описание функций
библиотеки */
    ........
    /* Организация интерфейса с пользователем, запрос его имени и
пароля */
    /* Присвоение значений переменным:   dbname - имя базы данных
*/
    /*                                   username - имя пользователя
*/
```

```
    /*                                                  password - пароль
*/
    .........
    db=DB_connect(dbname,username,password);      /*      Установление
соединения */
    if (db == NULL) {
         error_message();   /* Выдача сообщения об ошибке на монитор
пользователя */
         exit(1);          /* Завершение работы */
    }
    ..........
    /* Ожидание запроса пользователя. Формирование строки s_query -
запроса */
    /*                на                выборку                данных
*/
    ..........
    result=DB_select(db,s_query);      /* Пересылка запроса на сервер
*/
    if (result==NULL) {
          error_message();    /*  Ошибка  выполнения  запроса.  Выдача
сообщения */
          exit(2);          /* Завершение работы */
    }
    ..........
    /*  Вывод  результатов  запроса  на  монитор  пользователя.  Ожидание
следующего  */
    /*  запроса.  Подготовка  строки  u_query="UPDATE  ...  SET  ...",
содержащей     */
    /*          запрос          на          изменение          данных.
*/
    ..........
    res=DB_exec(db,u_query);       /* Пересылка запроса на сервер */
    if (res != 0 ) {
          error_message();    /*  Ошибка  выполнения  запроса.  Выдача
сообщения */
          exit(2);          /* Завершение работы */
    }
    ..........

