МИНИСТЕРСТВО ПО РАЗВИТИЮ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И КОММУНИКАЦИЙ РЕСПУБЛИКИ УЗБЕКИСТАН

МИНИСТЕРСТВО ВЫСШЕГО И СРЕДНЕ-СПЕЦИАЛЬНОГО ОБРАЗОВАНИЯ РЕСПУБЛИКИ УЗБЕКИСТАН

ТАШКЕНТСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ ИМЕНИ МУХАММАДА АЛ-ХОРЕЗМИ

Т.А.ХУЖАКУЛОВ

“СИСТЕМЫ УПРАВЛЕНИЯ БАЗАМИ ДАННЫХ”

УЧЕБНИК

Ташкент

УЎК: 083
К 12
КВК: 08.981
Автор: Т.А. Хужакулов / ТУИТ, Ташкент, 2018 г.

“Маълумотлар базасини бошқариш тизимлари ” фани бўйича дарслик Муҳаммад Ал – Хоразмий номидаги Тошкент ахборот технологиялари университетининг “Ахборот технологиялари” кафедраси ўқитувчиси томонидан тайёрланган бўлиб, унда маълумотлар базаси ҳақида тушунча; маълумотлар базаси архитектураси; моҳият алоқа моделли; релацион модел; релацион алгебра тиллари; нормалаштириш формалари; SQL тили тушунчаси; маълумотларни манипуляциялаш тили; транзакцияларни бошқариш; маълумотлар базасини администраторлаш; ODBC интерфейси ҳамда XML ва маълумотлар базасини принциплари муҳокама этилиб, маълумотлар базасини бошқариш тизимлари меъёрий ҳужжатлари келтирилган.

Дарслик олий ўқув юртининг 5350300 – Ахборот хавфсизлиги (ахборот, коммуникация технологиялари ва сервис) факультети таълим йўналишлари талабалари учун мўлжалланган бўлиб, унда ахборот технологиялари соҳасида фаолият кўрсатувчилар фойдаланишлари мумкин.

Учебник по предмету “Системы управления базами данных” разработан предподователем кафедры “Информационные технологии” Ташкентского университета информационных технологий имени Мухаммада Аль-Хорезми, в нем даются понятия об информационной базе; архитектура базы данных; о релационной модель; о языках реляционной алгебры;о формах нормализации; о понятии SQL языка; о языке манипуляции информации; об управление транзакциями; об администрации информационной базы; об интерфейс ODBC, а также обсуждены принципы XML и информационной базы, приведены нормативные документы управлениея информационной базамы.

Учебник предназначен для студентов направления 5330300 – Информационная безопасность (информация, коммуникационная технологии и сервис) высших учебных заведений и может быть занимающимися с сферы информационных технологии.

The textbook on the subject "Database management systems" was developed by the peradventures of the "Information Technologies" department of the Tashkent University of Information Technologies named after Muhammad Al-Khorezmi, he gives the notion of an information base; database architecture; o relational model; on the languages of relational algebra, on the forms of normalization, about the concept of SQL language; about the language of information manipulation; about the management of transactions; on the administration of the information base; ODBC interface, as well as discussed the principles of XML and the information base, provides regulatory documents for managing the information database.

The textbook is prenaznachen for students direction 5330300 - Information security (information, communication technology and service) of higher education institutions and can be engaged in the sphere of information technology.

Рецензенты:

Турсунбоев Ф. –ТУИТ, д.т.н., профессор кафедры «Информационных технологии».

Гулямов Ш.М. – ТашГТУ, д.т.н., профессор кафедры «Автоматизации производственных процессор».

ОГЛАВЛЕНИЕ

Введение

Базы данных всегда были важнейшей темой при изучении информационных систем. Однако в последние годы всплеск популярности Интернета и бурное развитие новых технологий для Интернета сделали знание технологии баз данных для многих одним из актуальнейших путей карьеры. Технологии баз данных увели Интернет - приложения далеко от простых брошюрных публикаций, которые характеризовали ранние приложения. В то же время Интернет-технология обеспечивает пользователям стандартизированные и доступные средства публикации содержимого баз данных. Правда, ни одна из этих новых разработок не отменяет необходимости в классических приложениях баз данных, которые появились еще до развития Интернета для нужд бизнеса. Это только расширяет важность знания баз данных.

Многие студенты считают этот предмет приятным и интересным, даже несмотря на его сложность. Проектирование и разработка базы данных требуют и искусства, и умения. Понимание пользовательских требований и перевод их в эффективный проект базы данных можно назвать творческим процессом. Преобразование этих проектов в физические базы данных с помощью функционально полных и высокопроизводительных приложений — инженерный процесс. Оба процесса полны сложностей и приятных интеллектуальных головоломок. Поскольку сейчас существует большая необходимость в развитии технологии баз данных, навыки, которые вы разовьете, и знания, которые вы получите в процессе изучения этого курса, будут востребованы. Цель книги — предоставить твердое обоснование фундаментальных положений технологии баз данных, чтобы вы могли начать успешную карьеру в этой области, если вам этого захочется. В этой главе мы обсудим, что, как и почему в базах данных. Мы поймем, почему используются базы данных, расскажем, какие существуют компоненты системы базы данных и как разрабатывать такие системы. Глава завершится экскурсом в историю баз данных.

В силу все более широкого распространения персональных компьютеров важность организации информации в виде База данных баз данных непрерывно возрастает. Основные публикации ссылка на источники литературы учебного характера, освещающие понятия и основные принципы построения баз данных, относятся к 1983-1987 гг. Даже учебники по курсу "Базы и банки данных", изданные позднее ссылка на источники литературы, отражают фактически уровень этих лет.

Вместе с тем в области баз данных за последние десять лет произошли серьезные изменения, практически не нашедшие отражения в учебной литературе. Сюда относятся проблемы, связанные с распределенными, объектно-ориентированными и объектно-реляционными базами данных, средствами автоматизации проектирования и программирования баз данных, языком структурированных запросов SQL и многие другие.

Многочисленная литература ссылка на источники литературы по применяемым Система управления базы данныхСУБД и различным Windows-приложениям конкретизирована вплоть до кнопок, что затрудняет понимание не только самих программных средств, но принципов и правил их построения. Более того, в ней часто вводятся новые термины, порой не определяемые или имеющие разные названия в различных источниках. Много отдельных вопросов по базам данных опубликованы в труднодоступных журналах, прежде всего иностранных. Кроме того, в свете современного представления о базе данных трансформировался ряд понятий и положений о ней.

Указанные обстоятельства, особенно сильно проявляющиеся на длительных курсах лекций (свыше 100 часов), явились причиной написания данной работы.

Ее главной целью является систематизация и упорядочение имеющегося материала по базам данных с учетом специфики полиграфических производств. Основой работы является курс лекций, читаемый в СПИ МГУП на протяжении ряда последних лет.

В ней систематически изложено современное представление БД, включая такие новые и перспективные направления, как объектно-ориентированные и распределенные базы данных, автоматизация проектирования БД.

Работа состоит из трех частей, каждая из которых соответствует одно - семестровому курсу.

В первой части рассматриваются основные понятия, классификация, состав и порядок работы БД. Обсуждаются концепция и методология БД, излагается общая теория реляционных баз данных.

Во второй части изучаются характеристики иерархических, сетевых и реляционных моделей данных, вопросы интеграции моделей, методы физической реализации БД. Обсуждаются проблемы проектирования БД, способы его автоматизации. Рассматриваются недостатки База данных реляционнаяреляционных БД, необходимость и способы построения объектно-ориентированных БД.

Третья часть посвящена База данных распределеннаяраспределенным базам данных (РБД). Обсуждаются их специфика, архитектура, технология и система клиент-сервер. Рассматриваются фрагментация и локализация данных, интеграция локальных БД как этапы проектирования РБД. Изучаются особенности создания и использования РБД. Приводится системное изложение процесса проектирования РБД.

Глава I. Введение в базы данных

Обзор

Исторически сложившееся развитие вычислительных систем обусловило необходимость хранения в электронном (машиночитаемом) виде все большего количества информации. Одновременно с совершенствованием и дальнейшим развитием вычислительных систем росли объемы информации, подлежащей обработке и хранению. Сложности, возникшие при решении на практике задач структурированного хранения и эффективной обработки возрастающих объемов информации, стимулировали исследования в соответствующих областях. Задачи хранения и обработки данных были формализованы. Была создана теоретическая база для решения задач такого класса, результатом реализации на практике которой стали системы, предназначенные для организации обработки, хранения и предоставления доступа к информации. Позже такие системы стали называть системами баз данных.

Одновременно с развитием систем баз данных, происходило интенсивное развитие средств вычислительной техники, используемой для работы с большими объемами информации. Вычислительная мощность и, особенно, объемы запоминающих устройств первых вычислительных систем были недостаточны для хранения и обработки информации в объемах, необходимых на практике.

По мере развития систем баз данных, менялись принципы организации данных в них: первоначально данные представлялись на основе иерархической, а в последствии сетевой модели. В конце 1970-х – начале 1980-х годов начали появляться первые реляционные продукты. В настоящее время системы баз данных на основе реляционной модели занимают лидирующее положение, несмотря на заявления многих исследователей о скором переходе к объектно-ориентированным системам. В настоящее время объектно-ориентированные системы, тем не менее, развиваются, хотя темпы их развития и сдерживаются медленным принятием соответствующих стандартов. Кроме того, многие коммерческие реляционные системы приобретают объектно-ориентированные черты. На основании этого, можно предположить, что в будущем

объектно-ориентированные системы будут постепенно вытеснять реляционные.

В настоящее время ведутся исследования в следующих направлениях:

1. дедуктивные системы;
2. экспертные системы;
3. расширяемые системы;
4. объектно-ориентированные системы;
5. NoSQL.

1.1 История развития баз данных

История развития СУБД насчитывает более 30 лет. В 1968 году была введена в эксплуатацию первая промышленная СУБД система IMS фирмы IBM. В 1975 году появился первый стандарт ассоциации по языкам систем обработки данных — Conference of Data System Languages (CODASYL), который определил ряд фундаментальных понятий в теории систем баз данных, которые и до сих пор являются основополагающими для сетевой модели данных.

В дальнейшее развитие теории баз данных большой вклад был сделан американским математиком Э. Ф. Коддом, который является создателем реляционной модели данных. В 1981 году Э. Ф. Кодд получил за создание реляционной модели и реляционной алгебры престижную премию Тьюринга Американской ассоциации по вычислительной технике.

Менее двух десятков лет прошло с этого момента, но стремительное развитие вычислительной техники, изменение ее принципиальной роли в жизни общества, обрушившийся бум персональных ЭВМ и, наконец, появление мощных рабочих станций и сетей ЭВМ повлияло также и на развитие технологии баз данных. Можно выделить четыре этапа в развитии данного направления в обработке данных. Однако необходимо заметить, что все же нет жестких временных ограничений в этих этапах: они плавно переходят один в другой и даже сосуществуют параллельно, но тем не менее выделение этих этапов позволит более четко охарактеризовать отдельные стадии развития технологии баз данных, подчеркнуть особенности, специфичные для конкретного этапа.

Первый этап развития СУБД связан с организацией баз данных на больших машинах типа IBM 360/370, ЕС-ЭВМ и мини-ЭВМ типа PDP11 (фирмы Digital Equipment Corporation — DEC), разных моделях HP (фирмы Hewlett Packard).

Базы данных хранились во внешней памяти центральной ЭВМ, пользователями этих баз данных были задачи, запускаемые в основном в пакетном режиме. Интерактивный режим доступа обеспечивался с помощью консольных терминалов, которые не обладали собственными вычислительными ресурсами (процессором, внешней памятью) и служили только устройствами ввода-вывода для центральной ЭВМ. Программы доступа к БД писались на различных языках и запускались как обычные числовые программы. Мощные операционные системы обеспечивали возможность условно параллельного выполнения всего множества задач. Эти системы можно было отнести к системам распределенного доступа, потому что база данных была централизованной, хранилась на устройствах внешней памяти одной центральной ЭВМ, а доступ к ней поддерживался от многих пользователей-задач.

Особенности этого этапа развития выражаются в следующем:

- Все СУБД базируются на мощных мультизадачных операционных системах (MVS, SVM, RTE, OSRV, RSX, UNIX), поэтому в основном поддерживается работа с централизованной базой данных в режиме распределенного многопользовательского доступа.
- Функции управления распределением ресурсов в основном осуществляются операционной системой (ОС).
- Поддерживаются языки низкого уровня манипулирования данными, ориентированные на навигационные методы доступа к данным.
- Значительная роль отводится администрированию данных.
- Проводятся серьезные работы по обоснованию и формализации реляционной модели данных, и была создана первая система (System R), реализующая идеологию реляционной модели данных.
- Проводятся теоретические работы по оптимизации запросов и управлению распределенным доступом к централизованной БД, было введено понятие транзакции.

- Результаты научных исследований открыто обсуждаются в печати, идет мощный поток общедоступных публикаций, касающихся всех аспектов теории и практики баз данных, и результаты теоретических исследований активно внедряются в коммерческие СУБД.
- Появляются первые языки высокого уровня для работы с реляционной моделью данных. Однако отсутствуют стандарты для этих первых языков.

Персональные компьютеры стремительно ворвались в нашу жизнь и буквально перевернули наше представление о месте и роли вычислительной техники в жизни общества. Теперь компьютеры стали ближе и доступнее каждому пользователю. Исчез благоговейный страх рядовых пользователей перед непонятными и сложными языками программирования. Появилось множество программ, предназначенных для работы неподготовленных пользователей. Эти программы были просты в использовании и интуитивно понятны: это прежде всего различные редакторы текстов, электронные таблицы и другие. Простыми и понятными стали операции копирования файлов и перенос информации с одного компьютера на другой, распечатка текстов, таблиц и других документов. Системные программисты были отодвинуты на второй план. Каждый пользователь мог себя почувствовать полным хозяином этого мощного и удобного устройства, Позволяющего автоматизировать многие аспекты деятельности. И, конечно, это сказалось и на работе с базами данных. Появились программы, которые назывались системами управления базами данных и позволяли хранить значительные объемы информации, они имели удобный интерфейс для заполнения данных, встроенные средства для генерации различных отчетов. Эти программы позволяли автоматизировать многие учетные функции, которые раньше велись вручную. Постоянное снижение цен на персональные компьютеры сделало их доступными не только для организаций и фирм, но и для отдельных пользователей. Компьютеры стали инструментом для ведения документации и собственных учетных функций. Это все сыграло как положительную, так и отрицательную роль в области развития баз данных. Кажущаяся простота и доступность персональных компьютеров и их программного обеспечения

породила множество дилетантов. Эти разработчики, считая себя знатоками, стали проектировать недолговечные базы данных, которые не учитывали многих особенностей объектов реального мира. Много было создано систем-однодневок, которые не отвечали законам развития и взаимосвязи реальных объектов. Однако доступность персональных компьютеров заставила пользователей из многих областей знаний, которые ранее не применяли вычислительную технику в своей деятельности, обратиться к ним. И спрос на развитые удобные программы обработки данных заставлял поставщиков программного обеспечения поставлять все новые системы, которые принято называть настольными (desktop) СУБД. Значительная конкуренция среди поставщиков заставляла совершенствовать эти системы, предлагая новые возможности, улучшая интерфейс и быстродействие систем, снижая их стоимость. Наличие на рынке большого числа СУБД, выполняющих сходные функции, потребовало разработки методов экспорта - импорта данных для этих систем и открытия форматов хранения данных.

Но и в этот период появлялись любители, которые вопреки здравому смыслу разрабатывали собственные СУБД, используя стандартные языки программирования. Это был тупиковый вариант, потому что дальнейшее развитие показало, что перенести данные из нестандартных форматов в новые СУБД было гораздо труднее, а в некоторых случаях требовало таких трудозатрат, что легче было бы все разработать заново, но данные все равно надо было переносить на новую более перспективную СУБД. И это тоже было результатом недооценки тех функций, которые должна была выполнять СУБД.

Особенности этого этапа следующие:

- Все СУБД были рассчитаны на создание БД в основном с монопольным доступом. И это понятно. Компьютер персональный, он не был подсоединен к сети, и база данных на нем создавалась для работы одного пользователя. В редких случаях предполагалась последовательная работа нескольких пользователей, например, сначала оператор, который вводил бухгалтерские документы, а потом главбух, который

определял проводки, соответствующие первичным документам.

- Большинство СУБД имели развитый и удобный пользовательский интерфейс. В большинстве существовал интерактивный режим работы с БД как в рамках описания БД, так и в рамках проектирования запросов. Кроме того, большинство СУБД предлагали развитый и удобный инструментарий для разработки готовых приложений без программирования. Инструментальная среда состояла из готовых элементов приложения в виде шаблонов экранных форм, отчетов, этикеток (Labels), графических конструкторов запросов, которые достаточно просто могли быть собраны в единый комплекс.
- Во всех настольных СУБД поддерживался только внешний уровень представления реляционной модели, то есть только внешний табличный вид структур данных.
- При наличии высокоуровневых языков манипулирования данными типа реляционной алгебры и SQL в настольных СУБД поддерживались низкоуровневые языки манипулирования данными на уровне отдельных строк таблиц.
- В настольных СУБД отсутствовали средства поддержки ссылочной и структурной целостности базы данных. Эти функции должны были выполнять приложения, однако скудость средств разработки приложений иногда не позволяла это сделать, и в этом случае эти функции должны были выполняться пользователем, требуя от него дополнительного контроля при вводе и изменении информации, хранящейся в БД.
- Наличие монопольного режима работы фактически привело к вырождению функций администрирования БД и в связи с этим — к отсутствию инструментальных средств администрирования БД.
- И, наконец, последняя и в настоящий момент весьма положительная особенность — это сравнительно скромные требования к аппаратному обеспечению со стороны настольных СУБД. Вполне работоспособные

приложения, разработанные, например, на Clipper, работали на PC 286.

- В принципе, их даже трудно назвать полноценными СУБД. Яркие представители этого семейства — очень широко использовавшиеся до недавнего времени СУБД Dbase (DbaseIII+, DbaselV), FoxPro, Clipper, Paradox.

1.2 Файловые системы

Историческим шагом явился переход к использованию централизованных систем управления файлами. С точки зрения прикладной программы файл - это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные. Правила именования файлов, способ доступа к данным, хранящимся в файле, и структура этих данных зависят от конкретной системы управления файлами и, возможно, от типа файла. Система управления файлами берет на себя распределение внешней памяти, отображение имен файлов в соответствующие адреса во внешней памяти и обеспечение доступа к данным.

Первая развитая файловая система была разработана фирмой IBM для ее серии 360. К настоящему времени она очень устарела, и мы не будем рассматривать ее подробно. Заметим лишь, что в этой системе поддерживались как чисто последовательные, так и индексно-последовательные файлы, а реализация во многом опиралась на возможности только появившихся к этому времени контроллеров управления дисковыми устройствами. Если учесть к тому же, что понятие файла в OS/360 было выбрано как основное абстрактное понятие, которому соответствовал любой внешний объект, включая внешние устройства, то работать с файлами на уровне пользователя было очень неудобно. Требовался целый ряд громоздких и перегруженных деталями конструкций. Все это хорошо знакомо программистам среднего и старшего поколения, которые прошли через использование отечественных аналогов компьютеров IBM.

Структуры файлов

Дальше мы будем говорить о более современных организациях файловых систем. Начнем со структур файлов. Прежде всего, практически во всех современных компьютерах основными устройствами внешней памяти являются магнитные

диски с подвижными головками, и именно они служат для хранения файлов. Такие магнитные диски представляют собой пакеты магнитных пластин (поверхностей), между которыми на одном рычаге двигается пакет магнитных головок. Шаг движения пакета головок является дискретным, и каждому положению пакета головок логически соответствует цилиндр магнитного диска. На каждой поверхности цилиндр "высекает" дорожку, так что каждая поверхность содержит число дорожек, равное числу цилиндров. При разметке магнитного диска (специальном действии, предшествующем использованию диска) каждая дорожка размечается на одно и то же количество блоков таким образом, что в каждый блок можно записать по максимуму одно и то же число байтов. Таким образом, для произведения обмена с магнитным диском на уровне аппаратуры нужно указать номер цилиндра, номер поверхности, номер блока на соответствующей дорожке и число байтов, которое нужно записать или прочитать от начала этого блока.

Именование файлов

Остановимся коротко на способах именования файлов. Все современные файловые системы поддерживают многоуровневое именование файлов за счет поддержания во внешней памяти дополнительных файлов со специальной структурой - каталогов. Каждый каталог содержит имена каталогов и/или файлов, содержащихся в данном каталоге. Таким образом, полное имя файла состоит из списка имен каталогов плюс имя файла в каталоге, непосредственно содержащем данный файл. Разница между способами именования файлов в разных файловых системах состоит в том, с чего начинается эта цепочка имен.

В этом отношении имеются два крайних варианта. Во многих системах управления файлами требуется, чтобы каждый архив файлов (полное дерево справочников) целиком располагался на одном дисковом пакете (или логическом диске, разделе физического дискового пакета, представляемом с помощью средств операционной системы как отдельный диск). В этом случае полное имя файла начинается с имени дискового устройства, на котором установлен соответствующий диск.

Защита файлов

Поскольку файловые системы являются общим хранилищем файлов, принадлежащих, вообще говоря, разным пользователям,

системы управления файлами должны обеспечивать авторизацию доступа к файлам. В общем виде подход состоит в том, что по отношению к каждому зарегистрированному пользователю данной вычислительной системы для каждого существующего файла указываются действия, которые разрешены или запрещены данному пользователю. Существовали попытки реализовать этот подход в полном объеме. Но это вызывало слишком большие накладные расходы как по хранению избыточной информации, так и по использованию этой информации для контроля правомочности доступа.

Режим многопользовательского доступа

Исторически в файловых системах применялся следующий подход. В операции открытия файла (первой и обязательной операции, с которой должен начинаться сеанс работы с файлом) помимо прочих параметров указывался режим работы (чтение или изменение). Если к моменту выполнения этой операции от имени некоторой программы А файл уже находился в открытом состоянии от имени некоторой другой программы В (правильнее говорить "процесса", но мы не будем вдаваться в терминологические тонкости), причем существующий режим открытия был несовместимым с желаемым режимом (совместимы только режимы чтения), то в зависимости от особенностей системы программе А либо сообщалось о невозможности открытия файла в желаемом режиме, либо она блокировалась до тех пор, пока программа В не выполнит операцию закрытия файла.

1.3 Понятие БД и СУБД

Система баз данных – это компьютеризированная система основная задача которой – хранение информации и предоставление доступа к ней по требованию.

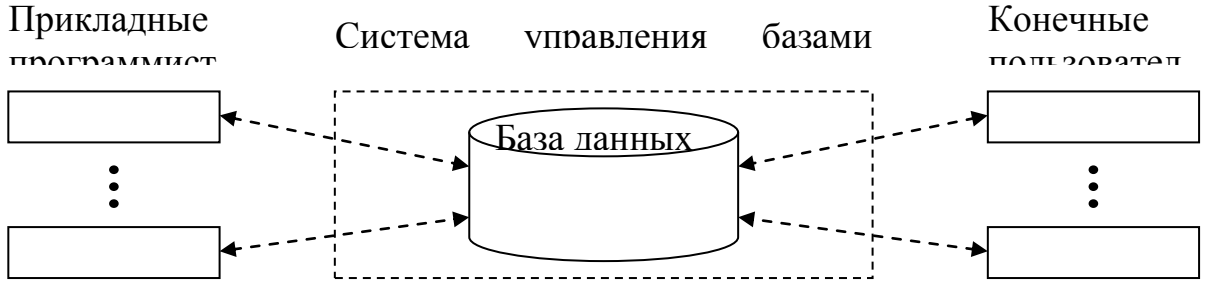


Рис. 1.1. Система баз данных.

Система баз данных включает в себя (рис 1.1):

1. данные, непосредственно сохраняемые в базе данных;
2. аппаратное обеспечение;
3. программное обеспечение;
4. пользователей:
5. прикладные программисты;
6. конечные пользователи;
7. администраторы баз данных.

Данные в базе данных являются интегрированными и, как правило, общими. Понятие интегрированных данных подразумевает возможность представления базы данных как объединение нескольких отдельных файлов данных, полностью или частично не перекрывающихся. Понятие общие подразумевает возможность использования отдельных областей данных, в базе данных несколькими различными пользователями.

К ***аппаратному обеспечению*** системы относятся накопители для хранения информации, вместе с устройствами ввода-вывода, контролерами устройств и т.д.; вычислительная техника, используемая для поддержки работы ПО системы.

Программное обеспечение является промежуточным слоем между собственно физической базой данных и пользователями системы и называется диспетчером базы данных или системой управления базами данных, СУБД (DBMS). Все запросы пользователей обрабатываются СУБД.

СУБД – это специализированное программное обеспечение, предоставляющее пользователю базы данных возможность работать с ней, не вникая в детали хранения информации на уровне программного обеспечения.

Прикладные программисты – отвечают за написание прикладных программ, использующих базу данных.

Конечные пользователи – работают с базой данных непосредственно, через рабочую станцию или терминал. Конечный пользователь может получить доступ к базе данных используя соответствующее прикладное ПО.

Администраторы базы данных – технические специалисты, осуществляющие создание БД, технический контроль работы СУБД и др. операции. Администраторы базы данных отвечают за реализацию решений администратора данных. Администратор

данных решает, какие данные необходимо хранить в БД, обеспечивает поддержание порядка при обслуживании и использовании хранимых в БД данных.

Функции администратора базы данных:

Определение концептуальной схемы. Администратор БД определяет какие именно данные необходимо сохранять в БД. Этот процесс обычно называют логическим (или концептуальным) проектированием БД. После определения содержимого БД на абстрактном уровне, администратор БД создает соответствующую концептуальную схему, с помощью концептуального ЯОД.

Определение внутренней схемы. Администратор БД решает, как данные должны быть представлены в хранимой БД. Этот процесс называют физическим проектированием. После завершения физического проектирования администратор БД с помощью внутреннего ЯОД должен создать соответствующую структуру хранения, а также определить отображение между внутренней и концептуальной схемой.

Взаимодействие с пользователями. Администратор БД обеспечивает пользователей необходимыми им данными. Для этого администратор БД должен написать (или оказать пользователям помощь в написании) необходимых внешних схем. Кроме этого необходимо определить отображение между внешней и концептуальной схемами.

1. Определение правил безопасности и целостности.
2. Определение процедур резервного копирования и восстановления.
3. Управление производительностью и реагирование на изменяющиеся требования.

База данных состоит из некоторого набора постоянных данных, которые используются прикладными системами какого-либо предприятия. Под словом "постоянные" подразумеваются данные, которые отличаются от других, более изменчивых данных, таких, как промежуточные данные и вообще все транзитные данные. "Постоянные" данные на самом деле могут недолго оставаться таковыми, поскольку данные в БД должны отражать об изменчивых объектах реального мира и отношениях между ними.

Использование баз данных для хранения информации позволяет организовать централизованное управление данными, что обеспечивает следующие преимущества:

1. возможность сокращения избыточности;
2. возможность устранения (до некоторой степени) противоречивости;
3. возможность общего доступа к данным;
4. возможность соблюдения стандартов;
5. возможность введения ограничений для обеспечения безопасности
6. возможность обеспечения целостности данных;
7. возможность сбалансировать противоречивые требования;
8. возможность обеспечения независимости данных. Поскольку программное обеспечение отделяется от данных, хранимых БД, изменения, вносимые в структуру БД, в большинстве случаев не приводят к необходимости внесения радикальных изменений в программное обеспечение.

Контрольные вопросы:

1. Как развивались базы данных?
2. Что такое файловая система?
3. Какие проблемы возникают при многопользовательском доступе?
4. Что такое СУБД?
5. Какие есть пользователи баз данных?
6. Какие преимущества использования баз данных?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p.
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p.

Глава II. Архитектура систем баз данных

Существует 3 уровня архитектуры СУБД:

1. Внутренний уровень - наиболее близкий к физическому хранению. Он связан со способами хранения информации на физических устройствах хранения;
2. Внешний уровень - наиболее близкий к пользователям. Он связан со способами представления данных для отдельных пользователей;
3. Концептуальный уровень - является промежуточным между двумя первыми. Этот уровень связан с обобщенными представлениями пользователей, в отличие от внешнего уровня, связанного с индивидуальными представлениями пользователей.

2.1 Внешний уровень – внешнее представление

Внешний уровень – индивидуальный уровень пользователя. Пользователь может быть, как прикладным программистом, так и конечным пользователем с любым уровнем профессиональной подготовки. Каждый пользователь имеет свой язык общения с СУБД. Для программиста - это какой-либо язык программирования, для пользователя - язык запросов или язык, основанный на формах и меню. Любой из этих языков включает подъязык данных, т.е. множество операторов всего языка, связанное только с объектами и операциями баз данных. Т.о. подъязык данных встроен в базовый язык пользователя, который также обеспечивает на связанные с БД возможности.

Представление отдельного пользователя о БД на внешнем уровне архитектуры называют внешним представлением. Т.о. внешнее представление – это содержимое БД, каким его видит отдельный пользователь. Например, сотрудник отдела кадров видит БД как набор записей о сотрудниках плюс набор записей о подразделениях. В общем случае внешнее представление состоит из множества экземпляров каждого типа внешней записи, которые не обязательно совпадают с хранимыми записями. Подъязык данных пользователя определен в терминах внешних записей. Каждое внешнее представление определяется средствами внешней

схемы, которая, в основном, состоит из определений каждого типа записей во внешнем представлении.

2.2 Концептуальный уровень – концептуальное представление

Концептуальное представление – это представление всей информации БД в несколько более абстрактной форме по сравнению с физическим способом хранения данных. Концептуальное представление представляет данные такими, какими они есть на самом деле, а не такими, какими их вынужден видеть пользователь в рамках определенного языка. Концептуальное представление состоит из множества экземпляров каждого типа концептуальной записи, хотя в некоторых системах в способы концептуального представления данных могут быть другими - например, в виде объектов и связей между ними. Концептуальное представление определяется средствами концептуальной схемы, которая состоит из определений каждого типа концептуальных записей. При определении концептуальной схемы используется концептуальный язык определения данных, определения которого относятся только к содержанию информации. Концептуальное представление, т.о. обеспечивает независимость данных от способа их хранения.

Внутренний уровень – внутреннее представление

Внутреннее представление – это представление нижнего уровня всей БД. Оно состоит из множества экземпляров каждого типа внутренней записи. Внутренняя запись соответствует хранимой записи. Внутреннее представление не связано с физическим уровнем и в нем не рассматриваются физические записи. Внутреннее представление предполагает существование бесконечного линейного адресного пространства. Подробности отображения этого пространства на физические устройства хранения не включены в общую архитектуру из-за сильной зависимости от системы.

Внутреннее представление описывается с помощью внутренней схемы, которая описывается с помощью внутреннего языка определения данных.

Между тремя уровнями представлений имеются два уровня отображений. Отображения концептуального уровня на внутренний и внешний уровень на концептуальный. Отображения

сохраняют независимость данных случае внесения в структуру БД изменений.

2.3 Функции СУБД

1. Определение данных. СУБД должна допускать определения данных (внешние схемы, концептуальную и внутреннюю схемы, соответствующие отображения). Для этого СУБД включает в себя языковый процессор для различных языков определений данных.
2. Обработка данных. СУБД должна обрабатывать запросы пользователя на выборку, а также модификацию данных. Для этого СУБД включает в себя компоненты процессора языка обработки данных.
3. Безопасность и целостность данных. СУБД должна контролировать запросы и пресекать попытки нарушения правил безопасности и целостности.
4. Восстановление данных и дублирование. СУБД должна обеспечить восстановление данных после сбоев.
5. Словарь данных. СУБД должна обеспечить функцию словаря данных. Сам словарь можно считать системной базой данных, которая содержит данные о данных пользовательской БД, т.е. содержит определения других объектов системы. Словарь интегрирован в определяемую им БД и, поэтому, содержит описание самого себя.
6. Производительность. СУБД должна выполнять свои функции с максимальной производительностью.

Контрольные вопросы:

1. Какие уровни архитектуры СУБД вы знаете?
2. Какие промежуточные уровни существуют между 3-мя основными?
3. Какие функции несет СУБД?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p.
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p.

III – глава. Модели данных. Моделирование сущность-связь

3.1 Общие сведения об инфологическом моделировании

В базе данных отображается какая-то часть реального мира. Естественно, что полнота ее описания будет зависеть от целей создаваемой информационной системы.

Для того чтобы база данных адекватно отражала предметную область, проектировщик базы данных должен хорошо представлять себе все нюансы, присущие данной предметной области (ПО), и уметь отобразить их в базе данных. Предметная область должна быть предварительно описана. Для этого в принципе может использоваться и естественный язык, но его применение имеет много недостатков, основными из них являются громоздкость описания и неоднозначность его трактовки. Поэтому обычно для этих целей используют искусственные формализованные языковые средства. В связи с этим под ***инфологической моделью*** (ИЛМ) понимают описание предметной области, выполненное с использованием специальных языковых средств, не зависящих от используемых в дальнейшем программных средств.

Инфологическая модель должна строиться вне зависимости от того, будете ли вы в дальнейшем использовать какую-либо СУБД или пользоваться другими программными средствами для реализации своей информационной системы.

Требования, предъявляемые к инфологической модели.

Основным требованием к ИЛМ, вытекающим из ее назначения, является требование адекватного отображения предметной области. ИЛМ должна быть непротиворечивой.

Несмотря на то, что реальный мир, отображаемый в ИЛМ, является по своей природе бесконечным, инфологическая модель является конечной, что обеспечивается четким ограничением предметной области. ИЛМ должна в связи с этим обладать свойством легкой расширяемости, обеспечивающим ввод новых данных без изменения ранее определенных. То же самое можно сказать и об удалении данных. В связи с большой размерностью реальных инфологических моделей должна обеспечиваться возможность композиции и декомпозиции модели.

Инфологическая модель должна легко восприниматься разными категориями пользователей. Желательно, чтобы ИЛМ

строил специалист, работающий в этой предметной области, а не проектировщик систем машинной обработки данных или хотя бы проверить сделанное описание, чтобы убедиться, что специфика предметной области воспринята правильно. Инфологическая модель должна также легко и однозначно восприниматься всеми специалистами, которые в дальнейшем участвуют в процессе проектирования баз данных и программного обеспечения.

Она является ядром системы проектирования. ИЛМ содержит необходимую и достаточную информацию для дальнейшего проектирования автоматизированной системы обработки информации.

3.2. Компоненты инфологической модели

Инфологическая модель предметной области включает в себя ряд компонентов (рис. 3.1). Центральной компонентой мифологической модели является описание объектов предметной области и связей между ними (ER-модель).

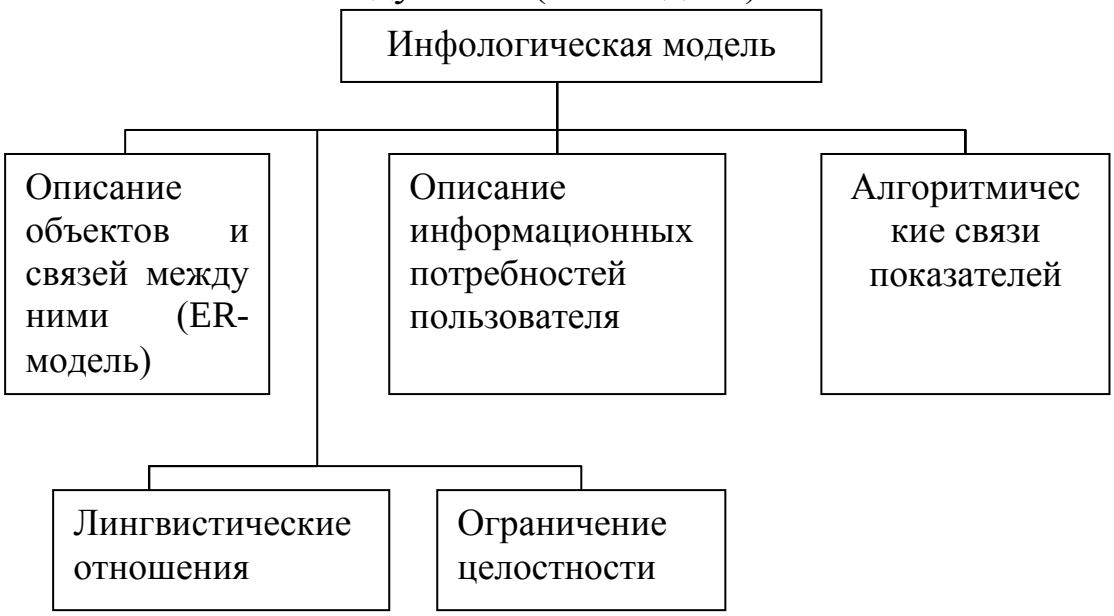


Рис 3.1. Компоненты инфологической модели

Построение модели “ОБЪЕКТ – СВОЙСТВО–ОТНОШЕНИЕ”.

Для описания ИЛМ используются как языки аналитического (описательного) типа, так и графические средства в дальнейшем применяется графический способ отображения модели “объект—свойство—отношение”. В предметной области в процессе ее обследования и анализа выделяют классы объектов. Классом

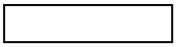
объектов называют совокупность объектов, обладающих одинаковым набором свойств. Например, если в качестве предметной области рассмотреть вуз, то в ней можно выделить следующие классы объектов: учащиеся, преподаватели, аудитории и т. д. Объекты могут быть реальными, как названные выше, а могут быть и абстрактными, как, например, предметы, которые изучают студенты.

При отражении в информационной системе каждый объект представляется своим идентификатором, который отличает один объект класса от другого, а каждый класс объектов представляется именем этого класса. Так, для объектов класса “ИЗУЧАЕМЫЕ ПРЕДМЕТЫ” идентификатором каждого объекта будет “НАЗВАНИЕ ПРЕДМЕТА”. Идентификатор должен быть уникальным.

Каждый объект обладает определенным набором свойств. Для объектов одного класса набор этих свойств одинаков, а их значения, естественно, могут различаться. Например, для объектов класса “СТУДЕНТ” таким набором свойств, описывающим объекты класса, может быть “ГОД РОЖДЕНИЯ”, “ПОЛ” и др.

При описании предметной области надо изобразить каждый из существующих классов объектов и набор свойств, фиксируемый для объектов данного класса.

Будем использовать для отображения объектов и их свойств следующие обозначения (рис. 3.2).

 - объект

 - свойство

Рис 3.2. Обозначение объектов и их свойств

Каждому классу объектов в инфологической модели присваивается уникальное имя.

При построении инфологической модели желательно дать словесную интерпретацию каждой сущности, особенно если возможно неоднозначное толкование понятия.

Рис 3.3. Изображение связи «объект - свойство»

При описании предметной области надо отразить связи между объектом и характеризующими его свойствами. Это изображается просто в виде линии, соединяющей обозначение объекта и его свойств.

Связь между объектом и его свойством может быть различной. Объект может обладать только одним значением какого-то свойства. Например, каждый человек может иметь только одну дату рождения. Назовем такие свойства единичными. Для других свойств возможно существование одновременно нескольких значений у одного объекта. Пусть, например, при описании “СОТРУДНИКА” фиксируется в качестве его свойства “ИНОСТРАННЫЙ ЯЗЫК”, которым он владеет. Так как сотрудник может знать несколько иностранных языков, то такое свойство будем называть множественным. При изображении связи между объектом и его свойствами для единичных свойств будем использовать одинарную стрелку, а для множественных свойств — двойную.

Кроме того, некоторые свойства являются постоянными, их значение не может измениться с течением времени. Назовем такие свойства статическими, а те свойства, значение которых может изменяться со временем, будем называть динамическими.

Другой характеристикой связи между объектом и его свойством является признак того, присутствует ли это свойство у всех объектов данного класса либо отсутствует у некоторых объектов. Например, для отдельных служащих может иметь место свойство “УЧЕНАЯ СТЕПЕНЬ”, а другие объекты этого класса могут не обладать, указанным свойством. Назовем такие свойства условными.

В инфологической модели отображаются не отдельные экземпляры объектов, а классы объектов. Когда в ИЛМ изображено обозначение объекта, то ясно, что речь идет о классе объектов, обладающих описанными свойствами. Поэтому в инфологическую модель в большинстве случаев можно в явном виде не вводить еще и обозначение для класса объектов. Явное изображение класса объектов необходимо только в том случае, если в ПО для данного класса объектов фиксируются не только характеристики, относящиеся к отдельным объектам этого класса, но и какие-то интегральные характеристики, относящиеся ко всему классу в целом. Например, если для класса объектов “СОТРУДНИКИ ПРЕДПРИЯТИЯ” фиксируется не только возраст каждого из сотрудников, но и средний возраст всех сотрудников, то в инфологической модели необходимо отразить не только объект “СОТРУДНИК”, но и класс объектов “СОТРУДНИКИ”. Для отображения класса объектов можно использовать какое-то специфическое обозначение или такое же, которое используется для объектов.

3.3. Семантическое моделирование данных, ER-диаграммы

Широкое распространение реляционных СУБД и их использование в самых разнообразных приложениях показывает, что реляционная модель данных достаточна для моделирования предметных областей. Однако проектирование реляционной базы данных в терминах отношений на основе кратко рассмотренного нами механизма нормализации часто представляет собой очень сложный и неудобный для проектировщика процесс.

При этом проявляется ограниченность реляционной модели данных в следующих аспектах:

- Модель не предоставляет достаточных средств для представления смысла данных. Семантика реальной предметной области должна независимым от модели способом представляться в голове проектировщика. В частности, это относится к упоминавшейся нами проблеме представления ограничений целостности.
- Для многих приложений трудно моделировать предметную область на основе плоских таблиц. В ряде случаев на самой начальной стадии проектирования проектировщику приходится производить насилие над собой, чтобы описать предметную область в виде одной (возможно, даже ненормализованной) таблицы.
- Хотя весь процесс проектирования происходит на основе учета зависимостей, реляционная модель не предоставляет каких-либо средств для представления этих зависимостей.
- Несмотря на то, что процесс проектирования начинается с выделения некоторых существенных для приложения объектов предметной области ("сущностей") и выявления связей между этими сущностями, реляционная модель данных не предлагает какого-либо аппарата для разделения сущностей и связей.

На использовании разновидностей ER-модели основано большинство современных подходов к проектированию баз данных (главным образом, реляционных). Модель была предложена Ченом (Chen) в 1976 г. Моделирование предметной области базируется на использовании графических диаграмм, включающих небольшое число разнородных компонентов. В связи с наглядностью представления концептуальных схем баз данных ER-модели получили широкое распространение в системах CASE, поддерживающих автоматизированное проектирование реляционных баз данных. Среди множества разновидностей ER-моделей одна из наиболее развитых применяется в системе CASE фирмы ORACLE. Ее мы и рассмотрим. Более точно, мы сосредоточимся на структурной части этой модели.

Основными понятиями ER-модели являются сущность, связь и атрибут.

Сущность – это реальный или представляемый объект, информация о котором должна сохраняться и быть доступна. В

диаграммах ER-модели сущность представляется в виде прямоугольника, содержащего имя сущности. При этом имя сущности - это имя типа, а не некоторого конкретного экземпляра этого типа. Для большей выразительности и лучшего понимания имя сущности может сопровождаться примерами конкретных объектов этого типа.

Ниже изображена сущность АЭРОПОРТ с примерными объектами Шереметьево и Хитроу:

Рис 3.4. Пример сущности.

Каждый экземпляр сущности должен быть отличим от любого другого экземпляра той же сущности (это требование в некотором роде аналогично требованию отсутствия кортежей-дубликатов в реляционных таблицах).

Связь – это графически изображаемая ассоциация, устанавливаемая между двумя сущностями. Эта ассоциация всегда является бинарной и может существовать между двумя разными сущностями или между сущностью и ей же самой (рекурсивная связь). В любой связи выделяются два конца (в соответствии с существующей парой связываемых сущностей), на каждом из которых указывается имя конца связи, степень конца связи (сколько экземпляров данной сущности связывается), обязательность связи (т.е. любой ли экземпляр данной сущности должен участвовать в данной связи).

Связь представляется в виде линии, связывающей две сущности или ведущей от сущности к ней же самой. При это в месте "стыковки" связи с сущностью используются трехточечный вход в прямоугольник сущности, если для этой сущности в связи могут использоваться много (many) экземпляров сущности, и одноточечный вход, если в связи может участвовать только один экземпляр сущности. Обязательный конец связи изображается сплошной линией, а необязательный - прерывистой линией.

Как и сущность, связь – это типовое понятие, все экземпляры обеих пар связываемых сущностей подчиняются правилам связывания.

В изображенном ниже примере связь между сущностями БИЛЕТ и ПАССАЖИР связывает билеты и пассажиров. При том конец сущности с именем "для" позволяет связывать с одним пассажиром более одного билета, причем каждый билет должен быть связан с каким-либо пассажиром. Конец сущности с именем "имеет" означает, что каждый билет может принадлежать только одному пассажиру, причем пассажир не обязан иметь хотя бы один билет.

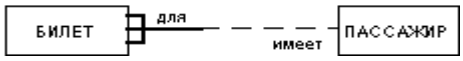


Рис 3.5. Пример связи

Лаконичной устной трактовкой изображенной диаграммы является следующая:

- Каждый БИЛЕТ предназначен для одного и только одного ПАССАЖИРА;
- Каждый ПАССАЖИР может иметь один или более БИЛЕТОВ.
- На следующем примере изображена рекурсивная связь, связывающая сущность ЧЕЛОВЕК с ней же самой. Конец связи с именем "сын" определяет тот факт, что у одного отца может быть более чем один сын. Конец связи с именем "отец" означает, что не у каждого человека могут быть сыновья.

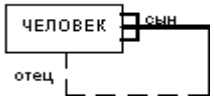


Рис 3.6. Рекурсивная связь

Лаконичной устной трактовкой изображенной диаграммы является следующая:

- Каждый ЧЕЛОВЕК является сыном одного и только одного ЧЕЛОВЕКА;
- Каждый ЧЕЛОВЕК может являться отцом для одного или более ЛЮДЕЙ ("ЧЕЛОВЕКОВ").

Атрибутом сущности является любая деталь, которая служит для уточнения, идентификации, классификации, числовой характеристики или выражения состояния сущности. Имена атрибутов заносятся в прямоугольник, изображающий сущность,

под именем сущности и изображаются малыми буквами, возможно, с примерами.

Три типа бинарных связей.

На рис. 3.7 показаны три типа бинарных связей. В связи 1:1 («один к одному») одиночный экземпляр сущности одного типа связан с одиночным экземпляром сущности другого типа. На рис. 3.7, а связь СЛУЖЕБНЫЙ_АВТОМОБИЛЬ связывает одиночную сущность класса СОТРУДНИК с одиночной сущностью класса АВТОМОБИЛЬ. В соответствии с этой диаграммой, ни за одним сотрудником не закреплено более одного автомобиля, и ни один автомобиль не закреплен более чем за одним сотрудником.

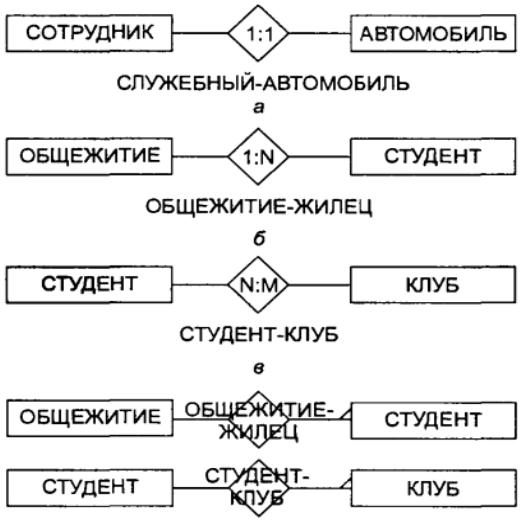


Рис 3.7. Три типа бинарных связей.

На рис. 3.7, б изображен второй тип связи, 1:N («один к N» или «один ко многим»), В этой связи, которая называется ОБЩЕЖИТИЕ-ЖИЛЕЦ, единичный экземпляр сущности класса ОБЩЕЖИТИЕ связан со многими экземплярами сущности класса СТУДЕНТ. В соответствии с этим рисунком, в общежитии проживает много студентов, но каждый студент живет только в одном общежитии.

Позиция, в которой стоят 1 и N, имеет значение. Единица стоит на той стороне связи, где располагается ОБЩЕЖИТИЕ, а N стоит на той стороне связи, где располагается СТУДЕНТ. Если бы 1 и N располагались наоборот, и связь записывалась бы как N:l, получилось бы, что в общежитии живет один студент, причем каждый студент живет в нескольких общежитиях. Это, разумеется, не так. На рис.7, в показан третий тип бинарной связи, N:M (читается «N к M» или «многие ко многим»). Эта связь называется

СТУДЕНТ-КЛУБ, и она связывает экземпляры сущностей класса СТУДЕНТ с экземплярами сущностей класса КЛУБ. Один студент может быть членом нескольких клубов, а в одном клубе может состоять много студентов.

Числа внутри ромба, символизирующего связь, обозначают максимальное количество сущностей на каждой стороне связи. Эти ограничения называются максимальными кардинальными числами, а совокупность из двух таких ограничений для обеих сторон связи называется максимальной кардинальностью (maximum cardinality) связи. Например, о связи, изображенной на рис. 3.3, б, говорят, что она обладает максимальной кардинальностью 1:N. Кардинальные числа могут иметь и другие значения, а не только 1 и N. Например, связь между сущностями БАСКЕТБОЛЬНАЯ КОМАНДА и ИГРОК может иметь кардинальность 1:5, что говорит нам о том, что в баскетбольной команде может быть не более пяти игроков.

Связи трех типов, представленных на рис. 7, называются иногда связями типа «ИМЕЕТ», или связями обладания (HAS-A relationships). Такой термин используется потому, что одна сущность имеет (has) связь с другой сущностью. Например: сотрудник имеет автомобиль, студент имеет общежитие, клуб имеет студентов.

Более сложные элементы ER-модели

Мы остановились только на самых основных и наиболее очевидных понятиях ER-модели данных. К числу более сложных элементов модели относятся следующие:

- Подтипы и супертипы сущностей. Как в языках программирования с развитыми типовыми системами (например, в языках объектно-ориентированного программирования), вводится возможность наследования типа сущности, исходя из одного или нескольких супертипов. Интересные нюансы связаны с необходимостью графического изображения этого механизма.
- Связи "many – to - many". Иногда бывает необходимо связывать сущности таким образом, что с обоих концов связи могут присутствовать несколько экземпляров сущности (например, все члены кооператива сообща

владеют имуществом кооператива). Для этого вводится разновидность связи "многие – со - многими".

- Уточняемые степени связи. Иногда бывает полезно определить возможное количество экземпляров сущности, участвующих в данной связи (например, служащему разрешается участвовать не более, чем в трех проектах одновременно). Для выражения этого семантического ограничения разрешается указывать на конце связи ее максимальную или обязательную степень.
- Каскадные удаления экземпляров сущностей. Некоторые связи бывают настолько сильными (конечно, в случае связи "один – ко - многим"), что при удалении опорного экземпляра сущности (соответствующего концу связи "один") нужно удалить и все экземпляры сущности, соответствующие концу связи "многие". Соответствующее требование "каскадного удаления" можно сформулировать при определении сущности.
- Домены. Как и в случае реляционной модели данных бывает полезна возможность определения потенциально допустимого множества значений атрибута сущности (домена).