    ..........
    DB_close(db);         /*Завершение работы */
```

Данная программа, обеспечивающая взаимодействие пользователя с СУБД, компилируется совместно с библиотекой доступа. Библиотечные вызовы преобразуются драйвером базы данных в сетевые вызовы и передаются сетевым программным обеспечением на сервер.

На сервере происходит обратный процесс преобразования: сетевые пакеты -> функции библиотеки -> SQL-запросы, запросы обрабатываются, их результаты передаются клиенту.

Как видим, такой способ создания приложений чрезвычайно гибок, позволяет реализовать практически любое приложение, но в то же время имеет явные недостатки:

- разработка клиентской программы возможна только для той операционной системы и на том языке программирования, который поддерживатеся библиотекой

- необходим драйвер базы данных, который определяет допустимые типы сетевых интерфейсов
- большой объем кодирования
- нестандартизованные библиотечные функции.

В результате получаем приложение, которое привязано как к сетевой среде, так и к программно-аппаратной платформе и используемой базе данных.

Некоторой модификацией данного способа является использование "встроенного" языка SQL. В этом случае в текст программы на языке третьего поколения включаются не вызовы библиотек, а непосредственно предложения SQL, которые предваряются ключевым выражением "EXEC SQL". Перед компиляцией в машинный код такая программа обрабатывается препроцессором, который транслирует смесь операторов "собственного" языка СУБД и операторов SQL в "чистый" исходный код. Затем коды SQL замещаются вызовами соответствующих процедур из библиотек исполняемых модулей, служащих для поддержки конкретного варианта СУБД.

Такой подход позволил несколько снизить степень привязанности к СУБД, например, при переключении прикладной программы на работу с другим сервером базы данных достаточно было заново обработать ее исходный текст новым препроцессором и перекомпилировать.

4.7.2.CLI - интерфейс уровня вызовов.

Большим достижением явилось появление (1994 г.) в стандарте SQL интерфейса уровня вызова - CLI (Call Level Interface), в котором стандартизован общий набор рабочих процедур, обеспечивающий совместимость со всеми основными серверами баз данных. Ключевой элемент CLI - специальная библиотека для компьютера-клиента, в которой хранятся вызовы процедур и большинство часто используемых сетевых компонентов для организации связи с сервером. Это ПО поставляется разработчиком средств SQL, не является универсальным и поддерживает разнообразные транспортные протоколы.

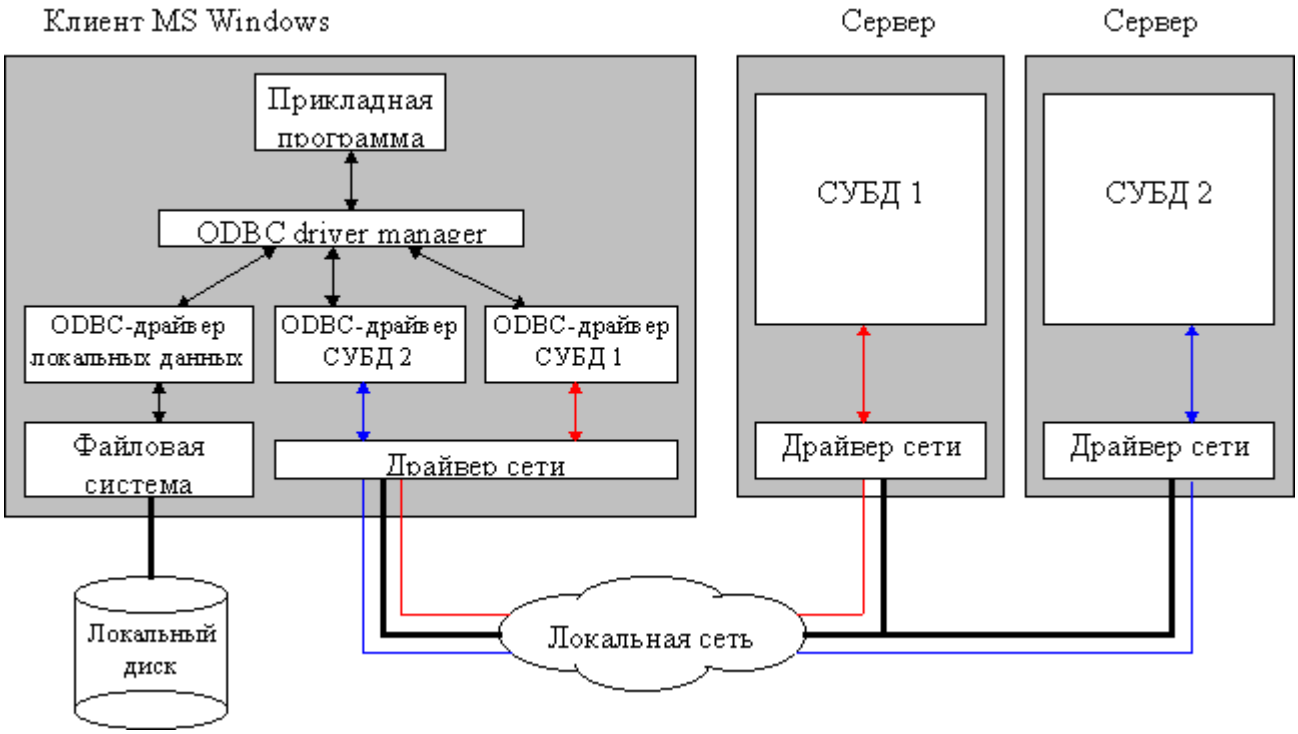
Использование программных вызовов позволяет свести к минимуму операции на компьютере-клиенте. В общем случае клиент формирует оператор языка SQL в виде строки и пересылает ее на сервер посредством процедуры исполнения (execute). Когда же сервер в качестве ответа возвращает несколько строк данных, клиент считывает результат с помощью серии вызовов процедуры выборки данных. Далее информация из столбцов полученной таблицы может быть связана с соответствующими переменными приложения. Вызов специальной процедуры позволяет клиенту определить считанное число строк, столбцов и типы данных в каждом столбце.

Интерфейс CLI построен таким образом, что перед передачей запроса серверу клиент не должен заботиться о типе оператора SQL, будь то выборка, обновление, удаление или вставка.

4.7.3.ODBC - открытый интерфейс к базам данных на платформе MS WIndows.

Очень важный шаг к созданию переносимых приложений обработки данных сделала фирма Microsoft, опубликовавшая в 1992 году спецификацию ODBC (Open Database Connetcivity - открытого интерфейса к базам данных), предназначенную для унификации доступа к данным с персональных компьютеров работающих под управлением операционной системы Windows.

(Заметим, что ODBC опирается на спецификации CLI). Структурная схема доступа к данным с использованием ODBC:

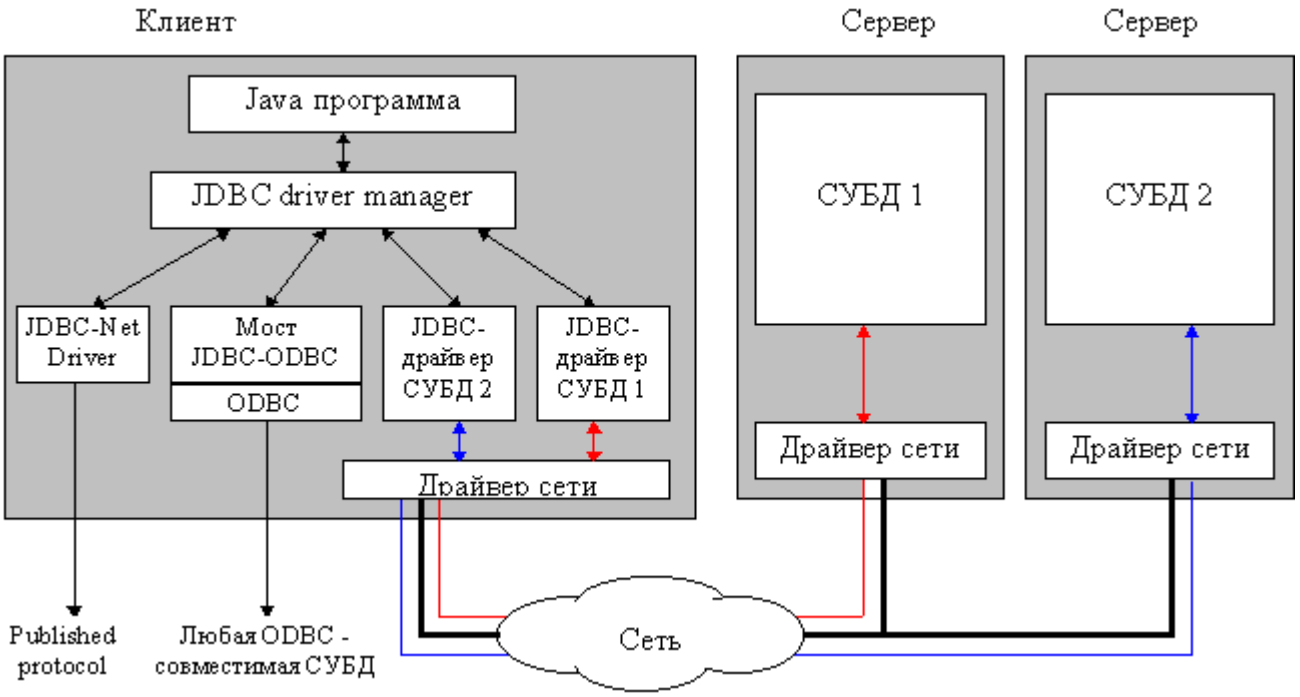


ODBC представляет из себя программный слой, унифицирующий интерфейс приложений с базами данных. За релизацию особенностей доступа к каждой отдельной СУБД отвечает специальный ODBC-драйвер. Пользовательское приложение этих особенностей не видит, т.к. взаимодействует с универсальным программным слоем более высокого уровня. Таким образом, приложение становится в значительной степени независимым от СУБД. Однако, этот способ также не лишен недостатков:

- приложения становятся привязанными к платформе MS Windows
- увеличивается время обработки запросов (как следствие введения дополнительного программного слоя)
- необходимо предварительная инсталляция ODBC-драйвера и настройка ODBC (указание драйвера, сетевого пути к серверу, базы данных и т.д.) на каждом рабочем месте. Параметры этой настройки являются статическими, т.е. приложение их самостоятельно изменить не может.

4.7.4.JDBC - мобильный интерфейс к базам данных на платформе Java.

JDBC (Java DataBase Connectivity) - это интерфейс прикладного программирования (API) для выполнения SQL-запросов к базам данных из программ, написанных на языке Java. Напомним, что язык Java, созданный компанией Sun, является платформенно - независимым и позволяет создавать как собственно приложения (standalone application), так и программы (апплеты), встраиваемые в web-страницы. Более подробная информация о Java и связанных с ним технологиях находится на серверах java.sun.ru.



JDBC во многом подобен ODBC (см. рисунок), также построен на основе спецификации CLI, однако имеет ряд замечательных отличий. Во-первых, приложение загружает JDBC-драйвер динамически, следовательно администрирование клиентов упрощается, более того, появляется возможность переключаться на работу с другой СУБД без перенастройки клиентского рабочего места. Во-вторых, JDBC, как и Java в целом, не привязан к конкретной аппаратной платформе, следовательно проблемы с переносимостью приложений практически снимаются. В-третьих, использование Java-приложений и связанной с ними идеологии "тонких клиентов" обещает снизить требования к оборудованию клиентских рабочих мест.

Навигационный подход к манипулированию данными и персональные СУБД.

Обсуждая вопрос практического написания приложений, взаимодействующих с реляционными базами данных (см. раздел 4.7.1), мы обнаружили один любопытный факт: для пользовательского приложения очень важна обработка не реляционного отношения в целом, как множества неупорядоченных строк, а именно каждой отдельной строки. Для этого служат функции **DB_next**, **DB_prev** и прочие, позволяющие перемещаться по строкам. При этом возможен доступ к значениям полей только одной строки, той которая является "текущей". Такой подход к манипулированию данными, при котором пользователь (или его программа) явно различает "предыдущие" и "последующие" строки и может управлять перемещением указателя от одной строки к другой, получил название навигационного (напомним, что навигационный подход типичен для сетевых и иерархических СУБД).

Ясно, что без навигационного подхода при создании клиентского приложения обойтись невозможно, в то же время для взаимодействия с сервером базы данных предлагается язык SQL как более эффективный. Однако, существует ряд СУБД, в которых навигационный подход распространен и на манипулирование хранимыми данными. Такие системы (dBase, FoxPro, Paradox) появились в начале 80-х годов и были предназначены для создания небольших однопользовательских приложений.

Рассмотрим кратко язык xBase, который поддерживается такими СУБД как dBase, FoxPro, Clipper и, возможно, некоторыми другими. Помимо собственно процедурных операторов (цикл,

условие и т.п.) он включает операторы создания пользовательского интерфейса (меню, окна, экранные формы,...). Для работы с данными служит следующий набор команд:

- **USE <имя_таблицы>** - открыть таблицу. При этом одна строка таблицы считается "текущей". К полям "текущей" строки можно обращаться по их именам.
- **GO [TOP | BOTTOM]** - сделать "текущей" первую / последнюю запись таблицы.
- **SKIP** - сделать текущей следующую запись.
- **REPLACE <имя_столбца> WITH <значение> [,<имя_столбца> WITH <значение> ...]** - изменить значения полей текущей записи

Существуют также команды экранного редактирования записей (**EDIT**, **BROWSE**), поиска данных в таблице (**FIND**) и другие. Приведем пример небольшой программы на языке xBase, которая распечатывает таблицу **authors**, а затем добаляет в нее новую строку:

```
USE AUTHORS                    /* Открыть таблицу authors */
GO TOP                         /* Перейти на первую запись */
DO WHILE .NOT.EOF()                 /* Выполнять цикл пока не будет
достигнут конец таблицы */
  PRINT AUTHOR                   /* Напечатать содержимое поля author
*/
  SKIP                           /* Перейти на следующую запись */
ENDDO                          /* Конец цикла */

APPEND BLANK           /* Добавить пустую запись. Она автоматически
становится текущей */
REPLACE AU_ID WITH 90, AUTHOR WITH "L.Pinter" /* Изменить значения
полей текущей записи */
```

Данные персональных СУБД хранятся в обычных файлах (как правило, каждая таблица в отдельном файле), средств описания ограничений целостности не существует. Следовательно, описание данных отделено от самих данных (находится в обрабатывающей программе), поэтому не гарантируется единый способ интерпретации данных для всех пользователей.

С развитием компьютерных сетей персональные СУБД стали переходить в разряд многопользовательских. При этом файлы данных размещались на разделяемом сетевом диске. Однако, создание достаточно больших приложений (10-20 одновременно работаюших пользователей) показало, что в этом случае резко снижается производительность и возникают проблемы с поддержанием целостности (точнее с изоляцией пользователей, подробнее см. следующий параграф). Поэтому, в настоящее время практически все персональные СУБД дополнены средствами доступа к SQL-серверам (как правило, с использованием ODBC). Теперь они могут служить не только средством для создания небольших локальных приложений, но и для разработки клиентских рабочих мест в архитектуре "клиент-сервер".

Транзакции, блокировки и многопользовательский доступ к данным.

Любая база данных годна к использованию только тогда, когда ее состояние соответствует состоянию предметной области. Такие состояния называют целостными. Очевидно, что при изменении данных БД должна переходить от одного целостного состояния к другому. Однако, в процессе обновления данных возможны ситуации, когда состояние целостности нарушается. Например:

- В банковской системе производится перевод денежных средств с одного счета на другой. На языке SQL эта операция описывается последовательностью двух команд UPDATE:
- ```
  UPDATE accounts SET summa=summa-1000 WHERE account="PC_1"
  UPDATE accounts SET summa=summa+1000 WHERE account="PC_2"
  ```

  Как видим, после выполнения первой команды и до завершения второй команды база данных не находится в целостном состоянии (искомая сумма списана с первого счета, но не зачислена на второй). Если в этот момент в системе произойдет сбой (например, выключение электропитания), то целостное состояние БД будет безвозвратно утеряно.

- Целостность БД может нарушаться и во время обработки одной команды SQL. Пусть выполняется операция увеличения зарплаты всех сотрудников фирмы на 20%:

  ```
  UPDATE employers SET salary=salary*1.2
  ```

  При этом СУБД последовательно обрабатывает все записи, подлежащие обновлению, т.е. существует временной интервал, когда часть записей содержит новые значения, а часть - старые.

Во избежание таких ситуаций в СУБД вводится понятие **транзакции** - атомарного действия над БД, переводящего ее из одного целостного состояния в другое целостное состояние. Другими словами, транзакция - это последовательность операций, которые должны быть или все выполнены или все не выполнены (все или ничего).

Методом контроля за транзакциями является ведение *журнала*, в котором фиксируются все изменения, совершаемые транзакцией в БД. Если во время обработки транзакции происходит сбой, транзакция откатывается - из журнала восстанавливаться состояние БД на момент начала транзакции.

В СУБД различных поставщиков начало транзакции может задаваться явно (например, командой **BEGIN TRANSACTION**), либо предполагаться неявным (так определено в стандарте SQL), т.е. очередная транзакция открывается автоматически сразу же после удачного или неудачного завершения предыдущей. Для завершения транзакции обычно используют команды SQL:

- **COMMIT** - успешно завершить транзакцию
- **ROLLBACK** - откатить транзакцию, т.е. вернуть БД в состояние, в котором она находилась на момент начала транзакции.

Стандарт SQL определяет, что транзакция начинается с первого SQL-оператора, инициируемого пользователем или содержащегося в прикладной программе. Все последующие SQL-операторы составляют тело транзакции. Транзакция завершается одним из возможных способов:

1. оператор **COMMIT** означает успешное завершение транзакции, все изменения, внесенные в базу данных делаются постоянными
2. оператор **ROLLBACK** прерывает транзакцию и отменяет все внесенные ею изменения
3. успешное заверешение программы, инициировавшей транзакцию, означает успешное завершение транзакции (как использование **COMMIT**)
  ```

4. ошибочное завершение программы прерывает транзакцию (как **ROLLBACK**)

Пример явно заданной транзакции:

```
        BEGIN TRANSACTION;              /* Начать транзакцию */
        DELETE ...;                     /* Изменения */
        UPDATE ...;                     /* данных    */
        if (обнаружена_ошибка) ROLLBACK;
        else COMMIT;                    /* Завершить транзакцию */
```

Пример неявно заданной транзакции:

```
        COMMIT;                          /* Окончание предыдущей
транзакции */
        DELETE ...;                     /* Изменения */
        UPDATE ...;                     /* данных    */
        if (обнаружена_ошибка) ROLLBACK;
        else COMMIT;                    /* Завершить транзакцию */
```

К сожалению, описанный механизм транзакций гарантирует обеспечение целостного состояния базы данных только в том случае, когда все транзакции выполняются последовательно, т.е. в каждую единицу времени активна только одна транзакция. Если работу с данными ведут одновременно несколько пользователей, вряд ли их устроит такой способ организации обработки запросов, т.к. это приведет к увеличению времени реакции системы. В то же время, если одновременно выполняются две транзакции, могут возникнуть следующие ошибочные ситуации:

- **Грязное чтение** (Dirty Read) - транзакция T1 модифицировала некий элемент данных. После этого другая транзакция T2 прочитала содержимое этого элемента данных до завершения транзакции T1. Если T1 заврешается операцией `ROLLBACK`, то получается, что транзакция T2 прочитала не существующие данные.
- **Неповторяемое (размытое) чтение** (Non-repeatable or Fuzzy Read) - транзакция T1 прочитала содержимое элемента данных. После этого другая транзакция T2 модифицировала или удалила этот элемент. Если T1 прочитает содержимое этого элемента занова, то она получит другое значение или обнаружит, что элемент данных больше не существует.
- **Фантом (фиктивные элементы)** (Phantom) - транзакция T1 прочитала содержимое нескольких элементов данных, удовлетворяющих некому условию. После этого T2 создала элемент данных, удовлетворяющий этому условию и зафиксировалась. Если T1 повторит чтение с тем же условием, она получит другой набор данных.

Как уже было сказано, ни одна из этих ситуаций не может возникнуть при последовательном выполнении транзакций. Отсюда возникло понятие **сериализуемости** (способности к упорядочению) параллельной обработки транзакций. Т.е. чередующееся (параллельное) выполнение заданного множества транзакций будет верным, если при его выполнении будет получен такой же результат, как и при *последовательном* выполнении тех же транзакций.

Все описанные выше ситуации возникли только потому, что чередующееся выполнение транзакций T1 и T2 не было упорядочено, т.е. не было эквивалентно выполнению сначала транзакции T1, а затем T2, либо, наоборот, сначала транзакции T2, а затем T1.

Принудительное упорядочение транзакций обеспечивается с помощью механизма **блокировок.** Суть этого механизма в следующем: если для выполнения некоторой транзакции необходимо, чтобы некоторый объект базы данных (кортеж, набор кортежей, отношение, набор отношений,..) не изменялся непредсказуемо и без ведома этой транзакции, такой объект блокируется. Основными видами блокировок являются:

- **блокировка со взаимным доступом**, называемая также *S-блокировкой* (от Shared locks) и *блокировкой по чтению*.
- **монопольная блокировка** (без взаимного доступа), называемая также *X-блокировкой* от (eXclusive locks) или *блокировкой по записи*. Этот режим используется при операциях изменения, добавления и удаления объектов.

При этом:

- если транзакция налагает на объект X-блокировку, то любой запрос другой транзакции с блокировкой этого объекта будет отвергнут.
- если транзакция налагает на объект S-блокировку, то
  - запрос со стороны другой транзакции с X-блокировокй на этот объект будет отвергнут
  - запрос со стороны другой транзакции с S-блокировокй этого объекта будет принят

Транзакция, запросившая доступ к объекту, уже захваченному другой транзакцией в несовместимом режиме, останавливается до тех пор, пока захват этого объекта не будет снят.

Доказано, что сериализуемость транзакций (или, иначе, их изоляция) обеспечивается при использовании двухфазного протокола блокировок (2LP - Two-Phase Locks), согласно которому все блокировки, произведенные транзакцией, снимаются только при ее завершении. Т.е выполение транзакции разбивается на две фазы: (1) - накопление блокировок, (2) - освобождение блокировок в результате фиксации или отката.

К сожалению, применение механизма блокировки приводит к замедлению обработки транзакций, поскольку система вынуждена ожидать пока освободятся данные, захваченные конкурирующей транзакцией. Решить эту проблему можно за счет уменьшения фрагментов данных, захватываемых транзакцией. В зависимости от захватываемых объектов различают несколько *уровней блокировки*:

- блокируется вся база данных - очевидно, этот вариант неприемлим, поскольку сводит многопользовательский режим работы к однопользовательскому
- блокируются отдельные таблицы
- блокируются страницы (страница - фрагмент таблицы размером обычно 2-4 Кб, единица выделения памяти для обработки данных системой)
- блокируются записи
- блокируются отдельные поля

Современные СУБД, как правило, могут осуществлять блокировку на уровне записей или страниц.