Диаграммы сущность-связь

Схемы, изображенные на рис. 3.8, называются диаграммами «сущность—связь», или ER-диаграммами (entity-relationship diagrams, ER-diagrams). Такие диаграммы стандартизированы, но не слишком жестко. В соответствии с этим стандартом, классы сущностей обозначаются прямоугольниками, связи обозначаются ромбами, а максимальное кардинальное число каждой связи указывается внутри ромба1. Имя сущности указывается внутри прямоугольника, а имя связи указывается рядом с ромбом.

Рис 3.8. Пример диаграммы «сущность-связь»

Контрольные вопросы:

1. Что такое сущность?
2. Что такое связь?
3. Зачем применяется моделирование?
4. Сколько типов бинарных связей вы знаете?
5. Какие более сложные элементы схемы сущность-связь вы знаете?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p.
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p.

Глава IV. Реляционная модель

4.1. История реляционной модели

Теоретической основой этой модели стала теория отношений, основу которой заложили два логика — американец Чарльз Содерс Пирс (1839-1914) и немец Эрнст Шредер (1841-1902). В руководствах по теории отношений было показано, что множество отношений замкнуто относительно некоторых специальных

операций, то есть образует вместе с этими операциями абстрактную алгебру. Это важнейшее свойство отношений было использовано в реляционной модели для разработки языка манипулирования данными, связанного с исходной алгеброй.

Будучи математиком по образованию Э.Кодд предложил использовать для обработки данных аппарат теории множеств (объединение, пересечение, разность, декартово произведение). Он показал, что любое представление данных сводится к совокупности двумерных таблиц особого вида, известного в математике как отношение – relation.

Предложения Кодда были настолько эффективны для систем баз данных, что за эту модель он был удостоен престижной премии Тьюринга в области теоретических основ вычислительной техники.

В настоящее время основным предметом критики реляционных СУБД является не их недостаточная эффективность, а следующие недостатки:

1. присущая этим системам некоторая ограниченность (прямое следствие простоты) при использовании в так называемых нетрадиционных областях (наиболее распространенными примерами являются системы автоматизации проектирования), в которых требуются предельно сложные структуры данных.
2. невозможность адекватного отражения семантики предметной области. Другими словами, возможности представления знаний о семантической специфике предметной области в реляционных системах очень ограничены. Современные исследования в области постреляционных систем главным образом посвящены именно устранению этих недостатков.

Наименьшая единица данных реляционной модели – это отдельное атомарное (неразложимое) для данной модели значение данных. Так, в одной предметной области фамилия, имя и отчество могут рассматриваться как единое значение, а в другой – как три различных значения.

Доменом называется множество атомарных значений одного и того же типа. Наиболее правильной интуитивной трактовкой понятия домена является понимание домена как допустимого

потенциального множества значений данного типа. Например, домен "Имена" в нашем примере определен на базовом типе строк символов, но в число его значений могут входить только те строки, которые могут изображать имя (в частности, такие строки не могут начинаться с мягкого знака).

Следует отметить также семантическую нагрузку понятия домена: данные считаются сравнимыми только в том случае, когда они относятся к одному домену. В нашем примере значения доменов "Номера пропусков" и "Номера групп" относятся к типу целых чисел, но не являются сравнимыми. Заметим, что в большинстве реляционных СУБД понятие домена не используется, хотя в Oracle V.7 оно уже поддерживается.

Вхождение домена в отношение принято называть ***атрибутом***.

Схема отношения – это именованное множество пар {имя атрибута, имя домена (или типа, если понятие домена не поддерживается)}

Кортеж, соответствующий данной схеме отношения, – это множество пар {имя атрибута, значение}, которое содержит одно вхождение каждого имени атрибута, принадлежащего схеме отношения. "Значение" является допустимым значением домена данного атрибута (или типа данных, если понятие домена не поддерживается). Тем самым, степень или "арность" кортежа, т.е. число элементов в нем, совпадает с "арностью" соответствующей схемы отношения. Попросту говоря, кортеж - это набор именованных значений заданного типа.

Ключ (key) — это группа из одного или более атрибутов, которая уникальным образом идентифицирует строку. Рассмотрим отношение СЕКЦИЯ (рис. 5.3), имеющее атрибуты Номер Студента, Секция и Плата. Строка этого отношения содержит информацию о том, что студент посещает определенную секцию за определенную плату.

Отношение – это множество кортежей, соответствующих одной схеме отношения. Иногда, чтобы не путаться, говорят "отношение-схема" и "отношение-экземпляр", иногда схему отношения называют заголовком отношения, а отношение как набор кортежей - телом отношения. На самом деле, понятие схемы отношения ближе всего к понятию структурного типа данных в языках программирования. Было бы вполне логично разрешать

отдельно определять схему отношения, а затем одно или несколько отношений с данной схемой.

Обычным житейским представлением отношения является двумерная таблица. Каждая строка в таблице содержит данные, относящиеся к некоторой вещи или какой-то ее части. Каждый столбец таблицы описывает какой-либо атрибут этой вещи. Иногда строки называются кортежами (tuples), а столбцы — атрибутами (attributes).

Вышеупомянутые и некоторые другие математические понятия явились теоретической базой для создания реляционных СУБД, разработки соответствующих языковых средств и программных систем, обеспечивающих их высокую производительность, и создания основ теории проектирования баз данных. Однако для массового пользователя реляционных СУБД можно с успехом использовать неформальные эквиваленты этих понятий:

- Отношение – Таблица (иногда Файл);
- Кортеж – Строка (иногда Запись);
- Атрибут – Столбец, Поле.

Реляционная база данных – это совокупность отношений, содержащих всю информацию, которая должна храниться в БД. Однако пользователи могут воспринимать такую базу данных как совокупность таблиц.

Преимущества реляционной модели:

- Эта модель данных отображает информацию в наиболее простой для пользователя форме
- Основана на развитом математическом аппарате, который позволяет достаточно лаконично описать основные операции над данными.
- Позволяет создавать языки манипулирования данными не процедурного типа.
- Манипулирование данными на уровне выходной БД и возможность изменения.

Недостатки реляционной модели:

- Самый медленный доступ к данным.
- Трудоемкость разработки

Чтобы таблица была отношением, она должна удовлетворять определенным ограничениям:

1. Во-первых, значения в ячейках таблицы должны быть одиночными — ни повторяющиеся группы, ни массивы не допускаются
2. Строки имеют фиксированное число полей (столбцов) и значений (множественные поля и повторяющиеся группы недопустимы). Иначе говоря, в каждой позиции таблицы на пересечении строки и столбца всегда имеется в точности одно значение или ничего.
3. Строки таблицы обязательно отличаются друг от друга хотя бы единственным значением, что позволяет однозначно идентифицировать любую строку такой таблицы.
4. Все записи в столбце должны быть одного типа Например, если третий столбец первой строки таблицы содержит номер сотрудника, то и во всех остальных строках таблицы третий столбец также должен содержать номер сотрудника. Каждый столбец имеет уникальное имя; порядок столбцов в таблице несуществен. Наконец, в отношении не может быть двух одинаковых строк, и порядок строк не имеет значения.
5. Полное информационное содержание базы данных представляется в виде явных значений данных и такой метод представления является единственным.
6. При выполнении операций с таблицей ее строки и столбцы можно обрабатывать в любом порядке безотносительно к их информационному содержанию. Этому способствует наличие имен таблиц и их столбцов, а также возможность выделения любой их строки или любого набора строк с указанными признаками (например, рейсов с пунктом назначения "Париж" и временем прибытия до 12 часов).

Предложив реляционную модель данных, Э.Ф.Кодд создал и инструмент для удобной работы с отношениями – реляционную алгебру. Каждая операция этой алгебры использует одну или несколько таблиц (отношений) в качестве ее операндов и продуцирует в результате новую таблицу, т.е. позволяет "разрезать" или "склеивать" таблицы (рис. 4.1).

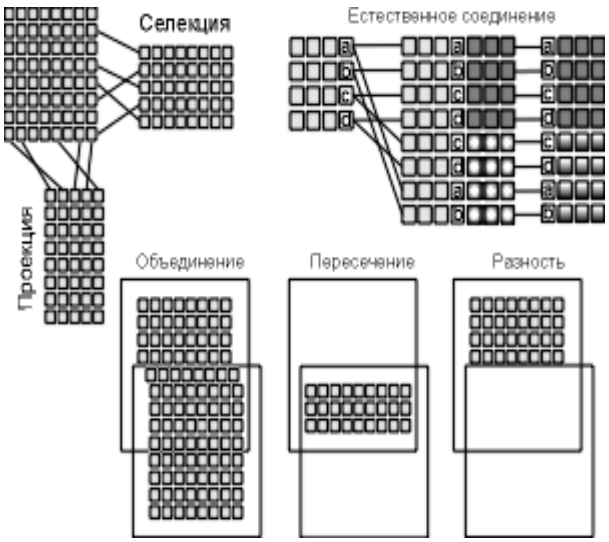


Рис. 4.1. Некоторые операции реляционной алгебры.

На рис. 4.2 представлен пример отношения. Отношение имеет семь строк, в каждой из которых четыре столбца. Если бы мы расположили столбцы в ином порядке (скажем, поместив Табельный Номер в крайний левый столбец) или переставили бы строки (например, по возрастанию значения столбца Возраст), мы получили бы эквивалентное отношение.

	Атрибут 1 **Имя**	Атрибут 2 **Возраст**	Атрибут 3 **Пол**	Атрибут 4 **ТабельныйНомер**
Кортеж 1	Андерсон	21	Ж	010110
Кортеж 2	Деккер	22	М	010100
.	Гловер	22	М	101000
.	Джексон	21	Ж	201100
.	Мур	19	М	111100
.	Наката	20	Ж	111101
Кортеж 7	Смит	19	М	111111

Рис. 4.2. Пример отношения «Сотрудник».

Свойства отношений.

Реляционное отношение обладает следующими свойствами.

- В отношении нет одинаковых кортежей.
- Кортежи отношения не упорядочены (сверху вниз).
- Атрибуты отношения не упорядочены (слева направо).
- Все значения атрибутов атомарны (скалярны, неделимы).

4.2. Отсутствие в отношении одинаковых кортежей

Это свойство следует из того факта, что тело отношения – это математическое множество (кортежей). В классической теории множеств по определению множество не содержит одинаковых элементов.

Это свойство является одним из примеров отмечаемой выше неэквивалентности понятий отношение и таблица. Таблица (в общем случае) может содержать одинаковые строки и, следовательно, таблица, содержащая одинаковые строки, не может быть отношением по определению.

Важным следствием того, что в отношении нет одинаковых строк-кортежей, является то, что в отношении всегда существует по крайней мере один потенциальный ключ. Действительно, так как кортежи уникальны, то обязательно комбинация всех или части атрибутов будет обладать свойством уникальности, и, следовательно, может служить ключом отношения, однозначно идентифицирующим кортежи.

Кортежи отношения не упорядочены (сверху вниз).

Это свойство также следует из того, что тело отношения – это математическое множество, а простые множества в математике не упорядочены. Так, в отношении, представленном на рис. 6.1, кортежи могли быть расположены в любом другом порядке, и, тем не менее, это все равно было бы то же самое отношение. Исходя из сказанного, к отношению не применимы такие понятия, как первый кортеж, последний кортеж, десятый кортеж, следующий или предыдущий кортеж и т. д. Другими словами, в отношении отсутствует позиционной адресация кортежей.

Обращение к конкретному кортежу, его идентификация могут быть осуществлены только по ключу отношения.

Это свойство также служит иллюстрацией не эквивалентности понятий отношение и таблица. В отличие от отношения, строки в таблице всегда упорядочены – есть первая строка, вторая, последняя и т. д.

Атрибуты отношения не упорядочены (слева направо).

Это свойство следует из того факта, что заголовок отношения также определен как простое математическое множество, а именно: множество пар <имя‑атрибута: имя-домена>.

Исходя из этого, если в представленной на рис. 4.3 таблице те же атрибуты были бы расставлены в каком-либо другом порядке, то это все равно было бы то же самое отношение. Поэтому не существует таких понятий, как первый атрибут, последний атрибут, следующий или предыдущий атрибут и т. д. Атрибут всегда адресуется или определятся по его имени, а не по расположению в отношении.

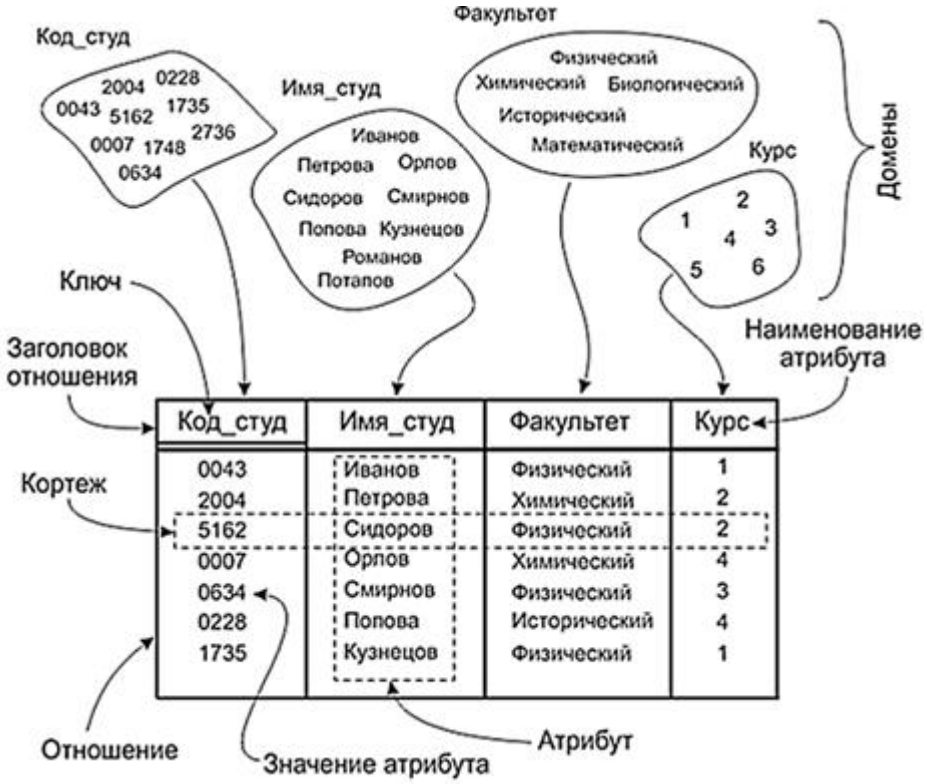


Рис. 4.3. Отношение и его компоненты

По этому пункту понятия отношение и таблица также не совпадают. Столбцы обычной таблицы, в отличие от атрибутов отношения, всегда упорядочены слева направо.

Значения всех атрибутов являются атомарными.

В реляционной модели домены, на которых определены атрибуты отношения, и из которых «черпаются» фактические значения атрибутов, могут содержать только атомарные (неделимые, скалярные) значения. Другими словами на пересечении столбца и строки таблицы, представляющей отношение, должно быть в точности одно значение, а не набор значений или какая либо сложная (составная) структура значений.

Отношение, удовлетворяющее этому условию, называется нормализованным, или представленным в первой нормальной

форме (другие нормальные формы будут обсуждаться позже, в следующих разделах).

Из сказанного следует, что реляционная модель рассматривает только нормализованные отношения, хотя отношение, понимаемое в математическом смысле, в общем случае может быть и ненормализованным. В качестве значений атрибутов ненормализованного отношения могут использоваться и более сложные структуры значений, например другие отношения.

Примером ненормализованного отношения может служить приведенная на рисунке 4.4 таблица Отношение 1. Как следует из вышесказанного, в реляционной модели отношения такого вида являются недопустимыми. Следует, однако, заметить, что это ограничение ни в коем смысле не ограничивает нас с точки зрения самой возможности отражения необходимой информации. Так вся информация, представленная в ненормализованном отношении Отношение 1, может быть полностью представлена в виде нормализованного отношения Отношение 2.

Отношение 1

Код_студ	Успеваемость	
	Дисциплина	Оценка
С9	Физика	5
	Математика	4
	История	4
	Химия	3
	Информатика	5
С6	Физика	3
	Математика	4
	Информатика	3
С1	Математика	5
С7	Информатика	5
	Химия	4

Отношение 2

Код_студ	Дисциплина	Оценка
С9	Физика	5
С9	Математика	4
С9	История	4
С9	Химия	3
С9	Информатика	5
С6	Физика	3
С6	Математика	4
С6	Информатика	3
С1	Математика	5
С7	Информатика	5
С7	Химия	4

Рис. 4.4. Пример ненормализованного и нормализованного отношений

С математической точки зрения требование использования только нормализованных отношений позволяет (за счет более простой их структуры) упростить операторы для работы с отношениями и уменьшить их количество, не ограничивая возможности адекватного отображения информационного содержания предметной области, так как для любого

ненормализованного отношения существует эквивалентная ему нормализованная форма.

Созданы языки манипулирования данными, позволяющие реализовать все операции реляционной алгебры и практически любые их сочетания. Среди них наиболее распространены SQL (Structured Query Language – структурированный язык запросов) и QBE (Quere - By-Example – запросы по образцу). Оба относятся к языкам очень высокого уровня, с помощью которых пользователь указывает, какие данные необходимо получить, не уточняя процедуру их получения.

С помощью единственного запроса на любом из этих языков можно соединить несколько таблиц во временную таблицу и вырезать из нее требуемые строки и столбцы (селекция и проекция).

4.3. Получение реляционной схемы из ER-схемы

Шаг 1. Каждая простая сущность превращается в таблицу. Простая сущность - сущность, не являющаяся подтипом и не имеющая подтипов. Имя сущности становится именем таблицы.

Шаг 2. Каждый атрибут становится возможным столбцом с тем же именем; может выбираться более точный формат. Столбцы, соответствующие необязательным атрибутам, могут содержать неопределенные значения; столбцы, соответствующие обязательным атрибутам, - не могут.

Шаг 3. Компоненты уникального идентификатора сущности превращаются в первичный ключ таблицы. Если имеется несколько возможных уникальных идентификатора, выбирается наиболее используемый. Если в состав уникального идентификатора входят связи, к числу столбцов первичного ключа добавляется копия уникального идентификатора сущности, находящейся на дальнем конце связи (этот процесс может продолжаться рекурсивно). Для именования этих столбцов используются имена концов связей и/или имена сущностей.

Шаг 4. Связи многие к одному (и один к одному) становятся внешними ключами. Т.е. делается копия уникального идентификатора с конца связи "один", и соответствующие столбцы составляют внешний ключ. Необязательные связи соответствуют столбцам, допускающим неопределенные значения;

обязательные связи - столбцам, не допускающим неопределенные значения.

Шаг 5. Индексы создаются для первичного ключа (уникальный индекс), внешних ключей и тех атрибутов, на которых предполагается в основном базировать запросы.

Шаг 6. Если в концептуальной схеме присутствовали подтипы, то возможны два способа:

- все подтипы в одной таблице (а)
- для каждого подтипа - отдельная таблица (б)

При применении способа (а) таблица создается для наиболее внешнего супертипа, а для подтипов могут создаваться представления. В таблицу добавляется по крайней мере один столбец, содержащий код ТИПА; он становится частью первичного ключа.

При использовании метода (б) для каждого подтипа первого уровня (для более нижних - представления) супертип воссоздается с помощью представления UNION (из всех таблиц подтипов выбираются общие столбцы - столбцы супертипа).

Все в одной таблице	**Таблица - на подтип**
Преимущества	
Все хранится вместе Легкий доступ к супертипу и подтипам Требуется меньше таблиц	Более ясны правила подтипов Программы работают только с нужными таблицами
Недостатки	

Слишком общее решение Требуется дополнительная логика работы с разными наборами столбцов и разными ограничениями Потенциальное узкое место (в связи с блокировками) Столбцы подтипов должны быть необязательными В некоторых СУБД для хранения неопределенных значений требуется дополнительная память	Слишком много таблиц Смущающие столбцы в представлении UNION Потенциальная потеря производительности при работе через UNION Над супертипом невозможны модификации

Шаг 7. Имеется два способа работы при наличии исключающих связей:

- общий домен (а)
- явные внешние ключи (б)

Если остающиеся внешние ключи все в одном домене, т.е. имеют общий формат (способ (а)), то создаются два столбца: идентификатор связи и идентификатор сущности. Столбец идентификатора связи используется для различения связей, покрываемых дугой исключения. Столбец идентификатора сущности используется для хранения значений уникального идентификатора сущности на дальнем конце соответствующей связи.

Если результирующие внешние ключи не относятся к одному домену, то для каждой связи, покрываемой дугой исключения, создаются явные столбцы внешних ключей; все эти столбцы могут содержать неопределенные значения.

Контрольные вопросы:

1. Что такое реляционная модель?
2. Что такое домен?
3. Что такое отношение?
4. Чем отношение отличается от таблицы?
5. Какими преимуществами и недостатками обладает реляционная модель?

6. Как производтся перевод модели сущность-связь в реляционную модель?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p.
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p.

Глава V. Реляционная алгебра и реляционное исчисление.

5.1. Обзор реляционной алгебры

Третья часть реляционной модели, манипуляционная часть, утверждает, что доступ к реляционным данным осуществляется при помощи реляционной алгебры или эквивалентного ему реляционного исчисления.

В реализациях конкретных реляционных СУБД сейчас не используется в чистом виде ни реляционная алгебра, ни реляционное исчисление. Фактическим стандартом доступа к реляционным данным стал язык SQL (Structured Query Language). Язык SQL представляет собой смесь операторов реляционной алгебры и выражений реляционного исчисления, использующий синтаксис, близкий к фразам английского языка и расширенный дополнительными возможностями, отсутствующими в реляционной алгебре и реляционном исчислении. Вообще, язык доступа к данным называется реляционно полным, если он по выразительной силе не уступает реляционной алгебре (или, что то же самое, реляционному исчислению), т.е. любой оператор реляционной алгебры может быть выражен средствами этого языка. Именно таким и является язык SQL.

Замкнутость реляционной алгебры.

Реляционная алгебра представляет собой набор операторов, использующих отношения в качестве аргументов, и возвращающие отношения в качестве результата. Таким образом, реляционный оператор f выглядит как функция с отношениями в качестве аргументов:

$$R = f(R_1, R_2, \ldots, R_n)$$

Реляционная алгебра является замкнутой, т.к. в качестве аргументов в реляционные операторы можно подставлять другие реляционные операторы, подходящие по типу:

$$R = f(f_1(R_{11}, R_{12}, \ldots), f_2(R_{21}, R_{22}, \ldots), \ldots)$$

Таким образом, в реляционных выражениях можно использовать вложенные выражения сколь угодно сложной структуры.

Каждое отношение обязано иметь уникальное имя в пределах базы данных. Имя отношения, полученного в результате выполнения реляционной операции, определяется в левой части равенства. Однако можно не требовать наличия имен от отношений, полученных в результате реляционных выражений, если эти отношения подставляются в качестве аргументов в другие реляционные выражения. Такие отношения будем называть неименованными отношениями. Неименованные отношения реально не существуют в базе данных, а только вычисляются в момент вычисления значения реляционного оператора.

Теоретико-множественные операции реляционной алгебры.

Реляционная алгебра в том виде, в котором она была определена Эдгаром Коддом, состоит из восьми операторов, составляющих две группы по четыре оператора.

1. Традиционные операции над множествами: объединение, пересечение, разность и декартово произведение (все они модифицированы с учетом того, что их операндами являются отношения, а не произвольные множества).
2. Специальные реляционные операции: выборка, проекция, соединение и деление.

Объединение (union) – возвращает отношение, содержащее все кортежи, которые принадлежат либо одному из двух заданных отношений, либо им обоим

Пересечение (intersect) – возвращает отношение, содержащее все кортежи, которые принадлежат одновременно двум заданным отношениям

Разность (minus) – возвращает отношение, содержащее все кортежи, которые принадлежат первому из двух заданных отношений и не принадлежат второму

Произведение (times) – возвращает отношение, содержащее все возможные кортежи, которые являются сочетанием двух кортежей, принадлежащих соответственно двум заданным отношениям

Выборка – возвращает отношение, содержащее все кортежи из заданного отношения, которые удовлетворяют указанным условиям. Операцию выборки также иногда называют операцией ограничения, поэтому далее в этой книге будет также употребляться термин ограничение, если подразумевается данная алгебраическая операция

Проекция – Возвращает отношение, содержащее все кортежи (подкортежи) заданного отношения, которые остались в этом отношении после исключения из него некоторых атрибутов

Соединение – *в*озвращает отношение, содержащее все возможные кортежи, которые представляют собой комбинацию атрибутов двух кортежей, принадлежащих двум заданным отношениям, при условии, что в этих двух комбинируемых кортежах присутствуют одинаковые значения в одном или нескольких общих для исходных отношений атрибутах (причем эти общие значения в результирующем кортеже появляются один раз, а не дважды)

Произведение – Возвращает отношение, содержащее все возможные кортежи, которые являются сочетанием двух кортежей, принадлежащих соответственно двум заданным отношениям

Свойства реляционных операций.

Пусть A, B и C – произвольные реляционные выражения (дающие совместимые по типу отношения). Тогда для операции объединения:

- **(A UNION B) UNION C ≡ A UNION (B UNION C)** – (свойство ассоциативности)
- **A UNION B ≡ B UNION A** – (свойство коммутативности).

Аналогично свойства ассоциативности и коммутативности определяются для остальных операций.

5.2. Реляционное исчисление

Реляционное исчисление – декларативный теоретический язык запросов, реализованный на основе исчисления предикатов первого порядка (высказываний в виде функции), которым должны удовлетворять искомые кортежи или домены отношений.

Запрос к БД, выполненный с использованием реляционного исчисления, содержит описание желаемого результата, для которого может существовать несколько способов его вычисления, представленных выражениями реляционной алгебры или непосредственно командами СУБД. Преимуществом реляционного исчисления перед реляционной алгеброй можно считать то, что пользователю не требуется самому строить алгоритм выполнения запроса, Программа СУБД (при достаточной ее интеллектуальности) сама строит эффективный алгоритм.

Существует два варианта исчислений: исчисление, кортежей и исчисление, доменов. В первом случае для описания отношений используются переменные, допустимыми значениями которых являются кортежи отношения, а во втором случае - элементы домена.

Реляционное исчисление основанное на кортежах (исчисление кортежей).

В исчислении кортежей, как и в процедурных языках программирования, сначала нужно описать используемые переменные, а затем записать выражения запроса к данным.

Описательную часть исчисления можно представить в виде:

```
RANGE OF <переменная> IS <список>.
```

Конструкция RANGE указывает идентификатор переменной кортежа (переменная) и область ее допустимых значений - список - последовательность одного или более элементов: x1, ..., xn, каждый из которых является либо отношением, либо выражением над отношением (порядок записи выражений описывается далее). При этом в любой момент переменная принимает в качестве значения только один из кортежей списка отношений.

Схемы отношений списка должны быть эквивалентными. Область допустимых значений переменной образуется путем объеди<нения значений всех элементов списка.

Использование переменных:

```
RANGE OF Var1 IS Table1, Table2
```

Область определения переменной Var1 включает в себя все значения из отношения, которое является объединением отношений Table1 и Table2.

Выражением реляционного исчисления кортежей называется конструкция вида <целевой_список > WHERE <WFF>

Значением выражения является отношение, тело (множество кортежей) которого должно удовлетворять WFF (well formulated formula - правильно построенная формула), а схема (набор атрибутов и их имена) определяется целевым списком. Целевой список по существу определяет операцию проекции, а формула WFF - селекцию кортежей.

В паре <переменная>.<атрибут> первая составляющая служит для указания переменной кортежа (определенной конструкцией RANGE), а вторая - для определения атрибута отношения, на котором изменяется переменная кортежа. Необязательная часть «AS <атрибут>» используется для переименования атрибута целевого отношения. Если она отсутствует, то имя атрибута целевого отношения наследуется от соответствующего имени атрибута исходного отношения.

Употребление в качестве элемента целевого отношения имени переменной равносильно перечислению в списке всех атрибутов соответствующего отношения.

Правильно построенная формула (*Well-Formed Formula, WFF*) служат для выражения условий, накладываемых на кортежные переменные. Основой WFF являются простые сравнения, представляющие собой операции сравнения скалярных значений (значений атрибутов переменных или констант).

Простые условия представляют собой операции сравнения скалярных значений. Приведем некоторые из них:

```
ИмяПеременной.ИмяАтрибута = СкалярноеЗначение
ИмяПеременнойА.ИмяАтрибутаБ =
ИмяПеременнойВ.ИмяАтрибутаГ
ИмяПеременной.ИмяАтрибута <> СкалярноеЗначение
ИмяПеременнойА.ИмяАтрибутаБ <
ИмяПеременнойВ.ИмяАтрибутаГ
```

Более сложные варианты WFF строятся с помощью логических связок NOT, AND, OR и IF ... THEN. Так, если

<формула> - WFF, а <сравнение> - простое сравнение, топеречисленные ниже выражения являются WFF:

```
NOT <формула>
<сравнение> AND <формула>
<сравнение> OR <формула>
IF <сравнение> THEN <формула>.
```

Допускается построение WFF с помощью кванторов. Если <формула> - это WFF, в которой участвует <переменная>, то перечисленные конструкции также являются WFF.

```
EXISTS <переменная> (<формула>)
FORALL <переменная> (<формула>.
```

В первом случае WFF означает: "Существует по крайней мере одно такое значение <переменной>, что вычисление <формулы> дает значение ИСТИНА".

Во втором случае WFF означает: "Для всех значений переменной <переменной> вычисление <формулы> дает значение ИСТИНА".

Переменные, входящие в WFF, могут быть свободными или связанными. Все переменные, входящие в WFF, при построении которой не использовались кванторы, являются свободными. Фактически, это означает, что если для какого-то набора значений свободных кортежных переменных при вычислении WFF получено значение ИСТИНА, то эти значения кортежных переменных могут входить в результирующее отношение.

Если же имя переменной использовано сразу после квантора при построении WFF вида `EXISTS <переменная> <формула>)` или `FORALL <переменная> (<формула>)`, то в этой WFF и во всех WFF, построенных с ее участием, <переменная> - это связанная переменная. Это означает, что такая переменная не видна за пределами минимальной WFF, связавшей эту переменную. При вычислении значения такой WFF используется не одно значение связанной переменной, а вся ее область определения.

Рассмотрим использование кванторов. Пусть Var1 и Var2 - две кортежные переменные, определенные на отношении Table1. Тогда, WFF:

`EXISTS Var2 (Var1.Table1_field1 > Var2.Table1_field1)` для текущего кортежа переменной Var1 принимает значение true в том и только в том случае, если во всем отношении Table1 найдется кортеж (связанный с переменной Var2) такой, что значение его атрибута Table1_field1 удовлетворяет внутреннему условию сравнения.

`FORALL Var2 (Var1.Table1_field1 > Var2.Table1_field1)` для текущего кортежа переменной Var1 принимает значение true в том и только в том случае, если для всех кортежей отношения Table1 (связанных с переменной Var2) значения атрибута Table1_field1 удовлетворяют условию сравнения.

На самом деле, правильнее говорить не о свободных и связанных переменных, а о свободных и связанных вхождениях переменных. Легко видеть, что если переменная var является связанной в WFF form, то во всех WFF, включающих данную, может использоваться имя переменной var, которая может быть свободной или связанной, но в любом случае не имеет никакого отношения к вхождению переменной var в WFF form.

Приведем пример использования двух связанных вхождений переменных:

```
EXISTS Var2 (Var1.Table1_field2 =
Var2.Table1_field2) AND FORALL Var2
(Var1.Table1_field1 > Var2.Table1_field1)
```

Здесь мы имеем два связанных вхождения переменной Var2 с совершенно разным смыслом.

Описанное исчисление не обладает вычислительной полнотой, так как не позволяет выполнять вычисления. Добавление вычислительных функций в исчисление можно реализовать путем расширения определения операндов сравнения и элементов целевого списка таким образом, чтобы они допускали использование скалярных выражений с литералами, ссылками на атрибуты и итоговыми функциями.

В качестве итоговых могут выступать следующие функции: COUNT (количество), SUMM (сумма), AVG (среднее), MAX (максимальное), MIN (минимальное).

Реляционное исчисление кортежей является основой для языка запросов QUEL.

5.3. Реляционное исчисление доменов

В исчислении доменов областью определения переменных являются не отношения, а домены. Применительно к базе данных Сотрудники можно говорить, например, о доменных переменных ФИО (значения - допустимые ФИО) или Номер отдела (значения - допустимые номера отделов предприятия).

Основным формальным отличием исчисления доменов от исчисления кортежей является наличие дополнительного набора предикатов, позволяющих выражать так называемые условия членства.

Во всем остальном формулы и выражения исчисления доменов выглядят похожими на формулы и выражения исчисления кортежей. В частности, конечно, различаются свободные и связанные вхождения доменных переменных.

Реляционное исчисление доменов является основой большинства языков запросов, основанных на использовании форм. В частности, на этом исчислении базировался известный язык Query by Example, который был первым (и наиболее интересным) языком в семействе языков, основанных на табличных формах.

Контрольные вопросы:

1. Что такое реляционная алгебра?
2. Какие виды операторов реляционной алгебры заложил Эдгар Кодд?
3. Какие свойства есть у реляционных операторов?
4. Что такое реляционное исчисление?
5. Какие два вида реляционных исчислений вы знаете?
6. Что такое правильно построенная формула?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p.
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p.

Глава VI. Планирование и разработка баз данных.

6.1. Жизненный цикл информационной системы

Любая информация имеет «время жизни». Она может существовать кратковременно (в памяти калькулятора в процессе проводимых на нем вычислений), в течение некоторого времени (при подготовке какой-либо справки) или очень долго (при хранении важных личных, коммерческих, общественных или государственных данных). Эти периоды времени определяют *жизненный цикл информации*.

Жизненный цикл ИС является производной жизненного цикла информации, информационных продуктов и услуг и технических средств.

Стадии жизненного цикла для информационных систем в различных отраслях человеческой деятельности, по сути, одинаковы:

1. постановка задачи;
2. проектирование услуг;
3. разработка и развертывание;
4. гарантированное предоставление услуг;
5. модернизация или ликвидация услуги.

Жизненный цикл создания и использования компьютерных программ отражает различные их состояния, начиная с момента возникновения необходимости в данном программном изделии и заканчивая моментом его полного выхода из употребления у всех пользователей.

Традиционно выделяются следующие основные этапы жизненного цикла программного обеспечения:

1. анализ требований;
2. проектирование;
3. кодирование (программирование);
4. тестирование и отладка;
5. эксплуатация и сопровождение.

Разработчики стремятся сделать максимально возможным период жизненного цикла информационных продуктов и услуг. Для большинства современных компьютерных программ длительность жизненного цикла равна двум–трём годам, хотя встречаются программы, существующие десять и более лет. Для

увеличения этого периода необходимо постоянно осуществлять маркетинговые и иные мероприятия по их поддержке.

Определённое время после снятия программного продукта с продажи может осуществляться его сопровождение. Отказ от продолжения выпуска и сопровождения программного продукта или от предоставления информационных услуг обычно обусловлен их неэффективностью, наличием неустранимых ошибок и отсутствием спроса.

Жизненный цикл ИС представляет собой модель ее создания и использования. Модель отражает различные состояния информационной системы, начиная с момента возникновения необходимости в данной системе и заканчивая моментом ее полного выхода из употребления у всех пользователей.

Под моделью жизненного цикла понимается структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач, выполняемых на протяжении всего ЖЦ.

Модель ЖЦ зависит от специфики информационной системы, а также специфики условий, в которых последняя создается и функционирует.

Наибольшее распространение получили три модели жизненного цикла информационных технологий: каскадная, поэтапная и спиральная.

Каскадная модель или «водопад» используется в технологиях, ориентированных на переход к следующему этапу после полного окончания работ на предыдущем этапе (рис. 6.1).

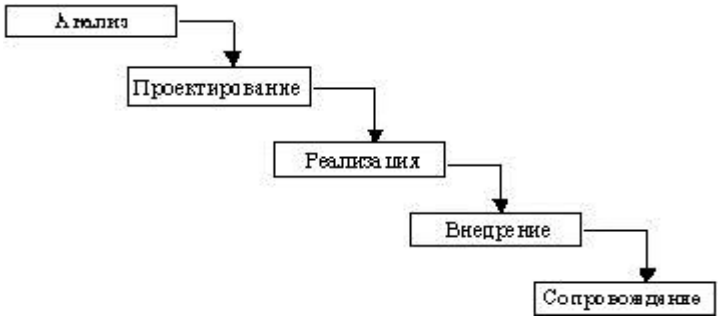


Рис. 6.1. Каскадная схема разработки ПО.

Недостатком такой модели является то, что реальный процесс создания ИС обычно полностью не укладывается в такую жесткую схему. Практически постоянно возникает потребность

возвращаться к предыдущим этапам, уточнять или пересматривать ранее принятые решения. В результате затягиваются сроки получения результатов, а пользователи могут вносить замечания лишь по завершению всех работ с системой. При этом модели автоматизируемого объекта могут устареть к моменту их утверждения.

Поэтапная модель обычно включает промежуточный контроль на любом этапе и межэтапные корректировки. Обеспечивает меньшую трудоемкость по сравнению с каскадной моделью, но время жизни каждого этапа становится равным всему жизненному циклу. Межэтапные корректировки позволяют уменьшить трудоемкость процесса разработки по сравнению с каскадной моделью (рис. 6.2).

Рис. 6.2. Поэтапная схема разработки ПО.

Спиральная модель (рис. 6.3) характеризуется тем, что на начальных этапах ЖЦ осуществляются выработка стратегии, анализ требований и предварительное детальное проектирование. При этом создаются прототипы (макеты), позволяющие проверить и обосновать реализуемость технических решений. Каждый виток спирали соответствует поэтапной модели создания фрагмента или версии изделия. На нём уточняются цели и характеристики проекта, определяется его качество, и планируются работы следующего витка спирали. В результате выбирается обоснованный вариант, который и реализуется.

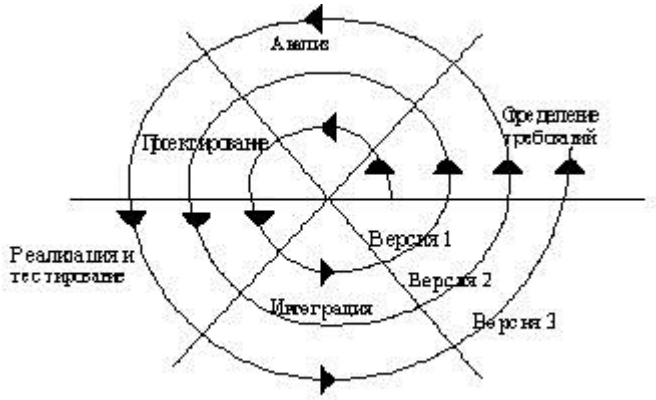


Рис. 6.3. Спиральная модель.

Жизненный цикл базы данных

Жизненный цикл базы данных (ЖЦБД) – это процесс проектирования, реализации и поддержки базы данных. ЖЦБД состоит из семи этапов:

1. предварительное планирование;
2. проверка осуществимости;
3. определение требований;
4. концептуальное проектирование;
5. логическое проектирование;
6. физическое проектирование;
7. оценка работы и поддержка базы данных.

В развитии любого экономического объекта наступает момент осознания того, что для достижения дальнейших успехов в развитии необходимо данные, находящиеся в личном пользовании работников, интегрировать для совместного использования в базе данных и воспринимать их как корпоративный ресурс.

1. *Предварительное планирование базы данных* – важный этап в процессе перехода от разрозненных данных к интегрированным. На этом этапе собирается информация об используемых и находящихся в процессе разработки прикладных программах и файлах, связанных с ними. Она помогает установить связи между текущими приложениями и то, как используется их информация. Кроме того, позволяет определить будущие требования к базе данных. Информация документируется в виде обобщенной концептуальной модели данных.

2. *Проверка осуществимости* предполагает подготовку отчетов по трем вопросам:

1. есть ли технология – необходимое оборудование и программное обеспечение – для реализации запланированной базы данных (технологическая осуществимость);
2. имеются ли персонал, средства и эксперты для успешного осуществления плана создания базы данных (операционная осуществимость);
3. окупится ли запланированная база данных (экономическая эффективность).

3. *Определение требований*. На этом этапе определяются:

- цели базы данных;
- информационные потребности различных структурных подразделений и их руководителей;
- требования к оборудованию;
- требования к программному обеспечению.

4. *Концептуальное проектирование.* На этом этапе создаются подробные модели пользовательских представлений данных предметной области. Затем они интегрируются в *концептуальную модель*, которая фиксирует все элементы корпоративных данных, подлежащих загрузке в базу данных. Эту модель еще называют *концептуальной схемой базы данных.*

5. *Логическое проектирование.* На этом этапе осуществляется выбор типа модели данных. Концептуальная модель отображается в *логическую модель*, основанную уже на структурах, характерных для выбранной модели.

6. *Физическое проектирование.* На этом этапе логическая модель расширяется характеристиками, необходимыми для определения способов физического хранения базы данных, типа устройств для хранения, методов доступа к данным базы, требуемого объема памяти, правил сопровождения базы данных и др.

7. *Оценка и поддержка базы данных.* Оценка включает опрос пользователей на предмет выяснения, какие их информационные потребности остались неучтенными. При необходимости в спроектированную базу данных вносятся изменения. Пользователи обучаются работе с базой данных. По мере расширения и изменения потребностей бизнеса поддержка базы данных обеспечивается путем внесения изменений, добавления

новых данных, разработки новых прикладных программ, работающих с базой данных.

6.2. Проектирование базы данных

Построение концептуальной модели самый сложный и трудно формализуемый процесс. Отсутствуют конструктивные методики, процесс структуризации по существу является искусством, опирающимся на опыт проектировщика и участие профессионала – работника моделируемой предметной области.

Концептуальное проектирование баз данных – это процесс создания модели, используемой информации, не зависящей от любых физических аспектов её представления – создание концептуального представления базы данных, включающее определение типов важнейших сущностей и существующих между ними связей и атрибутов. Далее этап логического проектирования.

Можно выделить два подхода к моделированию данных (проектированию концептуальной модели данных):

- в результате системного анализа предметной области (называют также семантическим моделированием) – наиболее распространенный, по крайней мере, в известных работах.
- в результате анализа информационных потребностей пользователей (проектирование БД на основе нормализации имеющихся).

Первый подход наиболее распространен (по крайней мере в известных работах), базируется на подходе, определенном Ченом примерно в 1976г. (Chen Peter). The Entity-Relationship Model 1976 ER-модель.

6.3. Реализация

CASE-средства – это автоматизированные средства, основанные на CASE-технологиях, позволяющие автоматизировать отдельные этапы жизненного цикла программного обеспечения. Все современные CASE-средства могут быть классифицированы по типам и категориям. Классификация по типам отражает функциональную ориентацию на процессы жизненного цикла программного обеспечения.

Классификация по категориям определяет степень интеграции по выполняемым функциям и включает отдельные локальные средства, решающие наиболее автономные задачи (по английские tools), набор частично интегрированных средств, охватывающих большинство этапов жизненного цикла (toolkit) и полностью интегрированные средства, поддерживающие весь жизненный цикл информационных систем.

Классификация по типам включает следующие основные CASE-средства:

1. Средства анализа, предназначенные для построения и анализа моделей предметной области (Bpwin, Design/IDEF);
2. Средства анализа и проектирования, предназначенные для создания проектных спецификаций (CASE.Аналитик, Vantage Team Builder, Designer/2000, Silverrun, PRO-IV);
3. Средства проектирования баз данных, обеспечивающие моделирование данных и генерацию схем баз данных для наиболее распространенных СУБД (Silverrun, Vantage Team Builder, Designer/2000, ERwin, S-Designor);
4. Средства разработки приложений и генераторы кодов (Vantage Team Builder, Silverrun, PRO-IV);
5. Средства реинжиниринга, обеспечивающие анализ программных кодов, схем баз данных и формирование на их основе различных моделей и проектных спецификаций. Средства анализа схем баз данных входят в состав: (Silverrun, Vantage Team Builder, Designer/2000, Erwin, S-Designor). Для анализа программных кодов используются такие средства, как Rational Rose и Object Team.

CASE-средство Silverrun американской фирмы Computer Systems Advisers (CSA) используется для анализа и проектирования информационных систем бизнес-класса и ориентировано, в большей степени, на спиральную модель жизненного цикла. Оно применимо для поддержки любой методологии, основанной на раздельном построении функциональной и информационной моделей (диаграмм потоков данных и диаграмм «сущность-связь»). Silverrun имеет модульную структуру и состоит из четырех модулей, каждый из которых является самостоятельным продуктом. Модуль построения моделей бизнес-процессов в форме диаграмм потоков данных

(BMP – Business Process Modeler) позволяет моделировать функционирование обследуемой организации или создаваемой информационной системы. Модуль концептуального моделирования данных (ERX – Entity-Relationship eXpert) обеспечивает построение моделей данных «сущность-связь», не привязанных к конкретной реализации. Модуль реляционного моделирования (RDM – Relational Data Modeler) позволяет создавать детализированные модели «сущность-связь», предназначенные для реализации в реляционной базе данных. Менеджер репозитория рабочей группы (WRM – Workgroup Repository Manager) применяется как словарь данных для хранения общей для всех моделей информации, а также обеспечивает интеграцию модулей Silverrun в единую среду проектирования. Платой за высокую гибкость и разнообразие изобразительных средств построения моделей является такой недостаток Silverrun, как отсутствие жесткого взаимного контроля между компонентами различных моделей (например, возможности автоматического распространения изменений между диаграммами потоков данных разного уровня). Но этот недостаток может иметь существенное значение только в случае использования каскадной модели жизненного цикла программного обеспечения. Для автоматической генерации схем баз данных у Silverrun существуют мосты к наиболее распространенным СУБД: Oracle, Informix, DB2, Ingres, Progress, SQL Server, SQLBase, Sybase. Для передачи данных в средства разработки приложений имеются мосты к языкам 4GL: JAM, PowerBuilder, SQL Windows, Uniface, NewEra, Delphi. Система Silverrun реализована на трех платформах – MS Windows, Macintosh, OS/2 Presentation Manager – с возможностью обмена проектными данными между ними.

Vantage Team Builder представляет собой интегрированный программный продукт, ориентированный на реализацию каскадной модели жизненного цикла программного обеспечения. Vantage Team Builder обеспечивает выполнение следующих функций: 1) проектирование диаграмм потоков данных, «сущность-связь», структур данных, структурных схем программ и последовательностей экранных форм; 2) генерацию кода программ на языке 4GL целевой СУБД с полным обеспечением программной среды и генерация SQL-кода для создания таблиц баз данных, индексов, ограничений целостности и хранимых

процедур; 3) программирование на языке С со встроенным SQL; 4) управление версиями и конфигурацией проекта; 5) генерация проектной документации по стандартным и индивидуальным шаблонам; 6) экспорт и импорт данных проекта. Vantage Team Builder поставляется в различных конфигурациях в зависимости от используемых СУБД (Oracle, Informix, Sybase, Ingress) или средств разработки приложений (Uniface). Конфигурация Vantage Team Builder обеспечивает совместное использование двух систем в рамках единой технологической среды проектирования, при этом схемы баз данных (SQL - модели) переносятся в репозиторий Uniface, и, наоборот, прикладные модели, сформированные средствами Uniface, могут быть перенесены в репозиторий Vantage Team Builder. Возможности рассогласования между репозиториями двух систем устаняются с помощью специальной утилиты. Разработка экранных форм в среде Uniface выполняется на базе диаграмм последовательностей форм (FSD) после импорта SQL – модели. Vantage Team Builder функционирует на всех основных Unix – платформах (Solaris, SCO UNIX, AIX, HP-UX) и VMS.

CASE-средство Designer/2000 фирмы Oracle является интегрированным CASE-средством, обеспечивающим в совокупности со средствами разработки приложений Developer/2000, поддержку полного жизненного цикла программного обеспечения для систем, использующих СУБД Oracle. В состав Designer/2000 входят следующие компоненты: 1) Repository Administrator – средства управления репозиторием (создание, удаление приложений, управление доступа к данным со стороны различных пользователей, экспорт и импорт данных); 2) Repository Object Navigator - средство доступа к репозиторию. Обеспечивающие многооконный объектно-ориентированный интерфейс доступа ко всем элементам репозитория; 3) Process Modeller – средство анализа и моделирования деловой деятельности, основывающиеся на концепциях реинжиниринга бизнес-процессов и глобальной системы управления качеством; 4) Systems Modeller – набор средств построения функциональных и информационных моделей проектируемой информационной системы, включающий средства для построения диаграмм «сущность-связь», диаграмм функциональных иерархий, диаграмм потоков данных и средство анализа и модификации связей

объектов репозитория различных типов; 5) Systems Designer – набор средств проектирования информационных систем, включающий средство построения структуры реляционной базы данных, а также средства построения диаграмм, отображающих взаимодействие с данными, иерархию, структуру и логику приложений, реализуемую хранимыми процедурами на языке SQL; 6) Server Generator – генератор описаний объектов базы данных Oracle (таблиц, индексов, ключей, последовательностей и т.д.). Помимо продуктов Oracle, генерация и реинжиниринг баз данных может выполняться для СУБД Informix, DB/2,Microsoft SQL Server, Sybase, а а также для баз данных, доступ к которым реализуется посредством ODBC; 7) Forms Generator – генератор приложения, включающий в себя различные экранные формы, средства контроля данных, проверки ограничений целостности и автоматические подсказки; 8) Repository Reports – генератор стандартных отчетов. Среда функционирования Designer/2000 – Windows 3.x, Windows 95, Windows NT.

Erwin – средство логического моделирования баз данных, использующее методологию IDEF1X. Erwin реализует проектирование схемы баз данных, генерацию её описания на языке целевой СУБД (Oracle, Informix, DB/2, Ingres, Progress, SQL Server, SQLBase, Sybase и др.) и реинжениринг существующей базы данных. Erwin выпускается в нескольких различных конфигурациях, ориентированных на наиболее распространенные средства разработки приложений 4GL. Версия Erwin/Open полностью совместима со средствами разработки приложений PowerBuilder и SQLWindows и позволяет экспортировать описание спроектированной базы данных непосредственно в репозитории данных средств.

S –Designor представляет собой CASE – средство для проектирования реляционных баз данных. S –Designor реализует стандартную методологию моделирования данных и генерирует описание баз данных для таких СУБД, как Oracle, Informix, DB/2, Ingres, Progress, SQL Server, SQLBase, Sybase и др. Для существующих систем выполняется реинжениринг баз данных.

Контрольные вопросы:

1. Что такое жизненный цикл информационной системы?

2. Какие основные модели разработки жизненного цикла ИС существуют?
3. В чем отличие жизненного цикла информационной системы от жизненного цикла базы данных?
4. Что такое концептуальное проектирование?
5. Какие категории CASE инструментов вы знаете?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p.
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p.