Язык SQL также предоставляет способ косвенного управления скоростью выполения транзакций с помощью указания *уровня изоляции* транзакции. Под уровнем изоляции транзакции понимается возможность возникновения одной из описанных выше ошибочных ситуаций. В стандарте SQL определены 4 уровня изоляции:

Уровень изоляции	**Грязное чтение**	**Размытое чтение**	**Фантом**
Незафиксированное чтение (READ UNCOMMITTED)	возможно	возможно	возможно
Зафиксированное чтение	невозможно	возможно	возможно

(READ COMMITED)			
Повторяемое чтение (REPEATABLE READ)	невозможно	невозможно	возможно
Сериализуемость (SERIALIZABLE)	невозможно	невозможно	невозможно

Для определения характеристик транзакции используется оператор

```
SET TRANSACTION <режим_доступа>, <уровень_изоляции>
```

Список уровней изоляции приведен в таблице. Режим доступа по умолчанию используется **READ WRITE** (чтение запись), если задан уровень изоляции READ UNCOMMITED, то режим доступа должен быть **READ ONLY** (только чтение).

Одним из наиболее серьезных недостатков метода сериализации транзакций на основе механизма блокировок является возможность возникновения тупиков (dead locks) между транзакциями. Пусть, например, транзакция T1 наложила монопольную блокировку на объект O1 и претендует на доступ к объекту O2, который уже монопольно заблокирован транзакцией T2, ожидающей доступа к объекту O1. В этом случае ни одна из транзакций продолжаться не может, следовательно, блокировки объектов O1 и O2 никогда не будут сняты. Естественного выхода из такой ситуации не существует, поэтому тупиковые ситуации обнаруживаются и устраняются искусственно. При этом СУБД откатывает одну из транзакций, попавших в тупик ("жертвует" ею), что дает возможность продолжить выполнение другой транзакции.

Вопрос обеспечения параллельного выполнения транзакций весьма сложен и многие моменты остались за рамками данного раздела. Заинтересованный читатель может найти более полное изложение этой темы в одном из источников, указанных в списке литературы.

Использования WWW - технологий для доступа к существующим базам данных

WWW - доступ к существующим базам данных может осуществляться по одному из трех основных сценариев. Ниже дается их краткое описание и основные характеристики.

Однократное или периодическое преобразование содержимого БД в статические документы

В этом варианте содержимое БД просматривает специальная программа, создающая множество файлов - связных HTML-документов (см.рис.1-2). Полученные файлы могут быть перенесены на один или несколько WWW-серверов. Доступ к ним будет осуществляться как к статическим гипертекстовым документам сервера.

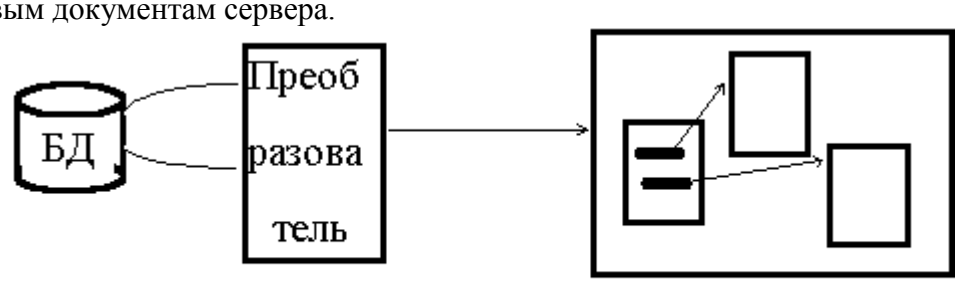


Рис. 1-2

Этот вариант характеризуется минимальными начальными расходами. Он эффективен на небольших массивах данных простой структуры и редким обновлением, а также при

пониженных требованиях к актуальности данных, предоставляемых через WWW. Кроме этого, очевидно полное отсутствие механизма поиска, хотя возможно развитое индексирование.
В качестве преобразователя может выступать программный комплекс, автоматически или полуавтоматически генерирующий статические документы. Программа-преобразователь может являться самостоятельно разработанной программой либо быть интегрированным средством класса генераторов отчетов.

Динамическое создание гипертекстовых документов на основе содержимого БД

В этом варианте доступ к БД осуществляется специальной CGI-программой, запускаемой WWW-сервером в ответ на запрос WWW - клиента. Эта программа, обрабатывая запрос, просматривает содержимое БД и создает выходной HTML-документ, возвращаемый клиенту (см.рис.1-3).