Глава VII. Нормализация баз данных

7.1. Проблемы при построении БД

Прежде чем мы приступим к обсуждению методов проектирования хороших схем баз данных, давайте посмотрим, почему некоторые схемы могут оказаться неадекватными. В частности, обратимся к схеме отношения ПОСТАВЩИКИ (НАЗВ_ПОСТ, АДРЕС_ПОСТ, ТОВАР, ЦЕНА)

НАЗВ_ПОСТ	АДРЕС_ПОСТ	ТОВАР	ЦЕНА
АГАТА-ИМПЕКС	ул Ата-тюрк 208	Компьютер P IV	2 000 000
НУРОН	ул Навои 158	Монитор LCD 17”	254 000
TS-TECHNOLOGY	ул Янгиюльская 64	Компьютер P IV	1 800 000
НУРОН	ул Навои 158	Клавиатура	25 000
SHARIFA-T	Чиланзар-6 64	Мышь	15 000

Рис. 7.1. Схема отношения ПОСТАВЩИКИ

В связи с этой схемой возникает несколько проблем:

1. *Избыточность*. Адрес поставщика повторяется для каждого поставляемого товара.
2. *Потенциальная противоречивость (аномалии обновления).* Вследствие избыточности мы можем обновлять адрес поставщика в одном кортеже, оставляя его неизменным в

другом. Таким образом, может оказаться, что для некоторых поставщиков нет единого адреса. Однако интуитивно мы чувствуем, что он должен быть.

3. *Аномалии включения.* В базу данных не может быть записан адрес поставщика, если он в настоящее время не поставляет по меньшей мере один товар. Можно, конечно, поместить неопределенные значения в компоненты ТОВАР и ЦЕНА кортежа для этого поставщика. Но если он начнет поставлять некоторый товар, не забудем ли мы удалить кортеж с неопределенными значениями? Хуже того, ТОВАР и НАЗВ_ПОСТ образуют ключ данного отношения, и поиск кортежей с неопределенными значениями в ключе может быть затруднительным или невозможным.
4. *Аномалии удаления.* Обратная проблема возникает при необходимости удаления всех товаров, поставляемых данным поставщиком, вследствие чего мы непреднамеренно утрачиваем его адрес.

Возникает вопрос: «Как найти хорошую замену для плохой схемы отношений?»

7.2. Нормальные формы

Отношения можно классифицировать по типам аномалий модификации, которым они подвержены. В 1970-х годах теоретики реляционных баз данных постепенно сокращали количество этих типов. Кто-то находил аномалию, классифицировал ее и думал, как предотвратить ее возникновение. Каждый раз, когда это происходило, критерии построения отношений совершенствовались. Эти классы отношений и способы предотвращения аномалий называются нормальными формами (normal forms). В зависимости от своей структуры, отношение может быть в первой, во второй или в какой-либо другой нормальной форме.

В своей работе, последовавшей за эпохальной статьей 1970 г., Кодд и другие определили первую, вторую и третью нормальные формы (1НФ, 2НФ и ЗНФ). Позднее была введена нормальная форма Бойса-Кодда (НФБК), а затем были определены четвертая и пятая нормальные формы. Как показывает рис. 7,2, эти нормальные формы являются вложенными. То есть отношение во второй нормальной форме является также отношением в первой

нормальной форме, а отношение в 5НФ (пятая нормальная форма) находится одновременно в 4НФ, НФБК, 3НФ, 2НФ и 1НФ.

Рис 7.2. Нормальные формы

Эти нормальные формы помогали, но у них было и серьезное ограничение. Не было теории, гарантирующей, что какая-либо из этих форм устранит все аномалии: каждая форма могла устранить только определенные их виды. Эта ситуация разрешилась в 1981 г., когда Р. Фагин (R. Fagin) ввел новую нормальную форму, которую он назвал домеино ключевой нормальной формой, или ДКНФ (domain/key normal form, DK/NF). В своей важной статье Фагин показал, что отношение в ДКНФ свободно от всех аномалий модификации, независимо от их типа1. Он также показал, что любое отношение, свободное от аномалий модификации, должно находиться в ДКНФ.

До введения ДКНФ теоретикам реляционных баз данных приходилось продолжать поиск все новых и новых аномалий и нормальных форм. Доказательство Фагина упростило ситуацию. Если мы можем привести отношение к ДКНФ, можем быть уверены, что в нем не будет аномалий модификации. Вся загвоздка в том, как привести отношение к ДКНФ.

Основные свойства нормальных форм:

- каждая следующая нормальная форма в некотором смысле лучше предыдущей;
- при переходе к следующей нормальной форме свойства предыдущих нормальных свойств сохраняются.

В основе процесса проектирования лежит метод нормализации, декомпозиция отношения, находящегося в предыдущей нормальной форме, в два или более отношения, удовлетворяющих требованиям следующей нормальной формы.

7.3. Нормализация и функциональные зависимости

Нормализация — это процесс преобразования отношения, имеющего некоторые недостатки, в отношение, которое этих недостатков не имеет. Что еще более важно, нормализацию можно использовать как критерий для определения желательности и правильности отношений. Вопрос о том, что такое хорошо структурированное отношение, был предметом многочисленных теоретических исследований. Термин нормализация обязан своим появлением одному из пионеров технологии баз данных, Э. Ф. Кодду (E. F. Codd), который определил различные нормальные формы (normal forms) отношений.

Наиболее важные на практике нормальные формы отношений основываются на фундаментальном в теории реляционных баз данных понятии *функциональной зависимости*. Для дальнейшего изложения нам потребуются несколько определений.

Функциональная зависимость (functional dependency) — это связь между атрибутами. Предположим, что если мы знаем значение одного атрибута, то можем вычислить (или найти) значение другого атрибута. Например, если нам известен номер счета клиента, то мы можем определить состояние его счета. В таком случае мы можем сказать, что атрибут Состояние Счета Клиента функционально зависит от атрибута Номер Счета Клиента.

Говоря более общим языком, атрибут Y функционально зависит от атрибута X, если значение X определяет значение Y. Другими словами, если нам известно значение X, мы можем определить значение Y.

Уравнения выражают функциональные зависимости. Например, если мы знаем цену и количество приобретенного товара, мы можем определить стоимость покупки по следующей формуле:

Стоимость = Цена x Количество.

В этом случае мы могли бы сказать, что атрибут Стоимость функционально зависит от атрибутов Цена и Количество.

Функциональные зависимости между атрибутами в отношении обычно не выражаются уравнениями. Пусть, например, каждому студенту присвоен уникальный

идентификационный номер, и у каждого студента есть одна и только одна специальность. Имея номер студента, мы можем узнать его специальность, поэтому атрибут Специальность функционально зависит от атрибута Номер Студента. Или рассмотрим компьютеры в вычислительной лаборатории. Каждый компьютер имеет конкретный размер основной памяти, поэтому атрибут Объем Памяти функционально зависит от атрибута Серийный Номер Компьютера.

В отличие от случая с уравнением, такие функциональные зависимости нельзя разрешить при помощи арифметики; вместо этого они хранятся в базе данных. Фактически, можно утверждать, что базу данных стоит иметь только ради хранения и выдачи функциональных зависимостей.

Функциональные зависимости обозначаются следующим образом:

Номер Студента > Специальность

Серийный Номер Компьютера > Объем Памяти

Первое выражение читается так: «атрибут Номер Студента функционально определяет атрибут Специальность», «атрибут Номер Студента определяет атрибут Специальность» или «атрибут Специальность зависит от атрибута Номер Студента». Атрибуты по правую сторону от стрелки называются ***детерминантами (determinants)***.

В функциональные зависимости могут быть вовлечены группы атрибутов. Рассмотрим отношение ОЦЕНКИ (Номер Студента, Дисциплина, Оценка). Комбинация номера студента и дисциплины определяет оценку. Такая функциональная зависимость записывается следующим образом:

(Номер Студента, Дисциплина) > Оценка

Заметьте, что для определения оценки требуется как номер студента, так и дисциплина. Мы не можем разделить эту функциональную зависимость, поскольку ни номер студента, ни дисциплина не определяют оценку сами по себе.

Ключ (key) — это группа из одного или более атрибутов, которая уникальным образом идентифицирует строку. Каждое отношение имеет минимум один ключ. Это утверждение должно быть верным, поскольку ни одно отношение не может иметь

одинаковых строк, и, следовательно, в крайнем случае, ключ будет состоять из всех атрибутов отношения.

Первая нормальная форма

О любой таблице данных, удовлетворяющей определению отношения, говорят, что она находится в первой нормальной форме (first normal form, 1NF). Вспомните, что для того, чтобы таблица была отношением, должно выполняться следующее: ячейки таблицы должны содержать одиночные значения и в качестве значений не допускаются ни повторяющиеся группы, ни массивы. Все записи в одном столбце (атрибуте) должны иметь один и тот же тип. Каждый столбец должен иметь уникальное имя, но порядок следования столбцов в таблице несуществен. Наконец, в таблице не может быть двух одинаковых строк, и порядок следования строк несуществен.

Вторая нормальная форма

Отношение находится во второй нормальной форме, если все его неключевые атрибуты зависят от всего ключа. В соответствии с этим определением, если отношение имеет в качестве ключа одиночный атрибут, то оно автоматически находится во второй нормальной форме.

Поскольку ключ является одиночным атрибутом, то по умолчанию каждый неключевой атрибут зависит от всего ключа, и частичных зависимостей быть не может. Таким образом, вторая нормальная форма представляет интерес только для тех отношений, которые имеют композитные ключи.

Третья нормальная форма

Отношения во второй нормальной форме также могут иметь аномалии. Рассмотрим отношение ПРОЖИВАНИЕ на рис. 7.3. Ключом здесь является Номер Студента, и имеются функциональные зависимости Номер Студента > Общежитие и Общежитие > Плата. Эти зависимости возникают потому, что каждый студент живет только в одном общежитии, и каждое общежитие взимает со всех проживающих в нем студентов одинаковую плату. Например, каждый живущий в общежитии Рэндольф - Холл платит $3200 за квартал.

Функциональные зависимости:Общежитие → Плата
НомерСтудента → Общежитие → Плата

НомерСтудента	Общежитие	Плата
100	Рэндольф	3200
150	Ингерсол	3100
200	Рэндольф	3200
250	Питкин	3100
300	Рэндольф	3200

Рис. 7.3. Схема Общежитие.

Поскольку Номер Студента определяет атрибут Общежитие, а Общежитие определяет атрибут Плата, то косвенным образом Номер Студента > Плата. Такая структура функциональных зависимостей называется транзитивной зависимостью (transitive dependence), поскольку атрибут Номер Студента определяет атрибут Плата через атрибут Общежитие.

Ключом отношения ПРОЖИВАНИЕ является Номер Студента, который является одиночным атрибутом, и, следовательно, отношение находится во второй нормальной форме (и Общежитие, и Плата определяются атрибутом Номер Студента). Несмотря на это, отношение ПРОЖИВАНИЕ имеет аномалии, обусловленные транзитивной зависимостью.

Что произойдет, если мы удалим вторую строку отношения на рис. 2? Мы потеряем не только тот факт, что студент № 150 живет в Ингерсолл -Холле, но и тот факт, что проживание в этом общежитии стоит $3100. Это аномалия удаления. А как мы можем записать тот факт, что плата за проживание в Кэррigг - Холле составляет $3500? Никак, пока туда не решит вселиться хотя бы один студент. Это аномалия вставки.

НомерСтудента	Общежитие
100	Рэндольф
150	Ингерсол
200	Рэндольф
250	Питкин
300	Рэндольф

Общежитие	Плата
Рэндольф	3200
Ингерсол	3100
Рэндольф	3200
Питкин	3100
Рэндольф	3200

Рис 7.4. Нормализованная схема Общежитие.

Чтобы удалить аномалии из отношения во второй нормальной форме, необходимо устранить транзитивную зависимость рис. 7.4. Отношение находится в третьей нормальной форме, если оно находится во второй нормальной форме и не имеет транзитивных зависимостей.

Нормальная форма Бойса - Кодда

Функциональные
зависимости: Преподаватель → Специальность

НомерСтудента	Специальность	Преподаватель
100	Математика	Коши
150	Психология	Юнг
200	Математика	Риман
250	Математика	Коши
300	Рэндольф	Перлс
300	Психология	Риман

Рис. 7.5 Схема отношения Преподаватель

Поскольку студенты могут специализироваться в нескольких областях, атрибут Номер Студента не определяет атрибут Специальность. Более того, так как студент может иметь несколько консультантов, Номер Студента не определяет и атрибут Преподаватель. Таким образом, Номер Студента сам по себе не может быть ключом.

Комбинация (Номер Студента, Специальность) определяет атрибут Преподаватель, а комбинация (Номер Студента,

Преподаватель) определяет атрибут Специальность. Следовательно, любая из этих комбинаций может быть ключом. Два или более атрибута или группы атрибутов, которые могут быть ключом, называются ключами-кандидатами (candidate keys). Тот из ключей-кандидатов, который выбирается в качестве ключа, называется первичным ключом (primary key).

Кроме ключей-кандидатов, есть еще одна функциональная зависимость, которую следует рассмотреть: атрибут Преподаватель определяет атрибут Специальность (любой из преподавателей является консультантом только по одному предмету; следовательно, зная имя преподавателя, мы можем определить специальность). Таким образом, Преподаватель является детерминантом.

По определению, отношение КОНСУЛЬТАНТ находится в первой нормальной форме. Оно также находится во второй нормальной форме, поскольку не имеет неключевых атрибутов (каждый из атрибутов является частью минимум одного ключа). Наконец, это отношение находится в третьей нормальной форме, так как не имеет транзитивных зависимостей. Тем не менее, несмотря на все это, отношение имеет аномалии модификации.

Пусть студент с номером 300 отчисляется из университета. Если мы удалим строку с информацией о студенте с номером 300, мы потеряем тот факт, что Перле является консультантом по психологии. Это аномалия удаления. Далее, как мы можем записать в базу тот факт, что Кейнс является консультантом по экономике? Никак, пока не появится хотя бы один студент, специализирующийся на экономике. Это аномалия удаления.

Ситуации, подобные только что описанной, приводят нас к определению нормальной формы Бойса - Кодда (Boyce-Codd normal form, BK/NF): отношение находится в НФБК, если каждый детерминант является ключом-кандидатом. Отношение КОНСУЛЬТАНТ не находится в НФБК, поскольку детерминант Преподаватель не является ключом-кандидатом.

НомерСтудента	Преподаватель
100	Коши
150	Юнг
200	Риман
250	Коши
300	Перлс
300	Риман

Консультант	Предмет
Коши	Математика
Юнг	Психология
Риман	Математика
Перлс	Психология

Рис. 7.6. Нормализованная схема Преподаватель.

Как и в других примерах, отношение КОНСУЛЬТАНТ можно разбить на два отношения, не имеющие аномалий. Например, отношения СТУДЕНТ-КОНСУЛЬТАНТ (Номер Студента, Преподаватель) и КОНСУЛЬТАНТ-ПРЕДМЕТ (Преподаватель, Специальность) не имеют аномалий рис. 7.6.

Четвертая нормальная форма

Многозначные зависимости: НомерСтудента →→ Специальность
НомерСтудента →→ Секция

НомерСтудента	Специальность	Секция
100	Музыка	Плавание
100	Бухгалтерский учет	Плавание
100	Музыка	Теннис
100	Бухгалтерский учет	Теннис
150	Математика	Оздоровительный бег

Рис. 7.7. Схема отношения Специальность.

Рассмотрим отношение СТУДЕНТ на рис. 7.7, которое отображает связи между студентами, специальностями и секциями. Предположим, что студенты могут иметь несколько специальностей и заниматься в нескольких различных секциях. В таком случае единственным ключом является комбинация (НомерСтудента, Специальность, Секция). Например, студентка с номером 100 специализируется на музыке и бухгалтерском учете и, кроме того, посещает секции плавания и тенниса, а студент с номером 150 специализируется только на математике и занимается бегом.

Какова связь между атрибутами Номер Студента и Специальность? Это не функциональная зависимость, поскольку у студента может быть несколько специальностей. Одному и тому же значению атрибута Номер Студента может соответствовать много значений атрибута Специальность. Помимо того, одному и тому же значению атрибута Номер Студента может соответствовать много значений атрибута Секция.

Такая зависимость атрибутов называется многозначной зависимостью (multivalue dependency). Многозначные зависимости приводят к аномалиям модификации. Для начала обратите внимание на избыточность данных на рис. 6. Студентке с номером 100 посвящено четыре записи, в каждой из которых указана одна из ее специализаций и одна из посещаемых ею секций. Если бы те же данные хранились в меньшем количестве строк (скажем, было бы две строки — одна для музыки и плавания, а другая для бухгалтерского учета и тенниса), это дезориентировало бы пользователей. Получалось бы, что студентка с номером 100 плавает только тогда, когда специализируется на музыке, а в теннис играет только тогда, когда специализируется на бухгалтерском учете. Но такая интерпретация нелогична. Специальности и секции совершенно независимы друг от друга. Поэтому, чтобы избежать таких неверных заключений, мы храним все сочетания специальностей и секций.

НомерСтудента	Специальность	Секция
100	Музыка	Лыжи
100	Музыка	Плавание
100	Бухгалтерский учет	Плавание
100	Музыка	Теннис
100	Бухгалтерский учет	Теннис
150	Математика	Оздоровительный бег

а

НомерСтудента	Специальность	Секция
100	Музыка	Лыжи
100	Бухгалтерский учет	Лыжи
100	Музыка	Плавание
100	Бухгалтерский учет	Плавание
100	Музыка	Теннис
100	Бухгалтерский учет	Теннис
150	Математика	Оздоровительный бег

б

Рис. 7.8. Нормализованная схема отношения Специальность.

Допустим, что студентка с номером 100 решила записаться в секцию лыж, и поэтому мы добавляем в таблицу строку [100, Музыка, Лыжи], как показано на рис. 7.8 а данный момент из

отношения можно сделать вывод, что студентка 100 занимается лыжами только как музыкант, но не как бухгалтер. Чтобы данные имели согласованный характер, мы должны добавить столько строк, сколько имеется специальностей, и в каждой из них указать секцию лыж. Таким образом, мы должны добавить строку [100, Бухгалтерский учет, Лыжи], как показано на рис. 7.8 б. Это аномалия обновления: требуется слишком много модификаций, чтобы внести одно простое изменение.

Вообще говоря, многозначная зависимость существует, когда отношение имеет минимум три атрибута, причем два из них являются многозначными, а их значения зависят только от третьего атрибута. Другими словами, в отношении R (A, B, C) существует многозначная зависимость, если A многозначным образом определяет B и C, а сами B и C не зависят друг от друга. Как мы видели из предыдущего примера, Номер Студента многозначно определяет атрибуты Специальность и Секция, но сами Специальность и Секция не зависят друг от друга.

Чтобы устранить эти аномалии, мы должны избавиться от многозначной зависимости. Мы сделаем это, создав два отношения, в каждом из которых будут храниться данные только по одному многозначному атрибуту. Результирующие отношения не будут иметь аномалий. Это отношения СТУДЕНТ-СПЕЦИАЛЬНОСТЬ (Номер Студента, Специальность) и СТУДЕНТ-СЕКЦИЯ (Номер Студента, Секция), приведенные на рис 7.9.

НомерСтудента	Специальность
100	Музыка
100	Бухгалтерский учет
150	Математика

НомерСтудента	Секция
100	Лыжи
100	Плавание
100	Теннис
150	Оздоровительный бег

Рис 7.9. Устранение многозначной зависимости.

Отношение находится в четвертой нормальной форме, если оно находится в НФБК и не имеет многозначных зависимостей.

Пятая нормальная форма

Пятая нормальная форма (fifth normal form, 5NF) связана с зависимостями, которые имеют несколько неопределенный характер. Речь здесь идет об отношениях, которые можно разделить на несколько более мелких отношений, как мы это делали выше, но затем невозможно восстановить.

Условия, при которых возникает эта ситуация, не имеют ясной, интуитивной интерпретации. Нам неизвестно, каковы следствия таких зависимостей; мы не знаем даже, есть ли у них какие-либо практические следствия.

Доменно-ключевая нормальная форма

В 1981 г. Фагин опубликовал важную статью, в которой он определил доменно-ключевую нормальную форму (domain/key normal form, DKNFI. Он показал, что отношение в ДКНФ не имеет аномалий модификации и, более того, любое отношение, не имеющее аномалий модификации, должно находиться в ДКНФ.

Это открытие положило конец введению нормальных форм, и теперь в нормальных формах более высокого порядка нет необходимости — по крайней мере, для устранения аномалий модификации.

Контрольные вопросы:

1. Что такое аномалия?
2. Какие виды аномалий могут встречаться в базах данных?
3. Какие виды енормальных форм существуют?
4. Какими своствами обладают все нормальные формы?
5. Что такое нормализация?
6. Что такое многозначные зависимости?
7. Что такое доменно-ключевая нормальная форма?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p.
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p.

Глава VIII. Введение в SQL

8.1. История SQL

Все языки манипулирования данными (ЯМД), созданные до появления реляционных баз данных и разработанные для многих систем управления базами данных (СУБД) персональных компьютеров, были ориентированы на операции с данными, представленными в виде логических записей файлов. Это требовало от пользователей детального знания организации хранения данных и достаточных усилий для указания не только того, какие данные нужны, но и того, где они размещены и как шаг за шагом получить их.

Разработка, в основном, шла в отделениях фирмы IBM (языки ISBL, SQL, QBE) и университетах США (PIQUE, QUEL). Последний создавался для СУБД INGRES (Interactive Graphics and Retrieval System), которая была разработана в начале 70-х годов в Университете шт. Калифорния и сегодня входит в пятерку лучших профессиональных СУБД. Сегодня из всех этих языков полностью сохранились и развиваются QBE (Query-By-Example - запрос по образцу) и SQL, а из остальных взяты в расширение внутренних языков СУБД только наиболее интересные конструкции.

В начале 70-х годов плодотворный труд исследователя из IBM доктора Кодда (E. F. Codd) привел к созданию продукта, связанного с реляционной моделью данных под названием SEQUEL (Structured English Query Language, структурированный английский язык для запросов), который в 1980 г. был переименован в SQL (Structured Query Language, структурированный язык запросов).

С тех пор, помимо IBM, многие производители присоединились к разработке программных продуктов для SQL. Компании IBM, а также другим производителям реляционных баз данных был нужен стандартизованный метод доступа к реляционной базе и манипулирования хранящимися в ней данными. Хотя компания IBM первая разработала теорию реляционных баз данных, первой на рынок с этой технологией вышла компания Oracle. Через какое-то время SQL завоевал на рынке достаточную популярность и привлек внимание Американского национального института по стандартизации (American National Standards Institute, ANSI), который в 1986, 1989,

1992, 1999 и 2003 годах выпустил стандарты языка SQL. Последняя версия стандарта, SQL3, содержит расширения языка для объектно-ориентированного программирования.

Начиная с 1986 года несколько конкурирующих между собой языков позволяли программистам и разработчикам обращаться к реляционным данным и манипулировать ими. Однако очень немногие из них были настолько же просты в изучении и повсеместно приняты, как SQL. Программистам и администраторам теперь можно изучить один язык, который с небольшими изменениями можно применять к разнообразным платформам баз данных, приложениям и прочим продуктам.

SQL, или язык структурированных запросов — на сегодняшний день наиболее важный из языков манипулирования реляционными данными. Он рекомендован Американским национальным институтом стандартов (ANSI) в качестве стандартного языка манипулирования реляционными базами данных и используется как язык доступа к данным многими коммерческими СУБД, включая DB2, SQL/DS, Oracle, INGRES, SYBASE, SQL Server, dBase for Windows, Paradox, Microsoft Access и многие другие. Благодаря своей популярности SQL стал стандартным языком для обмена информацией между компьютерами. Поскольку существует версия SQL, которая может работать почти на любом компьютере и операционной системе, компьютерные системы способны обмениваться данными, передавая друг другу запросы и ответы на языке SQL.

Постоянное развитие стандарта SQL способствовало появлению среди разных производителей и платформ многочисленных диалектов SQL. Эти диалекты развивались главным образом потому, что сообществу пользователей конкретной базы данных требовались какие-то возможности, до того как комитет ANSI разрабатывал стандарт. Однако иногда новую функциональность вводит научно-исследовательское сообщество по причине давления со стороны конкурирующих технологий. Например, многие производители баз данных к существующим возможностям программирования своих продуктов добавляют Java (как это делается в DB2, Oracle и Sybase) или VBScript (как это делает Microsoft). В будущем программисты и разработчики будут использовать эти языки программирования в SQL-программах наряду с самим SQL.

Многие из этих диалектов включают средства условной обработки (например, под контролем оператора IF...THEN), управляющие операторы (например, циклы WHILE), переменные и обработку ошибок. Поскольку ANSI пока не разработал стандарты для такой важной функциональности, а пользователи уже требуют ее, разработчики реляционных СУБД были вправе создавать свои собственные команды и синтаксис. Фактически многие из наиболее старых разработчиков сохранили с 80-х годов свои варианты самых элементарных команд (например, SELECT) поскольку их реализация предшествовала появлению стандарта. Сейчас ANSI производит уточнение стандартов, чтобы сгладить эти несоответствия.

В некоторых диалектах для обеспечения более полной функциональности языка программирования введены процедурные команды. Например, процедурные реализации содержат команды для обработки ошибок, операторы, контролирующие направление выполнения программы, условные команды, средства работы с переменными и массивами, а также многие другие дополнения. Хотя все эти процедурные реализации являются технической дивергенцией языка, здесь они называются диалектами. Пакет SQL/PSM (Persistent Stored Module) предлагает широкий спектр функциональности, связанной с хранимыми программными процедурами, и включает в себя многие расширения, содержащиеся в таких диалектах.

Вот некоторые популярные диалекты SQL:

- **PL/SQL.** Используется в Oracle. PL/SQL – это сокращение от Procedural Language/SQL. Он во многом похож на язык Ada.
- **Transact-SQL.** Используется в Microsoft SQL Server и Sybase Adaptive Server. По мере того как Microsoft и Sybase все больше отходят от общей платформы, которую они использовали в начале 90-х годов, их реализации Transact-SQL также подвергаются дивергенции.
- **PL/pgSQL.** Название диалекта и расширений SQL, реализованных в PostgreSQL. Является сокращением от Procedural Language/postgreSQL.
- **SQLPL.** Самый новый диалект от DB2 (SQLProcedural Language). Основан на стандартных операторах

управления SQL. Большинство других диалектов предшествовало стандарту, и это означает, что вы найдете в них массу отличий от стандарта SQL.

8.2. Язык SQL

Язык SQL обладает следующими достоинствами:

1. независимость от конкретных СУБД. Если при создании БД не использовались нестандартные возможности языка SQL предоставляемые некоторой СУБД, то такую БД можно без изменений перенести на СУБД другого производителя. К сожалению большинство БД используют особенности СУБД, на которой работают, что затрудняет их перенос на другую СУБД без изменений;
2. реляционная основа. Реляционная модель имеет солидный теоретический фундамент. Язык SQL основан на реляционной модели и является единственным языком для реляционных БД;
3. SQL обладает высокоуровневой структурой, напоминающей английский язык.
4. SQL позволяет создавать различные представления данных для различных пользователей;
5. SQL является полноценным языком для работы с БД;
6. стандарты языка SQL. Официальный стандарт языка SQL опубликован ANSI и ISO в 1989 году и значительно расширен в 1992 году.

Чтобы начать использовать SQL, вы должны понять, как пишутся инструкции. Синтаксические конструкции SQL делятся на четыре основные категории:

- **Идентификаторы.** Представляют собой пользовательские или системные имена объектов баз данных, таких, как база данных, таблица, ограничение в таблице, столбцы таблицы, представления и т. п.
- **Константы.** Представляют собой созданные пользователем или системой строки, или значения, не являющиеся идентификаторами или ключевыми словами. Константы могут представлять собой строки, например, «hello», числа, например, «1234», даты, например «1 января 2002», или булевы значения, например TRUE.

- **Операторы.** Символы, показывающие, какое действие выполняется над одним или несколькими выражениями, чаще всего в инструкциях DELETE, INSERT, SELECT или UPDATE. Операторы также часто применяются для создания объектов базы данных.
- **Зарезервированные и ключевые слова.** Имеют специальный смысл для обработчика кода SQL. Например, SELECT, GRANT, DELETE или CREATE. Зарезервированные слова (Reserved words), обычно команды и инструкции SQL, нельзя использовать в качестве идентификаторов на данной платформе. Ключевые слова (keywords) - это слова, которые могут стать зарезервированными в будущем.

Идентификаторы

Помните, что СУРБД созданы на основе теории множеств. В терминологии ANSI кластеры содержат множество каталогов, каталоги содержат множество схем, схемы содержат множество объектов и т. д. В большинстве платформ применяются дополнительные термины: экземпляры (instances) содержат одну или несколько баз данных, базы данных содержат одну или несколько схем, схемы содержат одну или несколько таблиц, представлений, хранимых процедур и привилегий, связанных с каждым объектом. На каждом уровне структуры элементам необходимы уникальные имена (т. е., идентификаторы), чтобы к ним могли обращаться программы и системные процессы. Это означает, что каждый объект (будь то база данных, таблица, представление, столбец, индекс, ключ, триггер, хранимая процедура или ограничение) в СУРБД должен получить свой идентификатор. Если вы запускаете команду, которая создает объект базы данных, вы должны указать идентификатор (т. е. имя) этого нового объекта.

Есть два важных набора правил, которые опытный программист держит в уме при выборе имени объекта. *Соглашения об именах*

Сюда входят практические руководства или соглашения об именах, применение которых в конечном счете улучшает структуру базы и отслеживание данных. Они являются не столько требованиями SQL, сколько накопленным опытом практикующих программистов. *Правила создания идентификаторов*

Они определены в стандарте SQL и реализованы в платформах. Эти правила включают, например, такие параметры, как максимальная длина имени. Эти соглашения описываются ниже в этой главе применительно к каждому производителю.

Соглашения об именах

- *Выбирайте имя так, чтобы оно было осмысленным, наглядным и соответствовало назначению объекта.* Не называйте таблицу ХРОЗ, назовите лучше Expenses_2005, чтобы было ясно, что в ней содержатся расходы за 2005 год. Помните, что и другим людям скорее всего придется использовать таблицу и базу данных, и возможно, еще в течение долгого времени после того, как вы уйдете. Имена должны быть понятны с первого взгляда. У каждого производителя есть свои ограничения на длину имени объектов, но, как правило, имена могут быть достаточно длинными, чтобы любой мог понять их смысл.
- *Используйте в именах один и тот же регистр по всей базе.* Используйте для всех имен объектов базы либо верхний, либо нижний регистр. Некоторые серверы баз учитывают регистр, и использование смешения регистров может позже вызвать проблемы.
- *Будьте последовательны в использовании сокращений.* Как только сокращение выбрано, его нужно последовательно использовать по всей базе данных. Например, если вы используете EMP как сокращение слова EMPLOYEE, то используйте EMP по всей базе данных. Не используйте в одних местах EMP, а в других - EMPLOYEE.
- *Для удобства восприятия используйте полные, наглядные и осмысленные имена с символами подчеркивания*. Имя столбца UPPERCASEWITHLNDERSCORES не такое понятное, как UPPERCASE_WITH_UNDERSCORES.
- *Не помещайте название компании и продуктов в имена объектов баз данных.* Компании приобретают друг друга, и продукты меняют названия. Такие элементы преходящи, и их не нужно включать в имена объектов базы.
- *Не используйте слишком очевидные префиксы и суффикс.* Например, не используйте в качестве префикса имени базы

данных сочетание «DB_», а в качестве префикса всех представлений - «V_». Простые запросы к системным таблицам базы могут дать администратору или программисту базы сведения о том, к какому типу относится объект, который представляет идентификатор.

- *Не заполняйте все пространство, отведенное для имени объекта.* Если платформа позволяет использовать имя таблицы из 32 символов, попробуйте оставить хотя бы несколько пустых мест в конце. Некоторые платформы при манипуляции временными копиями таблиц иногда добавляют к именам таблиц пре-фиксы и суффиксы.
- *Не используйте идентификаторы с разделителями.* Иногда имена объектов заключают в двойные кавычки. (Стандарт ANSI называет такие имена идентификаторами с разделителями (delimited identifiers). Такое заключение идентификатора в кавычки позволяет создавать имена, которые могут оказаться сложными в использовании и которые впоследствии могут вызывать проблемы. Такие идентификаторы чувствительны к регистру. Например, вы можете включать в них пробелы, специальные символы, символы в разных регистрах и даже управляющие символы. Однако, поскольку некоторые инструменты, выпускаемые сторонними производителями (и даже производителем самой базы), могут не обрабатывать специальные символы в именах, широко использовать подобные идентификаторы не следует. Некоторые платформы разрешают использовать другие символы-ограничители, помимо двойных кавычек. Например, в SQL Server для обозначения идентификаторов с ограничителями применяются квадратные скобки [].

Правила создания идентификаторов

Идентификаторы должны быть уникальны в пределах своей области действия. Таким образом, в иерархии объектов имена баз данных не должны повторяться в пределах данного экземпляра сервера базы, а имена таблиц, представлений, функций, триггеров и хранимых процедур - уникальны в пределах данной схемы. С другой стороны, таблица и хранимая процедура могут иметь одно имя, поскольку они являются объектами разных типов. Имена

столбцов, ключей и индексов должны быть уникальны в пределах одной таблицы или представления и т. д. За более подробной информацией обращайтесь к документации платформы. В некоторых платформах уникальность идентификаторов является обязательным условием, а в других - нет. Например, платформа DB2 требует, чтобы все идентификаторы индексов были уникальны по всей базе данных, а SQL Server требует, чтобы идентификаторы индексов были уникальными только в пределах таблицы, к которой они относятся.

Помните, что для обхода некоторых этих правил можно использовать идентификаторы с ограничителями (т. е. имена объектов, заключенные в специальные символы-ограничители, обычно в двойные кавычки). В частности, идентификаторы с разделителями можно применять для того, чтобы давать имена с зарезервированными словами, или для того, чтобы использовать в имени обычно не употребляемые там символы. Например, чаше всего вы не можете использовать в имени таблицы знак процента (%). Однако, если это необходимо, вы можете его использовать, если будете всегда заключать это имя таблицы в двойные кавычки. Чтобы назвать таблицу expense%%ratios, нужно заключить это имя в кавычки - "expense%%ratios". Также помните, что в SQL2003 такие имена иногда называются идентификаторами с разделителями (delimited identifiers).

Константы

В SQL константами считаются любые числовые значения, строки символов, значения, связанные с представлением времени (дата и время), и булевы значения, которые не являются идентификаторами или ключевыми словами. Базы данных на основе SQL разрешают использовать в коде SQL различные константы. Допустимы большинство числовых, символьных и булевых типов данных, а также даты. Например, к числовым типам данных SQL Server можно (среди прочих) отнести типы INTEGER, REAL и MONEY. Таким образом, числовые константы могут выглядеть так.

```
30
-17
-853 3888
-6.66
```

```
$70000
2E5
7E-3
```

Как показывает приведенный пример, в SQL Server допустимы числа со знаком и без знака, в обычной и экспоненциальной записи. А поскольку в SQL Server есть денежный тип данных, в константы можно включать даже знак доллара. В численных константах SQL Server не разрешается использовать другие символы (за исключением 0123456789+ -$.Ee), поэтому не включайте в них запятые (или точки, применяемые в Европе). Большинство баз данных интерпретируют запятую в числовой константе как ограничитель элементов. Так, константа 3,000 будет интерпретироваться как 3 и отдельно 000.

Булевы значения, строковые константы и даты выглядят примерно так.:

```
TRUE
'Hello world!'
10CT-28-1966 22:14:30:00'
```

Строковые константы должны всегда заключаться в одинарные кавычки ('), которые являются стандартным ограничителем всех строковых констант. Символы в строковых константах не ограничиваются алфавитными символами. По сути, любой символ из набора символов можно представить в виде строковой константы. Все приведенные ниже выражения являются строковыми константами.

```
'1998'
'70,000 + 14000'
'Жил-был один человек из Нантакета'
'Oct 28, 1966'
```

Все приведенные примеры фактически являются совместимыми с типом данных CHARACTER. Не путайте строковую константу '1998' с числовой константой 1998. Когда только строковые константы связаны с типом данных CHARACTER, не стоит использовать их в арифметических вычислениях, не преобразовав их явным образом в числовой тип. Некоторые базы данных выполняют автоматическое преобразование строковых констант, содержащих числа, при

выполнении сравнения их с любыми значениями, относящимися к типам DATE или NUMBER.

При необходимости вы можете отобразить в строковой константе символ одинарной кавычки. Для этого его необходимо написать два раза; т. е., каждый раз, когда вам нужно написать внутри строки одинарную кавычку, вы должны написать две. Проиллюстрируем эту идею примером из SQL Server.

```
SELECT 'So he said  ''who''s Le Petomaine?'' '
```

Получится следующий результат.

```
So he said 'Who's Le Petomaine?'
```

Операторы

Оператор – это символ, обозначающий действие, выполняемое над одним или несколькими выражениями. Операторы наиболее часто используются в инструкциях DELETE, INSERT, SELECT или UPDATE, а также часто применяются при создании объектов базы данных, таких, как хранимые процедуры, функции, триггеры и представления.

Операторы, как правило, делятся на следующие категории:

1. *Арифметические операторы.* Поддерживаются всеми базами данных.
2. *Операторы присваивания.* Поддерживаются всеми базами данных.
3. *Побитовые операторы.* Поддерживаются Microsoft SQL Server.
4. *Операторы сравнения.* Поддерживаются всеми базами данных.
5. *Логические операторы.* Поддерживаются в DB2, Oracle, SQL Server и PostgreSQL. *Унарные операторы.* Поддерживаются в DB2, Oracle и SQL Server.

Арифметические операторы

Арифметические операторы выполняют математические действия над двумя значениями любого типа, относящегося к числовой категории. Перечень арифметических операторов приведен в табл. 8.1.

Таблица 8.1. Арифметические операторы.

Арифметический оператор	Действие
+	Сложение
-	Вычитание
*	Умножение
/	Деление
%	Остаток отделения (только SQL Server). Возвращает остаток от операции деления в виде целого числа (integer)

Операторы присваивания

За исключением Oracle, где для этой цели применяется оператор :- оператор присваивания (=) присваивает значение переменной или псевдониму (alias) заголовка столбца. В SQL Server в качестве оператора для присваивания псевдонимов таблицам или заголовкам столбцов может служить ключевое слово AS.

Побитовые операторы

В Microsoft SQL Server существуют побитовые операторы, являющиеся удобным средством манипулирования битами в двух выражениях целого типа (см. табл. 8.2). Для побитовых операторов доступны следующие типы данных: *binary, hit, int, smallint, tinyint u varbinary*.

Таблица 8.2. Побитовые операторы

Побитовый оператор	Действие
&	Поразрядное И (два операнда)
|	Поразрядное ИЛИ (два операнда)
^	Поразрядное исключающее ИЛИ (два операнда)

Операторы сравнения

Операторы сравнения проверяют равенство или неравенство двух выражений. Результатом операции сравнения является булево значение: TRUE, FALSE или UNKNOWN. Также заметьте, что по стандарту ANSI сравнение выражений, когда одно или оба значения равны NULL, дает результат NULL. Например,

выражение 23 + NULL дает NULL, как и выражение Feb 23, 2003 + NULL.

Таблица 8.3. Операторы сравнения

Оператор сравнения	Действие
=	Равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно
<>	Не равно
! =	Не равно (не соответствует стандарту ANSI)
!<	Не меньше (не соответствует стандарту ANSI)
!>	Не больше (не соответствует стандарту ANSI)

Логические операторы

Логические операторы обычно применяются в предложении WHERE для проверки истинности какого-либо условия. Логические операторы возвращают булево значение TRUE или FALSE. Логические операторы также обсуждаются в разделе «Инструкция SELECT» главы 3. Не все базы данных поддерживают все операторы. Список логических операторов приведен в табл. 8.4.

Таблица 8.4. Логические операторы

Логический оператор	Действие
ALL	TRUE, если весь набор сравнений дает результат TRUE
AND	TRUE, если оба булевых выражения дают результат TRUE
ANY	TRUE, если хотя бы одно сравнение из набора дает результат TRUE
BETWEEN	TRUE, если операнд находится внутри диапазона
EXISTS	TRUE, если подзапрос возвращает хотя бы одну строку

IN	TRUE, если операнд равен одному выражению из списка или одной или нескольким строкам, возвращаемым подзапросом
LIKE	TRUE, если операнд совпадает с шаблоном
NOT	Обращает значение любого другого булевого оператора
OR	TRUE, если любое булево выражение равно TRUE
SOME	TRUE, если несколько сравнений из набора дают результат TRUE

Унарные операторы

Унарные операторы выполняют операцию над одним выражением любого типа, относящимся к числовой категории. Унарные операторы можно применять к целым типам, хотя операторы положительности и отрицательности можно применять к любому числовому типу данных (см. табл. 8.5).

Таблица 8.5. Унарные операторы

Унарный оператор	**Действие**
+	Числовое значение становится положительным
-	Числовое значение становится отрицательным
~	Поразрядное НЕ, возвращает двоичное дополнение числа (нет в Oracle и DB2)

Приоритет операторов

Иногда выражения, включающие операторы, могут быть довольно сложными. Когда в выражении присутствуют несколько операторов, последовательность их выполнения определяет приоритет операторов. Порядок выполнения может существенно повлиять на результат вычисления.

Ниже перечислены уровни приоритета операторов. Оператор с более высоким приоритетом выполняется до оператора с более низким приоритетом. В списке операторы перечислены в порядке от самого высокого к самому низкому приоритету. () (выражения, стоящие в скобках)

1. +, -, ~ (унарные операторы)
2. *, /, % (математические операторы)
3. +, - (арифметические операторы)
4. =, >, <, >=, <=, <>, !=, !>, !< (операторы сравнения)

5. ^ (побитовое исключающее ИЛИ), & (побитовое И), | (побитовое ИЛИ)
6. NOT, AND, ALL, ANY, BETWEEN IN LIKE, OR, SOME = (присваивание значение переменной)

Если операторы имеют одинаковый приоритет, вычисления производятся слева направо. Для того чтобы изменить применяемый по умолчанию порядок выполнения операторов, в выражении используются скобки. Выражения, заключенные в скобки, вычисляются первыми, а уже после них - все, что находится за скобками.

Строковые ограничители обозначают границы строки, состоящей из буквенно-цифровых символов. ***Системные ограничители*** - это такие символы из набора символов, которые имеют для сервера базы данных особое значение. ***Ограничители*** - это символы, которые применяются для определения иерархического порядка процессов и элементов списка. ***Операторы*** - это ограничители, используемые для определения значений при операциях сравнения, в том числе символы, обычно используемые для арифметических и математических операций. В табл. 6 перечислены системные ограничители и операторы, допустимые в SQL.

8.3. Ключевые и зарезервированные слова

Наряду с символами, которые имеют особый смысл и функции в SQL, существуют и некоторые слова и фразы, имеющие особую значимость. ***Ключевые слова SQL*** - это слова, которые настолько тесно связаны с функционированием реляционной базы данных, что их нельзя использовать ни для каких других целей. Как правило, такие слова используются в инструкциях SQL. (Заметьте, что на большинстве платформ их можно использовать в качестве идентификаторов, хотя этого делать не следует.) Например, слово «SELECT» является зарезервированным словом, и его не следует использовать в качестве имени таблицы.

С другой стороны, зарезервированные слова в настоящий момент не имеют специального назначения, но они могут приобрести его в будущем. Чтобы подчеркнуть тот факт, что ключевые слова не следует использовать в качестве идентификаторов, но тем не менее такая возможность существует,

в стандарте SQL они называются «незарезервированными ключевыми словами». Зарезервированные и ключевые слова SQL не всегда представляют собой слова, используемые в инструкциях SQL, они также могут быть связаны с технологией использования базы данных. Например, слово CASCADE применяется для описания таких манипуляций с данными, в которых действие (например, удаление или обновление) распространяется «вниз», т. е. «каскадом» на все нижележащие таблицы. Зарезервированные и ключевые слова часто публикуются, чтобы программисты не использовали их в качестве идентификаторов и чтобы в дальнейшем, в следующих версиях, не возникали проблемы.

Основные категории команд языка SQL:

- DDL – язык определения данных;
- DML – язык манипулирования данными;
- DQL – язык запросов;
- DCL – язык управления данными;
- команды администрирования данных;
- команды управления транзакциями.

Чаще всего выделяются два языка - язык определения схемы БД (SDL - Schema Definition Language) и язык манипулирования данными (DML - Data Manipulation Language). SDL служил главным образом для определения логической структуры БД, т.е. той структуры БД, какой она представляется пользователям. DML содержал набор операторов манипулирования данными, т.е. операторов, позволяющих заносить данные в БД, удалять, модифицировать или выбирать существующие данные.

Data Definition Language (DDL) (язык описания данных) — это семейство компьютерных языков, используемых в компьютерных программах для описания структуры баз данных.

Функции языков DDL определяются первым словом в предложении (часто называемом запросом), которое почти всегда является глаголом. В случае с SQL это глаголы — «create» («создать»), «alter» («изменить»), «drop» («удалить»). Эти запросы или команды часто смешиваются с другими командами SQL, в связи с чем DDL не является отдельным компьютерным языком.

Запрос «create» используется для создания базы данных, таблицы, индекса, представления или хранимой процедуры. Запрос «alter» используется для изменения существующего

объекта базы данных (таблицы, индекса, представления или хранимой процедуры) или самой базы данных. Запрос «drop» используется для удаления существующего объекта базы данных (таблицы, индекса, представления или хранимой процедуры) или самой базы данных. И наконец, в DDL существуют понятия первичного и внешнего ключа, которые осуществляют соблюдение целостности данных. Команды "первичный ключ" primary key, "внешний ключ" foreign key включаются в запросы «create table", «alter table».

Data Manipulation Language (DML) (язык управления (манипулирования) данными) — это семейство компьютерных языков, используемых в компьютерных программах или пользователями баз данных для получения, вставки, удаления или изменения данных в базах данных.

Языки DML изначально использовались только компьютерными программами, но с появлением SQL стали также использоваться и людьми.

Функции языков DML определяются первым словом в предложении (часто называемом запросом), которое почти всегда является глаголом. В случае с SQL эти глаголы — «select» («выбрать»), «insert» («вставить»), «update» («обновить»), и «delete» («удалить»). Это превращает природу языка в ряд обязательных утверждений (команд) к базе данных.

Языки DML могут существенно различаться у различных производителей СУБД. Существует стандарт SQL, установленный ANSI, но производители СУБД часто предлагают свои собственные «расширения» языка.

Языки DML разделяются в основном на два типа:

- Procedural DMLs — описывают действия над данными.
- Declarative DMLs — описывают сами данные.

Контрольные вопросы:

1. Как развивался язык запросов SQL?
2. В чем основные преимущества языка SQL?
3. Какие категории языков объединяет в себе SQL?
4. Что такое оператор?
5. Какие виды констант существуют в SQL?
6. Какие группы опертаоров языка SQL вы знаете?
7. Что такое идентификатор?

8. Что такое язык опреления данных?
9. Что такое язык манипулирование данных?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p.
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p.

Глава IX. SQL: манипулирование данными

9.1 Язык манипулирования данными

В основу языка манипулирования данными входят 4 основных оператора:

1. SELECT – используется для выборки записей из таблиц;
2. INSERT –используется для добавления записей в таблицу;
3. UPDATE – используется для обновления записей таблицы;
4. DELETE – используется для удаления записей из таблицы.

В самой простой форме, команда SELECT просто инструктирует базу данных чтобы извлечь информацию из таблицы.

Оператор INSERT.

Все строки в SQL вводятся с использованием команды модификации INSERT. В самой простой форме, INSERT использует следующий синтаксис:

```
INSERT INTO <имя_таблицы> [(<имя столбца_1> [,
<имя столбца_1> ...])]
   {VALUES (<значение_1> [, <значение_2> ...])
| <выражение SELECT>};
```

Так, например, чтобы ввести строку в таблицу Продавцов, вы можете использовать следующее условие:

```
INSERT INTO Salespeople VALUES (1001, 'Peel',
'London', .12);
```

Команды DML не производят никакого вывода, но ваша программа должна дать вам некоторое подтверждение того что данные были использованы.

Вы можете также указывать столбцы, куда вы хотите вставить значение имени. Это позволяет вам вставлять имена в любом порядке. Предположим, что вы берете значения для таблицы Заказчиков из отчета выводимого на принтер, который помещает их в таком порядке: city, cname, и cnum, и для упрощения, вы хотите ввести значения в том же порядке:

```
INSERT INTO Customers (city, cname, cnum)
VALUES ('London', 'Honman', 2001);
```

Обратите внимание что столбцы rating и snum - отсутствуют. Это значит, что эти строки автоматически установлены в значение - по умолчанию. По умолчанию может быть введено или значение NULL или другое значе- ние определяемое как - по умолчанию. Если ограничение запрещает использование значения NULL в данном столбце, и этот столбец не установлен как по умолчанию, этот столбец должен быть обеспечен значением для любой команды INSERT которая относится к таблице.

Оператор UPDATE.

Теперь, вы должны узнать как изменять некоторые или все значения в существующей строке. Это выполняется командой UPDATE. Эта команда содержит предложение UPDATE в которой указано им используемой таблицы и предложение SET которое указывает на изменение которое нужно сделать для определенного столбца. Например, чтобы изменить оценки всех заказчиков на 200, вы можете ввести

```
UPDATE TABLE <имя_таблицы>
   SET <имя столбца_1> = <значение_1> [, <имя
столбца_2> = <значение_2> ...]
   [WHERE <условие>];
```

Например

```
UPDATE Customers
   SET rating = 200;
```

Конечно, вы не всегда захотите указывать все строки таблицы для изменения единственного значения, так что UPDATE может брать предикаты. Вот как например можно выполнить

изменение одинаковое для всех заказчиков продавца Peel (имеющего snum=1001):

```
UPDATE Customers
   SET rating = 200
   WHERE snum = 1001;
```

Однако, вы не должны, ограничивать себя модифицированием единственного столбца с помощью команды UPDATE. Предложение SET может назначать любое число столбцов, отделяемых запятыми. Все указанные назначения могут быть сделаны для любой табличной строки, но только для одной в каждый момент времени. Предположим, что продавец Motika ушел на пенсию, и мы хотим переназначить его номер новому продавцу:

```
UPDATE Salespeople
   SET sname = 'Gibson',city = 'Boston',comm =
.10
   WHERE snum = 1004;
```

Вы можете использовать скалярные выражения в предложении SET команды UPDATE, однако, включив его в выражение пол которое будет изменено. В этом их отличие от предложения VALUES команды INSERT, в котором выражения не могут использоваться; это свойство скалярных выражений - весьма полезна особенность. Предположим, что вы решили удвоить комиссионные всем вашим продавцам. Вы можете использовать следующее выражение:

```
UPDATE Salespeople
   SET comm = comm * 2;
```

Оператор DELETE.