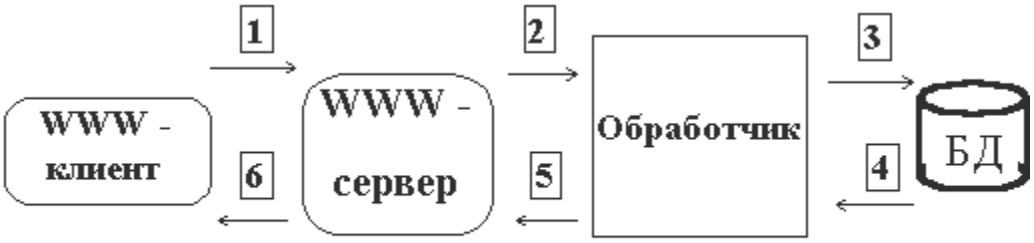


Рис. 1-3

Это решение эффективно для больших баз данных со сложной структурой и при необходимости поддержки операций поиска. Показаниями также являются частое обновление и невозможность синхронизации преобразования БД в статические документы с обновлением содержимого. В этом варианте возможно осуществлять изменение БД из WWW-интерфейсов.
К недостаткам этого метода можно отнести большое время обработки запросов, необходимость постоянного доступа к основной базе данных, дополнительную загрузку средств поддержки БД, связанную с обработкой запросов от WWW - сервера.
Для реализации такой технологии необходимо использовать взаимодействие WWW-сервера с запускаемыми программами CGI - *Common Gateway Interface*. Выбор программных средств достаточно широк - языки программирования, интегрированные средства типа генераторов отчетов. Для СУБД со внутренними языками программирования существуют варианты использования этого языка для генерации документов.

Создание информационного хранилища на основе высокопроизводительной СУБД с языком запросов SQL. Периодическая загрузка данных в хранилище из основных СУБД

В этом варианте предлагается использование технологии, получившей название "информационного хранилища" (ИХ). Для обработки разнообразных запросов, в том числе и от WWW-сервера, используется промежуточная БД высокой производительности (см. рис.1-5). Информационное наполнение промежуточной БД осуществляется специализированным программным обеспечением на основе содержимого основных баз данных (см. рис.1-4).

Этап 1 - перегрузка данных

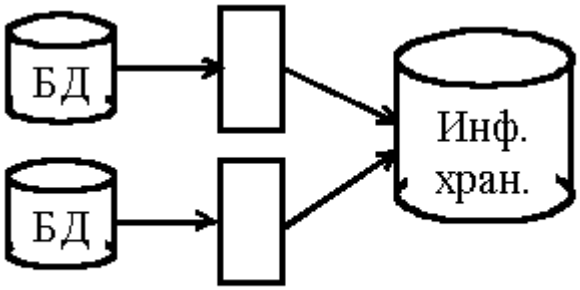


Рис. 1-4

Этап 2 - обработка запросов

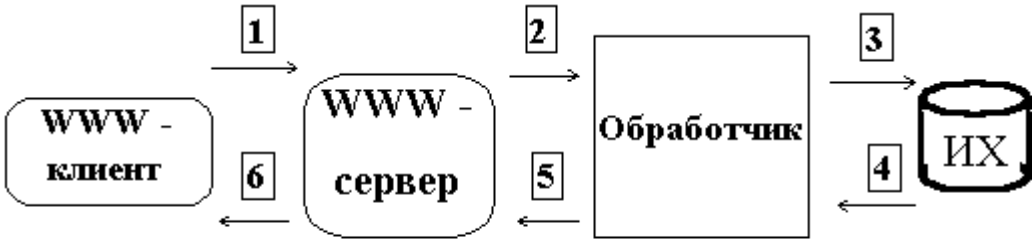


Рис. 1-5

Данный вариант свободен ото всех недостатков предыдущей схемы. Более того, после установления синхронизации данных информационного хранилища с основными БД возможен перенос пользовательских интерфейсов на информационное хранилище, что существенно повысит надежность и производительность, позволит организовать распределенные рабочие места.

Несмотря на кажущуюся громоздкость такой схемы, для задач обеспечения WWW-доступа к содержимому нескольких баз данных накладные расходы существенно уменьшаются.

Основой повышения производительности обработки WWW-запросов и резкого увеличения скорости разработки WWW-интерфейсов является использование внутренних языков СУБД информационного хранилища для создания гипертекстовых документов.

Для загрузки содержимого основной БД в информационное хранилище могут использоваться все перечисленные решения (языки программирования, интегрированные средства), а также специализированные средства перегрузки, поставляемые с SQL-сервером и продукты поддержки информационных хранилищ.