Вы можете удалять строки из таблицы командой модификации - DELETE. Она может удалять только введенные строки, а не индивидуальные значения полей.

```
DELETE FROM table [WHERE <search_condition>];
```

Теперь когда таблица пуста ее можно окончательно удалить командой DROP TABLE. Обычно, вам нужно удалить только некоторые определенные строки из таблицы. Чтобы определить какие строки будут удалены, вы используете предикат, так же как

вы это делали для запросов. Например, чтобы удалить продавца под номером 1003 из таблицы, вы можете ввести

```
DELETE FROM Salespeople
   WHERE snum = 1003;
```

Мы использовали поле snum вместо пол sname потому, что это лучшая тактика при использовании первичных ключей когда вы хотите чтобы действию подвергалась одна и только одна строка. Для вас - это аналогично действию первичного ключ. Конечно, вы можете также использовать DELETE с предикатом который бы выбирал группу строк, как показано в этом примере:

```
DELETE FROM Salespeople
   WHERE city = 'London';
```

Оператор SELECT

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

```
SELECT [[ALL] | DISTINCT]{ * | элемент_SELECT
[,элемент_SELECT] ...}
FROM       {базовая_таблица | представление}
[псевдоним]
       [,{базовая_таблица | представление}
[псевдоним]] ...
[WHERE фраза]
[GROUP BY фраза [HAVING фраза]];
[ORDER BY фраза]
```

Если вы хотите видеть каждый столбец таблицы, имеется необязательное сокращение которое вы можете использовать. Звездочка (*) может применяться для вывода полного списка столбцов следующим образом:

```
SELECT * FROM Salespeople;
```

9.2. Агрегатные функции языка SQL и группировка

Агрегатные функции предназначены для того, чтобы вычислять некоторое значение для заданного множества строк. Таким множеством строк может быть группа строк, если

агрегатная функция применяется к сгруппированной таблице, или вся таблица. В SQL имеется пять агрегатных функций, которые позволяют получать различные виды итоговой информации. Ниже описан синтаксис этих функций:

1. COUNT – возвращает количество значений в указанном столбце;
2. SUM – возвращает сумму значений в указанном столбце;
3. AVG – возвращает усредненное значение в указанном столбце;
4. MIN – возвращает минимальное значение в указанном столбце;
5. MAX – возвращает максимальное значение в указанном столбце.

Для всех агрегатных функций, кроме COUNT(*), фактический (т.е. требуемый семантикой) порядок вычислений следующий: на основании параметров агрегатной функции из заданного множества строк производится список значений. Затем по этому списку значений производится вычисление функции. Если список оказался пустым, то значение функции COUNT для него есть 0, а значение всех остальных функций null.

Агрегатные функции используются подобно именам полей в предложении SELECT запроса, но с одним исключением, они берут имена поля как аргументы. Только числовые поля могут использоваться с SUM и AVG. С COUNT, MAX, и MIN, могут использоваться и числовые или символьные поля. Когда они используются с символьными полями, MAX и MIN будут транслировать их в эквивалент ASCII, который должен сообщать, что MIN будет означать первое, а MAX последнее значение в алфавитном порядке.

Чтобы найти SUM всех наших покупок в таблицы Порядков, мы можем ввести следующий запрос, с его выводом в рисунке 9.1:

```
SELECT SUM ((amt))
FROM Orders;
```

```
==============  SQL Execution Log ============
                        |
                        |
                    | -------
                        |
```

```
                    | 26658.4
                        |
                        |
                        |
=============================================
```

Следует отметить, что агрегирующие функции могут использоваться только в списке выборки SELECT и в конструкции HAVING. Во всех других случаях применение этих функций недопустимо. Если список выборки SELECT содержит агрегирующую функцию, а в тексте запроса отсутствует конструкция GROUP BY, обеспечивающая объединение данных в группы, то ни один из элементов списка выборки SELECT не может включать каких-либо ссылок на столбцы, за исключением случая, когда этот столбец используется как параметр агрегирующей функции. Например, следующий запрос является некорректным:

```
SELECT staffNo, COUNT(salary) FROM Staff;
```

Ошибка состоит в том, что в данном запросе отсутствует конструкция GROUP BY, а обращение к столбцу staffNo в списке выборки SELECT выполняется без применения агрегирующей функции.

Пример:

```
SELECT MIN(salary) AS min, MAX(salary) AS max,
AVG(salary) AS avg FROM Staff;
 ==============  SQL Execution Log ============
                        |
                        |
                | min   |  max   |  avg
                        |
               |  9000    30000    17000
                        |
                        |
                        |
=============================================
```

Запросы с группировкой (предложение GROUP BY)

Приведенные выше примеры сводных данных подобны итоговым строкам, обычно размещаемым в конце отчетов. В

итогах все детальные данные отчета сжимаются в одну обобщающую строку. Однако очень часто в отчетах требуется формировать и промежуточные итоги. Для этой цели в операторе SELECT может указываться конструкция GROUP BY. Запрос, в котором присутствует конструкция GROUP BY, называется группирующим запросом, поскольку в нем группируются данные, полученные в результате выполнения операции SELECT, после чего для каждой отдельной группы создается единственная итоговая строка. Столбцы, перечисленные в конструкции GROUP BY, называются группируемыми столбцами. Стандарт ISO требует, чтобы конструкции SELECT и GROUP BY были тесно связаны между собой. При использовании в операторе SELECT конструкции GROUP BY каждый элемент списка в списке выборки SELECT должен иметь единственное значение для всей группы. Более того, конструкция SELECT может включать только следующие типы элементов:

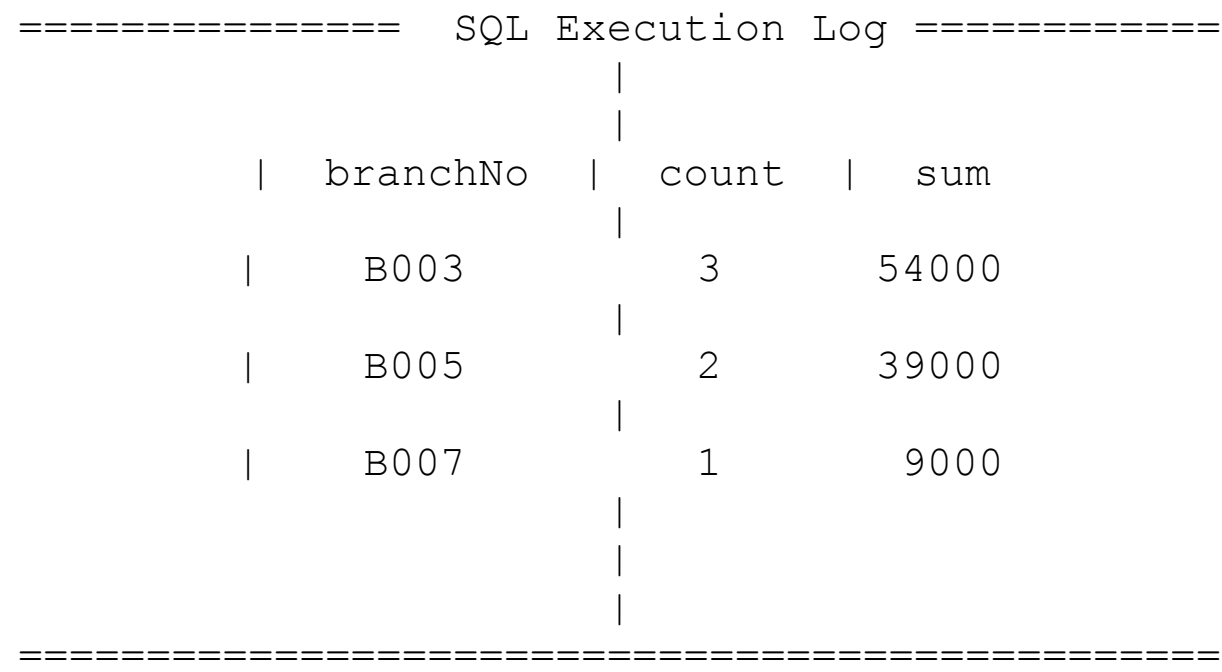
- имена столбцов;
- агрегирующие функции;
- константы;
- выражения, включающие комбинации перечисленных выше элементов.

Все имена столбцов, приведенные в списке выборки SELECT, должны присутствовать и в конструкции GROUP BY, за исключением случаев, когда имя столбца используется только в агрегирующей функции.

Противоположное утверждение не всегда справедливо — в конструкции GROUP BY могут присутствовать имена столбцов, отсутствующие в списке выборки SELECT. Если совместно с конструкцией GROUP BY используется конструкция WHERE, то она обрабатывается в первую очередь, а группированию подвергаются только те строки, которые удовлетворяют условию поиска. Стандартом ISO определено, что при проведении группирования все отсутствующие значения рассматриваются как равные. Если две строки таблицы в одном и том же группируемом столбце содержат значения NULL и идентичные значения во всех остальных непустых группируемых столбцах, они помещаются в одну и ту же группу.

Например:

```
SELECT branchNo, COUNT{staffNo} AS count,
SUM(salary) AS sum FROM Staff GROUP BY branchNo
ORDER BY branchNo;
```



Концептуально при обработке этого запроса выполняются следующие действия.

1. Строки таблицы Staff распределяются в группы в соответствии со значениями в столбце номера отделения компании. В пределах каждой из групп оказываются данные обо всем персонале одного из отделений компании. В нашем примере будут созданы три группы, как показано на рис. 9.2.
2. Для каждой из групп вычисляются общее количество строк, равное количеству работников отделения, а также сумма значений в столбце salary, которая и является интересующей нас суммой заработной платы всех работников отделения. Затем генерируется единственная итоговая строка для всей группы исходных строк.
3. Полученные строки результирующей таблицы сортируются в порядке возрастания номера отделения, указанного в столбце branchNo.

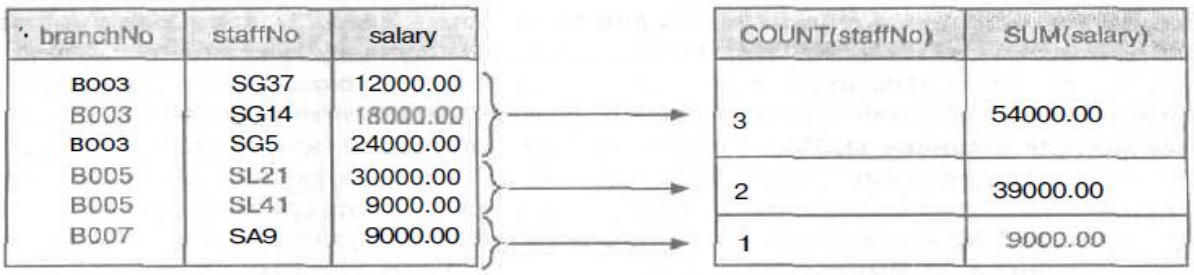


Рис. 9.2. Три группы записей, создаваемых при запросе.

Предложение HAVING.

Предположим, что в предыдущем примере, вы хотели бы увидеть только максимальную сумму приобретений значение которой выше $3000.00. Вы не сможете использовать агрегатную функцию в предложении WHERE (если вы не используете подзапрос, описанный позже), потому что предикаты оцениваются в терминах одиночной строки, а агрегатные функции оцениваются в терминах групп строк. Это означает, что вы не сможете сделать что-нибудь подобно следующему:

```
SELECT snum, odate, MAX (amt)
FROM Oreders
WHERE MAX ((amt)) > 3000.00
GROUP BY snum, odate;
```

Это будет отклонением от строгой интерпретации ANSI. Чтобы увидеть максимальную стоимость приобретений свыше $3000.00, вы можете использовать предложение HAVING. Предложение HAVING определяет критерии, используемые чтобы удалять определенные группы из вывода, точно также как предложение WHERE делает это для индивидуальных строк. Правильной командой будет следующая:

```
SELECT snum, odate, MAX(amt)
FROM Orders
GROUP BY snum, odate
HAVING MAX ((amt)) > 3000.00;
       ===============  SQL Execution Log
                  =============
                        |
                        |
```

```
|    snum          odate          MAX(amt)
                     |
 |   ------      ----------      --------
                     |
 |    1001       10/05/1990       4723.00
                     |
 |    1001       10/06/1990       9891.88
                     |
 |    1002       10/03/1990       5160.45
                     |
                     |
                     |
 ==========================================
                     =
```

Аргументы в предложении HAVING следуют тем же самым правилам, что и в предложении SELECT, состоящей из команд использующих GROUP BY. Они должны иметь одно значение на группу вывода. Следующая команда будет запрещена:

```
SELECT snum, MAX (amt)
FROM Orders
GROUP BY snum
HAVING odate = 10/03/1988;
```

Поле odate не может быть вызвано предложением HAVING, потому что оно может иметь (и действительно имеет) больше чем одно значение на группу вывода. Чтобы избегать такой ситуации, предложение HAVING должно ссылаться только на агрегаты и поля выбранные GROUP BY. Имеется правильный способ сделать вышеупомянутый запрос:

```
SELECT snum, MAX (amt)
FROM Orders
WHERE odate = 10/03/1990
GROUP BY snum;
       ===============  SQL Execution Log
                  ==============
                        |
                        |
                    |    snum
                        |
```

```
|  ------      --------
                |
|   1001          767.19
                |
|   1002         5160.45
                |
|   1014         1900.10
                |
|   1007         1098.16
                |
                |
                |
==========================================
                          =
```

Как и говорилось ранее, HAVING может использовать только аргументы, которые имеют одно значение на группу вывода. Практически, ссылки на агрегатные функции – наиболее общие, но и поля, выбранные с помощью GROUP BY также допустимы. Например, мы хотим увидеть наибольшие порядки для Serres и Rifkin:

```
SELECT snum, MAX (amt)
FROM Orders
GROUP BY snum
HAVING snum IN (1002,1007);
       ==============  SQL Execution Log
                  ==============
                        |
                        |
                    |   snum
                        |
|  ------      --------
                |
|   1002         5160.45
                |
|   1007         1098.16
                |
                |
                |
```

==========
=

SQL позволяет группировать результаты запроса на основании двух или более столбцов. Например, получить список фамилий студентов и их средних оценок за каждый семестр.

```
SELECT StName, Semester, AVG(Mark)
FROM Marks INNER JOIN Students USING(StNo)
GROUP BY StName, Semester;
```

9.3. Ограничения на запросы с группировкой

На запросы, в которых используется группировка, накладываются дополнительные ограничения. Столбцы с группировкой должны представлять собой реальные столбцы таблиц, перечисленных в предложении FROM. Нельзя группировать строки на основании значения вычисляемого выражения.

Кроме того, существуют ограничения на элементы списка возвращаемых столбцов. Все элементы этого списка должны иметь одно значение для каждой группы строк. Это означает, что возвращаемым столбцом может быть:

1. константа;
2. агрегатная функция, возвращающая одно значение для всех строк, входящих в группу;
3. столбец группировки, который, по определению, имеет одно и то же значение во всех строках группы;
4. выражение, включающее в себя перечисленные выше элементы.

На практике в список возвращаемых столбцов запроса с группировкой всегда входят столбец группировки и агрегатная функция. Если последняя не указана, значит, запрос можно более просто выразить с помощью ключевого слова DISTINCT без использования предложения GROUP BY. И наоборот, если не включить в результаты запроса столбец группировки, вы не сможете определить, к какой группе относится каждая строка результатов.

В строгой интерпретации ANSI SQL, вы не можете использовать агрегат агрегата. Предположим? что вы хотите выяснить, в какой день имелась наибольшая сумма приобретений.

```
SELECT odate, MAX ( SUM (amt) )
FROM Orders
GROUP BY odate;
```

Если вы попробуете сделать это, то ваша команда будет вероятно отклонена. (Некоторые реализации не предписывают этого ограничения, которое является выгодным, потому что вложенные агрегаты могут быть очень полезны, даже если они и несколько проблематичны.) В вышеупомянутой команде, например, SUM должен применяться к каждой группе поля odate, а MAX ко всем группам, производящим одиночное значение для всех групп. Однако предложение GROUP BY подразумевает что должна иметься одна строка вывода для каждой группы поля odate.

Соединение таблиц.

В стандарте SQL-92 был определен совершенно новый метод поддержки внешних объединений, который не опирался ни на одну популярную СУБД. В спецификации стандарта поддержка внешних Соединений осуществлялась в предложении FROM с тщательно разработанным синтаксисом, позволявшим пользователю точно определить, как исходные таблицы должны быть соединены в запросе.

Операция соединения используется в языке SQL для вывода связанной информации, хранящейся в нескольких таблицах, в одном запросе. В этом проявляется одна из наиболее важных особенностей запросов SQL - способность определять связи между многочисленными таблицами и выводить информацию из них в рамках этих связей. Именно эта операция придает гибкость и легкость языку SQL.

Операции соединения подразделяются на два вида - внутренние и внешние. Оба вида соединений задаются в предложении WHERE запроса SELECT с помощью специального условия соединения. Внешние соединения (о которых мы поговорим позднее) поддерживаются стандартом ANSI-92 и содержат зарезервированное слово "JOIN", в то время как внутренние соединения (или просто соединения) могут задаваться

как без использования такого слова (в стандарте ANSI-89), так и с использованием слова "JOIN" (в стандарте ANSI-92).

Связывание при соединении таблиц производится, как правило, по первичному ключу одной таблицы и внешнему ключу другой таблицы - для каждой пары таблиц. При этом очень важно учитывать все поля внешнего ключа, иначе результат будет искажен. Соединяемые поля могут (но не обязаны!) присутствовать в списке выбираемых элементов. Предложение WHERE может содержать множественные условия соединений. Условие соединения может также комбинироваться с другими предикатами в предложении WHERE.

Соединение является подмножеством более общей комбинации данных двух таблиц, называемой декартовым произведением. Декартово произведение двух таблиц представляет собой другую таблицу, состоящую из всех возможных пар строк, входящих в состав обеих таблиц. Набор столбцов результирующей таблицы представляет собой все столбцы первой таблицы, за которыми следуют все столбцы второй таблицы. Если ввести запрос к двум таблицам без задания конструкции WHERE, результат выполнения запроса в среде SQL будет представлять собой декартовое произведение этих таблиц.

Процедура генерации таблицы, содержащей результаты соединения двух таблиц с помощью оператора SELECT, состоит в следующем.

1. Формируется декартово произведение таблиц, указанных в конструкции FROM.
2. Если в запросе присутствует конструкция WHERE, применение условий поиска к каждой строке таблицы декартова произведения и сохранение в таблице только тех строк, которые удовлетворяют заданным условиям. В терминах реляционной алгебры эта операция называется ограничением декартового произведения.
3. Для каждой оставшейся строки определяется значение каждого элемента, указанного в списке выборки SELECT, в результате чего формируется отдельная строка результирующей таблицы.
4. Если в исходном запросе присутствует конструкция SELECT DISTINCT, из результирующей таблицы удаляются все

строки-дубликаты. В реляционной алгебре действия, выполняемые на 3 и 4 этапах, эквивалентны операции проекции по столбцам, заданным в списке выборки SELECT.

5. Если выполняемый запрос содержит конструкцию ORDER BY, осуществляется переупорядочивание строк результирующей таблицы.

Внутреннее соединение в стандарте SQL-92

Внутреннее соединение возвращает только те строки, для которых условие соединения принимает значение true.

```
SELECT * | <имена_столбцов> FROM
<имя_таблицы_1>  INNER JOIN <имя_таблицы_2> ON
<имя_столбца_в_1_таблице> =
<имя_столбца_во_2_таблице>
WHERE <условие>
ORDER BY <имена_столбцов>;
```

или

```
SELECT * | <имена_столбцов> FROM
<имя_таблицы_1>, <имя_таблицы_2> WHERE
<имя_столбца_в_1_таблице> =
<имя_столбца_во_2_таблице>
ORDER BY <имена_столбцов>;
```

В первом случае две таблицы соединяются явно посредством операции JOIN, а условие поиска, описывающее объединение, находится теперь в предложении ON внутри предложения FROM. В условии поиска, следующем за ключевым словом ON, могут быть заданы любые критерии сравнения строк двух объединяемых таблиц. Во втором случае условие соединения идет вместе с предикатом после ключевого слова WHERE.

Таблица 9.1. Branch

branchNo	bCity	cityID
B004	Bristol	1
B003	Glasgow	3
B002	London	2

Таблица 9.2. PropertyForRen

propertyNo	pCity	cityID

PA14	Aberdeen	4
PL94	London	2
PG4	Glasgow	3

Обычное (внутреннее) соединение этих таблиц выполняется с помощью следующего оператора SQL:

```
SELECT b.*, p.*
FROM Branch b, PropertyForRent p
WHERE b.bCity = p.pCity;
```

Или в соответствии со стандартом SQL-92 данный запрос буде выглядеть следующим образом:

```
SELECT b.*, p.*
FROM Branch b INNER JOIN PropertyForRent p
ON b.bCity = p.pCity;
```

```
     ====================  SQL Execution Log
              ======================
                          |
                          |
   |  branchNo   bcity     cityID   propertyNo
               pCity     cityID   |
 |  --------   -------   ------   ----------  --
                -----   ------   |
     |    B003     Glasgow     3         PG4
               Glasgow     3       |
     |    B002     London      2         PL94
               London      2       |
                          |
                          |
 ================================================
                  ===============
```

Как можно видеть, в результирующей таблице запроса имеются только две строки, содержащие одинаковые названия городов, выбранные из обеих таблиц. Обратите внимание, что в исходных данных нет соответствия для отделения компании в Bristol и для объекта, сдаваемого в аренду в городе Aberdeen. Как видно из примера соединение можно выполнять не только по полям, содержащим первичный и вторичный ключи.

Объединение двух таблиц, в котором связанные столбцы имеют идентичные имена, называется ***естественным объединением***, так как обычно это действительно самый "естественный" способ объединения двух таблиц.

```
SELECT * | <имена_столбцов> FROM
<имя_таблицы_1>  NATURAL JOIN <имя_таблицы_2>
ON (<имя_столбца>);
```

Например естественное соединение таблиц Branch и PropertyForRent выглядит следующим образом:

```
SELECT b.*, p.*
FROM Branch b NATURAL JOIN PropertyForRent p
ON (cityID);
```

Внешние соединения в стандарте SQL-92.

Во внешнем соединении в результирующую таблицу помещаются также строки, не удовлетворяющие условию соединения. Чтобы понять особенности выполнения операций внешнего соединения, воспользуемся упрощенными таблицами Branch и PropertyForRent, содержимое которых представлены выше.

Существуют три типа внешнего соединения:

1. левое;
2. правое;
3. полное.

Используем левое внешнее соединение этих двух таблиц, которое выглядит следующим образом:

```
SELECT b.*, p.*
FROM Branch b LEFT JOIN PropertyForRent p ON
b.bCity = p.pCity;
```

```
    ====================  SQL Execution Log
              ======================
                          |
                          |
   |  branchNo   bcity      cityID   propertyNo
              pCity     cityID    |
 |  --------   -------   ------   ----------  --
               -----   ------   |
```

```
    |    B003     Glasgow     3          PG4
              Glasgow     3       |
    |    B004     Bristol     1          NULL
              NULL        NULL    |
    |    B002     London      2          PL94
              London      2       |
                        |
                        |
=============================================
                ================
```

В этом примере за счет применения левого внешнего соединения в результирующую таблицу попали не только две строки, в которых имеется соответствие между названиями городов, но также та строка первой из соединяемых таблиц (левой), которая не нашла себе соответствия во второй таблице (правой). В этой строке все поля второй таблицы заполнены значениями NULL.

Используем правое внешнее соединение этих двух таблиц, которое выглядит следующим образом:

```
SELECT b.*, p.*
FROM Branch b RIGHT JOIN PropertyForRent p ON
b.bCity = p.pCity;
```

```
    ======================  SQL Execution Log
              ======================
                        |
                        |
   |  branchNo   bcity      cityID   propertyNo
              pCity     cityID    |
 |  --------   -------   ------   ----------  --
               -----   ------   |
    |    NULL     NULL        NULL        PA14
              Aberdeen    4       |
     |    B003     Glasgow     3          PG4
              Glasgow     3       |
    |    B002     London      2          PL94
              London      2       |
                        |
                        |
```

```
==========================================
==========
```

В этом примере при выполнении правого внешнего соединения в результирующую таблицу были включены не только те две строки, которые имеют одинаковые значения в сопоставляемых столбцах с названием города, но также и те строки из второй (правой) таблицы, которые не нашли соответствия со строками в первой (левой) таблице. В этой строке все поля из первой таблицы получили значения NULL.

Используем полное внешнее соединение этих таблиц, которое выглядит следующим образом:

```
SELECT b.*, p.*
FROM Branch b FULL JOIN PropertyForRent p ON
b.bCity = p.pCity;
```

```
    ======================  SQL Execution Log
            ====================
                        |
                        |
  |  branchNo   bcity     cityID   propertyNo
            pCity     cityID   |
 |  --------   -------   ------   ----------  --
             -----   ------   |
    |    NULL     NULL        NULL        PA14
            Aberdeen    4       |
     |    B003     Glasgow     3         PG4
            Glasgow     3       |
    |    B004     Bristol     1         NULL
            NULL        NULL    |
    |    B002     London      2         PL94
            London      2       |
                        |
                        |
==========================================
==========
```

В случае полного внешнего соединения в результирующую таблицу помещаются не только те две строки, которые имеют одинаковые значения в сопоставляемых столбцах с названием города, но и все остальные строки исходных таблиц, не нашедшие

себе соответствия. В этих строках все столбцы той таблицы, в которой не было найдено соответствия, заполняются значениями NULL.

Самосоединения в SQL-92.

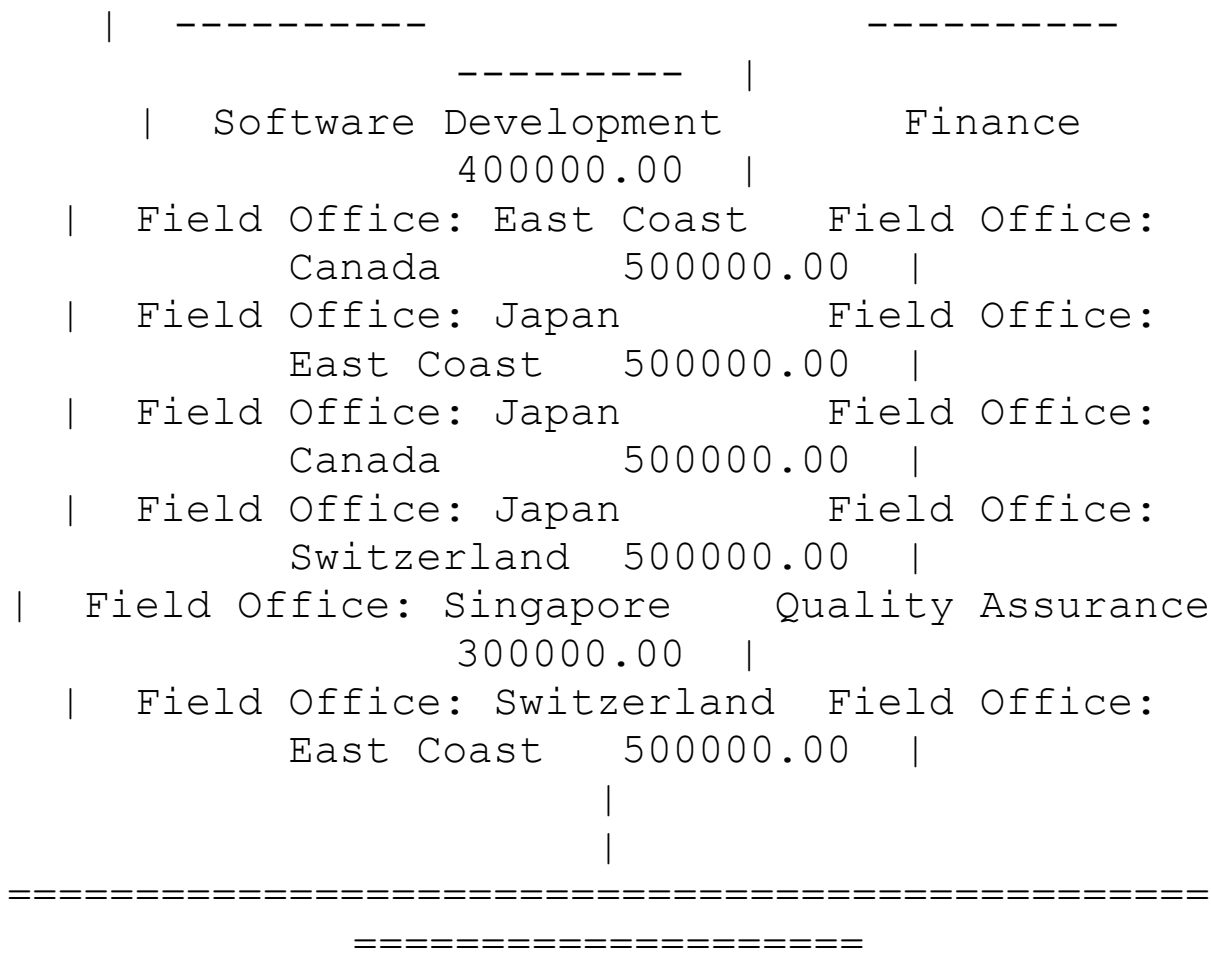
В некоторых задачах необходимо получить информацию, выбранную особым образом только из одной таблицы. Для этого используются так называемые ***сам соединения***, или ***рефлексивные соединения***. Это не отдельный вид соединения, а просто соединение таблицы с собой с помощью псевдонимов. Само соединения полезны в случаях, когда нужно получить пары аналогичных элементов из одной и той же таблицы.

Таблица 9.3. Department.

DEPT_NO	DEPARTMENT	BUDGET
1	Software Development	400000.00
2	Field Office: Canada	500000.00
3	Finance	400000.00
4	Field Office: East Coast	500000.00
5	Field Office: Japan	500000.00
6	Field Office: Singapore	300000.00
7	Field Office: Switzerland	500000.00
8	Quality Assurance	300000.00

Например, требуется получить список пар отделов с одинаковыми годовыми бюджетами:

```
SELECT d1.department, d2.department, d1.budget
FROM department d1, department d2
WHERE d1.budget = d2.budget;
    ====================  SQL Execution Log
          ========================
                        |
                        |
    |  DEPARTMENT                    DEPARTMENT
                  BUDGET      |
```



Вложенные запросы

С помощью SQL вы можете вкладывать запросы внутрь друга друга. Обычно, внутренний запрос генерирует значение, которое проверяется в предикате внешнего запроса, определяющего верно оно или нет. Вложенные запросы часто называют ***подзапросами***.

Таблица 9.4 Salespeople (Продавцы).

SNUM	SNAME	CITY	COMM
1001	Peel	London	0.12
1002	Serres	San Jose	0.13
1004	Motika	London	0.11
1007	Rifkin	Barcelona	0.15
1003	Axelrod	New York	0.10

Таблица 9.5 Customers (Покупатели).

CNUM	CNAME	CITY	RATING	SNUM
2001	Hoffman	London	100	1001
2002	Giovanni	Rome	200	1003

2003	Liu	SanJose	200	1002
2004	Grass	Berlin	300	1002
2006	Clemens	London	100	1001
2008	Cisneros	SanJose	300	1007
2007	Pereira	Rome	100	1004

Таблица 9.5 Customers (Покупатели).

ONUM	AMT	ODATE	CNUM	SNUM
3001	18.69	10/03/1990	2008	1007
3003	767.19	10/03/1990	2001	1001
3002	1900.10	10/03/1990	2007	1004
3005	5160.45	10/03/1990	2003	1002
3006	1098.16	10/03/1990	2008	1007
3009	1713.23	10/04/1990	2002	1003
3007	75.75	10/04/1990	2004	1002
3008	4723.00	10/05/1990	2006	1001
3010	1309.95	10/06/1990	2004	1002
3011	9891.88	10/06/1990	2006	1001

Например, предположим, что мы знаем имя продавца: Motika, но не знаем значение его поля snum, и хотим извлечь все заказы из таблицы заказов. Имеется один способ, чтобы сделать это:

```
SELECT * FROM Orders WHERE snum = ( SELECT snum
FROM Salespeople WHERE sname = 'Motika');
```

9.4 Ограничения, налагаемые на вложенные запросы.

Необходимо, чтобы наш подзапрос в предыдущем примере возвращал одно и только одно значение. Имея выбранным поле snum «WHERE city = ‘London’» вместо «WHERE sname = ‘Motika’», можно получить несколько различных значений. Это может сделать уравнение в предикате основного запроса невозможным для оценки верности или неверности, и команда выдаст ошибку. При использовании подзапросов в предикатах, основанных на реляционных операторах, вы должны убедиться, что использовали подзапрос, который будет выдавать одну и только одну строку вывода. Если вы используете подзапрос, который не выводит никаких значений вообще, команда не потерпит неудачи; но основной запрос не выведет никаких

значений. Подзапросы, которые не производят никакого вывода (или нулевой вывод), вынуждают рассматривать предикат ни как верный, ни как неверный, а как неизвестный. Однако, неизвестный предикат имеет тот же самый эффект что и неверный: никакие строки не выбираются основным запросом.

Один тип функций, который автоматически может производить одиночное значение для любого числа строк, конечно же, – агрегатная функция.

Любой запрос, использующий одиночную функцию агрегата без предложения GROUP BY, будет выбирать одиночное значение для использования в основном предикате. Например, вы хотите увидеть все заказы имеющие сумму приобретений выше средней на 4-е Октября:

```
SELECT * FROM Orders WHERE amt > ( SELECT AVG
(amt) FROM Orders WHERE odate = 10/04/1990 );
```

Вы можете использовать подзапросы, которые производят любое число строк если вы используете специальный оператор IN. Как вы помните, IN определяет набор значений, одно из которых должно совпадать с другим термином уравнения предиката в порядке, чтобы предикат был верным. Когда вы используете IN с подзапросом, SQL просто формирует этот набор из вывода подзапроса. Мы можем, следовательно, использовать IN чтобы выполнить такой же подзапрос, который не будет работать с реляционным оператором, и найти все атрибуты таблицы заказов для продавца в Лондоне:

```
SELECT * FROM Orders WHERE snum IN ( SELECT
snum FROM Salespeople WHERE city = 'LONDON' );
```

Операторы BETWEEN, LIKE, и IS NULL не могут использоваться с подзапросами, но можно использовать операторы ALL и ANY

Например: Выбрать заказчиков, которые имеют больший рейтинг чем любой заказчик в Риме.

```
SELECT * FROM Customers WHERE rating > ANY (
SELECT rating FROM Customers WHERE city =
'Rome' );
```

С помощью ALL, предикат является верным, если каждое значение выбранное подзапросом удовлетворяет условию в

предикате внешнего запроса. Если мы хотим пересмотреть наш предыдущий пример чтобы вывести только тех заказчиков, чьи оценки, фактически, выше, чем у каждого заказчика в Лондоне:

```
SELECT * FROM Customers WHERE rating > ALL
(SELECT rating FROM Customers WHERE city =
'Rome');
```

Смысл всех подзапросов тот, что все они выбирают одиночный столбец. Это обязательно, поскольку выбранный вывод сравнивается с одиночным значением. Подтверждением этому то, что SELECT * не может использоваться в подзапросе. Имеется исключение из этого, когда подзапросы используются с оператором EXISTS.

Соотнесенные вложенные запросы.

Когда вы используете подзапросы в SQL, вы можете обратиться к внутреннему запросу таблицы в предложении внешнего запроса FROM, сформировав – ***соотнесенный подзапрос***. Когда вы делаете это, подзапрос выполняется неоднократно, по одному разу для каждой строки таблицы основного запроса. Соотнесенный подзапрос - один из большого количества тонких понятий в SQL из-за сложности в его оценке. Если вы сумеете овладеть им, вы найдете что он очень мощный, потому что может выполнять сложные функции с помощью очень лаконичных указаний.

Например, имеется один способ найти всех заказчиков в заказах на 3-е Октября:

```
SELECT * FROM Customers outer WHERE 10/03/1990
IN ( SELECT odate
FROM Orders inner WHERE outer.cnum = inner.cnum
);
```

EXISTS – это оператор, который производит верное или неверное значение, другими словами, выражение Буля. Это означает, что он может работать автономно в предикате или в комбинации с другими выражениями Буля использующими Булевы операторы AND, OR, и NOT. Он берет подзапрос как аргумент и оценивает его как верный, если тот производит любой вывод или как неверный, если тот не делает этого. Этим он отличается от других операторов предиката, в которых он не

может быть неизвестным. Например, мы можем решить, извлекать ли нам некоторые данные из таблицы заказчиков если, и только если, один или более заказчиков в этой таблице находятся в San Jose:

```
SELECT cnum, cname, city FROM Customers WHERE
EXISTS (SELECT *
FROM Customers WHERE city = 'San Jose');
```

Или мы можем вывести продавцов, которые имеют многочисленых заказчиков:

```
SELECT DISTINCT snum FROM Customers outer WHERE
EXISTS (SELECT * FROM Customers inner WHERE
inner.snum = outer.snum AND inner.cnum < >
outer.cnum );
```

Однако для нас может быть полезнее вывести больше информации об этих продавцах, а не только их номера. Мы можем сделать это, объединив таблицу Заказчиков с таблицей Продавцов:

```
SELECT DISTINCT first.snum, sname, first.city
FROM Salespeople first, Customers second WHERE
EXISTS ( SELECT * FROM Customers third WHERE
second.snum = third.snum AND second.cnum < >
third.cnum ) AND first.snum = second.snum;
```

Контрольные вопросы:

1. Как производится выборка данных?
2. Как производится обновление данных?
3. Как производится удаление данных?
4. Как добавляются строки в таблицу?
5. Что такое агрегатная функция?
6. Перечислите основные агрегатные функции.
7. Где используется HAVING?
8. Какие виды соединений бывают?
9. Что такое самосоединение?
10. Объясните пошагово как производится соединение?
11. Как выполняется подзапрос?
12. Что такое соотнесенный подзапрос?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p.
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p.

Глава X. SQL: определение данных

10.1. Язык определения данных

В состав языка определения входят операторы:

1. CREATE – создает объектов базы данных
2. ALTER – изменяет объект
3. DROP –удаляет объект
4. Стандарт ANSI SQL-92 определяет команды для следующих объектов:
5. ASSERTION – утверждения для проверки
6. CHARACTER SET – набор символов
7. COLLATION – правила сортировки для набора символов
8. DOMAIN – домен (пользовательского типа данных столбца).
9. SCHEMA – схема (именованной группы объектов)
10. TABLE – таблица базы данных
11. TRANSLATION – правила преобразования (трансляции) из одного набора символов в другой (используется в операторе TRANSLATE)
12. VIEW – представления данных

Типы данных.

ANSI SQL включает в себя следующие типы данных

Символьные строки:

- `CHARACTER(`*n*`)` или `CHAR(`*n*`)` — строка фиксированной длины в *n* символов, разделенная пробелами;
- `CHARACTER VARYING(`*n*`)` или `VARCHAR(`*n*`)` — строка переменной длины с максимальным количеством символов *n*;
- `NATIONAL CHARACTER(`*n*`)` или `NCHAR(`*n*`)` — строка фиксированной длины с поддержкой международных кодировок

- NATIONAL CHARACTER VARYING(*n*) или NVARCHAR(*n*) — строка переменной длины NCHAR.

Битовые данные:

- BIT(*n*) — массив из *n* битов
- BIT VARYING(*n*) — массив длинной до *n* битов

Числа:

- INTEGER и SMALLINT — целые числа;
- FLOAT, REAL и DOUBLE PRECISION — вещественные числа;
- NUMERIC(*precision, scale*)или DECIMAL(*precision, scale*)— вещественное число с указанием в скобках количество знаков до запятой и после запятой.

Дата и время:

- DATE — дата (2010-05-30);
- TIME — время (14:55:37);
- TIME WITH TIME ZONE или TIMESTAMP — тоже самое, что и TIME, только исключаются данные о часовом поясе;
- TIMESTAMP — это DATE и TIME соединенные вместе в одной переменной (2010-05-30 14:55:37).
- TIMESTAMP WITH TIME ZONE or TIMESTAMPTZ — тоже самое, что и TIMESTAMP, только исключаются данные о часовом поясе.

Домен.

Создание домена:

```
CREATE DOMAIN <имя_домена> [AS] <тип_данных>
[DEFAULT {LITERAL | NULL | USER}]
  [NOT NULL] [CHECK (<условие>)]
  [COLLATE <имя_сортировки>];
```

где

- ***DEFAULT*** – Определяет значение по умолчанию, которое вставляется, когда ни какой другой ввод не сделан. Значения:
  - ***LITERAL*** – Вводится определенная строка, числовое значение или дата.

- ***NULL*** – Вводится значение NULL.
- ***USER*** – Вводится имя текущего пользователя. Столбец должен быть совместимый символьный тип, чтобы использовать значение по умолчанию.

- ***NOT NULL*** – Определяет, что значения, введенные в столбец, не могут быть NULL.
- ***CHECK*** – (<условие>) создает одиночное CHECK ограничение для домена.
- ***VALUE*** – Заменитель для имени столбца, в конечном счете, основанном на домене.
- ***COLLATE <имя_сортировки>*** – Устанавливает способ сортировки для домена.

Следующая инструкция создает домен, который может принимать положительные значения больше 1000, со значением по умолчанию 9999. Ключевое слово VALUE заменяется именем столбца основанном на этом домене.

```
CREATE DOMAIN CUSTNO
  AS INTEGER
    DEFAULT 9999
    CHECK (VALUE > 1000);
```

Следующая инструкция ограничивает введенные значения в домен до четырех определенных значений:

```
CREATE DOMAIN PRODTYPE
  AS VARCHAR(12)
    CHECK (VALUE IN ("software", "hardware",
"other", "N/A"));
```

Изменение домена:

```
ALTER DOMAIN <имя_домена> {
[SET DEFAULT {LITERAL | NULL | USER}]
  | [DROP DEFAULT]
  | [ADD [CONSTRAINT] CHECK (<условие>)]
  | [DROP CONSTRAINT]
  };
```

где

- ***SET DEFAULT*** – Определяет значение столбца по умолчанию, которое будет введено, когда ни какой другой ввод не сделан. Значения:
  - ***LITERAL –*** Вводится определенная строка, числовое значение или дата.
  - ***NULL –*** Вводится значение NULL.
  - ***USER –*** Вводится имя текущего пользователя. Столбец должен быть совместимого текстового типа для использования значения по умолчанию.

Установка значения по умолчанию на уровне столбца отменяет установку значения по умолчанию на уровне домена.

- ***DROP DEFAULT*** – Удаляет существующее значение по умолчанию.
- ***ADD [CONSTRAINT] CHECK (<условие>)*** – Добавляет CHECK ограничения в определение домена. Определение домена может включать только одно CHECK ограничение.
- ***DROP CONSTRAINT*** – Удаляет CHECK ограничения из определения домена.

Следующая инструкция устанавливает значение домена по умолчанию к 9999.

```
ALTER DOMAIN CUSTNO SET DEFAULT 9999;
```

Удаление домена:

```
DROP DOMAIN <имя_домена>;
```

Схема

Создание базы данных:

```
CREATE {DATABASE | SCHEMA} <имя_базы_данных>
```

Удаление базы данных:

```
DROP {DATABASE | SCHEMA} <имя_базы_данных>
```

Таблица.

Создание таблицы

```
CREATE TABLE [ IF NOT EXISTS ] <имя_таблицы>
(
<имя столбца_1> <тип данных> [ DEFAULT
expression ] [ {NULL | NOT NULL} ] [
{INDEX_BLIST | INDEX_NONE} ]
```

```
<имя столбца_2> <тип данных> [ DEFAULT
expression ] [ {NULL | NOT NULL} ] [
{INDEX_BLIST | INDEX_NONE} ]
...
<имя столбца_N> <тип данных> [ DEFAULT
expression ] [ {NULL | NOT NULL} ] [
{INDEX_BLIST | INDEX_NONE} ]
[ CONSTRAINT <имя ограничения> ]
PRIMARY KEY ( <имя столбца_1>, <имя столбца_2>,
... ) |
FOREIGN KEY (<имя столбца_1>, <имя столбца_2>,
... ) REFERENCES <имя_таблицы_2> [ (<имя
столбца_1>, <имя столбца_2>, ... ) ] [ ON
UPDATE {NO ACTION | SET NULL | SET DEFAULT |
CASCADE} ] [ ON DELETE {NO ACTION | SET NULL |
SET DEFAULT | CASCADE} ] |

UNIQUE (<имя столбца_1>, <имя столбца_2>, ... )
|

CHECK ( <условие> ) [ {INITIALLY DEFERRED |
INITIALLY IMMEDIATE} ] [ {NOT DEFERRABLE |
DEFERRABLE} ]
);
```

где

- ***DEFAULT*** expression – значение по умолчанию;
- ***NULL*** | ***NOT NULL*** – разрешается ли пустое поле;
- ***INDEX_BLIST*** | ***INDEX_NONE*** – есть или нет индекса;
- **CONSTRAINT** – ограничение
  - ***PRIMARY KEY –*** первичный ключ
  - ***FOREIGN KEY –*** вторичный ключ
    - ON DELETE – при удалении в родительской таблице
    - ON UPDATE – при обновлении в родительской таблице
    - NO ACTION – нет действий
    - SET NULL – устанавливается значение NULL
    - SET DEFAULT – устанавливается значение по умолчанию

        - CASCADE – каскадно
    - ***UNIQUE*** – уникальный
    - ***CHECK*** – проверка

Например

```
CREATE TABLE Customer (
  number VARCHAR(40)  NOT NULL,
  name   VARCHAR(100) NOT NULL,
  ssn    VARCHAR(50)  NOT NULL,
  age    INTEGER      NOT NULL,

  CONSTRAINT cust_pk PRIMARY KEY (number),
  UNIQUE ( ssn ),              // (An anonymous
constraint)
  CONSTRAINT age_check CHECK (age >= 0 AND age
< 200)
);
```

Изменение таблицы:

Переименование таблицы

```
ALTER TABLE <имя_таблицы> RENAME TO
<новое_имя_таблицы>
```

Переименование столбца

```
ALTER TABLE <имя_таблицы> RENAME [ COLUMN ]
<имя_столбца> TO <новое_имя_столбца>
```

Добавление столбца

```
ALTER TABLE <имя_таблицы> ADD [COLUMN]
<имя_столбца> <тип_данных> [ DEFAULT expression
] [ {NULL | NOT NULL} ] [ {INDEX_BLIST |
INDEX_NONE} ]
```

Добавление первичного ключа ограничения к таблице

```
ALTER TABLE <имя_таблицы> ADD [ CONSTRAINT <имя
ограничения> ]
PRIMARY KEY ( <имя столбца_1>, <имя столбца_2>,
... ) |
```

Добавление вторичного ключа ограничения к таблице

```
ALTER TABLE <имя_таблицы> ADD [ CONSTRAINT <имя
ограничения> ]
FOREIGN KEY (<имя столбца_1>, <имя столбца_2>,
... ) REFERENCES <имя_таблицы_2> [ (<имя
столбца_1>, <имя столбца_2>, ... ) ] [ ON
UPDATE {NO ACTION | SET NULL | SET DEFAULT |
CASCADE} ] [ ON DELETE {NO ACTION | SET NULL |
SET DEFAULT | CASCADE} ] |
```

Добавление уникального поля к таблице

```
ALTER TABLE <имя_таблицы> ADD [ CONSTRAINT <имя
ограничения> ]
UNIQUE (<имя столбца_1>, <имя столбца_2>, ... )
|
```

Добавление проверки столбца к таблице

```
ALTER TABLE <имя_таблицы> ADD [ CONSTRAINT <имя
ограничения> ]
CHECK ( <условие> ) [ {INITIALLY DEFERRED |
INITIALLY IMMEDIATE} ] [ {NOT DEFERRABLE |
DEFERRABLE} ]
```

Изменение типа данных столбца

```
ALTER TABLE <имя_таблицы> MODIFY "column 1"
"New Data Type"
```

Изменение столбца ограничений столбца

```
ALTER TABLE <имя_таблицы> ALTER [COLUMN]
column_name SET default_expr
ALTER TABLE <имя_таблицы> ALTER [COLUMN]
column_name DROP DEFAULT
```

Изменение удаление столбца

```
ALTER TABLE <имя_таблицы> DROP [COLUMN]
column_name
```

Удаление ограничения таблицы

```
ALTER TABLE <имя_таблицы> DROP CONSTRAINT
constraint_name
```

Удаление первичного ключа

```
ALTER TABLE <имя_таблицы> DROP PRIMARY KEY
```

Удаление таблицы

```
DROP TABLE [ IF EXISTS ] <имя_таблицы>
```

10.2. Представления

Представление (VIEW) - объект данных который не содержит никаких данных его владельца. Это - тип таблицы, чье содержание выбирается из других таблиц с помощью выполнения запроса. Поскольку значения в этих таблицах меняются, то автоматически, их значения могут быть показаны представлением. В этой главе, вы узнаете, что такое представления, как они создаются, и немного об их возможностях и ограничениях. Использование представлений, основанных на улучшенных средствах запросов, таких как объединение и подзапрос, разработанных очень тщательно, в некоторых случаях даст больший выигрыш по сравнению с запросами.

Представления - это таблицы чье содержание выбирается или получается из других таблиц. Они работают в запросах и операторах DML точно также, как и основные таблицы, но не содержат никаких собственных данных. Представления - подобны окнам, через которые вы просматриваете информацию (как она есть, или в другой форме, как вы потом увидите), которая фактически хранится в базовой таблице. Представление - это фактически запрос, который выполняется всякий раз, когда представление становится темой команды. Вывод запроса при этом в каждый момент становится содержанием представления.

Вы создаете представление командой CREATE VIEW. Она состоит из слов CREATE VIEW (СОЗДАТЬ ПРЕДСТАВЛЕНИЕ), имени представления которое нужно создать, слова AS (КАК), и далее запроса, как в следующем примере:

```
CREATE VIEW Londonstaff
    AS SELECT *
    FROM Salespeople
    WHERE city = 'London';
```

Теперь Вы имеете представление, называемое Londonstaff. Вы можете использовать это представление точно так же, как и любую другую таблицу. Она может быть запрошена,

модифицирована, вставлена в, удалена из, и соединена с, другими таблицами и представлениями. Давайте сделаем запрос такого представления (вывод показан в Рисунке 10.1):

```
SELECT *
    FROM Londonstaff;
```

```
==============  SQL Execution Log ============
                          |
                          |
                    | SELECT *
                          |
                | FROM  Londonstaff;
                          |
                          |
                          |
                          |
=============================================|
 |    snum       sname          city          comm
                          |
 | ------    ----------   -----------   -------
                          |
 |   1001       Peel          London       0.1200
                          |
 |   1004       Motika        London       0.1100
                          |
                          |
                          |
=============================================
```

Рис. 10.1. Представление Londonstaff .

Когда вы приказываете SQL выбрать (SELECT) все строки (*) из представления, он выполняет запрос содержащий в определении - Londonstaff, и возвращает все из его вывода. Имея предикат в запросе представления, можно вывести только те строки из представления, которые будут удовлетворять этому предикату. Если это так, вы будете должны выбрать другое им для вашего представления). Преимущество использования представления, по сравнению с основной таблицы, в том, что представление будет модифицировано автоматически всякий раз, когда таблица, лежащая в его основе, изменяется. Содержание

представления не фиксировано, и переназначается каждый раз, когда вы ссылаетесь на представление в команде. Если вы добавите завтра другого, живущего в Лондоне продавца, он автоматически появится в представлении.

Представления значительно расширяют управление вашими данными. Это - превосходный способ дать публичный доступ к некоторой, но не всей информации в таблице. Если вы хотите чтобы ваш продавец был показан в таблице Продавцов, но при этом не были показаны комиссии других продавцов, вы могли бы создать представление с использованием следующего оператора (вывод показан в Рис 10.2)

```
CREATE VIEW Salesown
    AS SELECT snum, sname, city
    FROM Salespeople:
```

```
    ============== SQL Execution Log ============
                          |
                          |
                    | SELECT *
                          |
                | FROM  Salesown;
                          |
                          |
                          |
                          |
 ==========================================|
         |   snum      sname          city
                          |
       | ------    ----------   -----------
                          |
        |   1001      Peel          London
                          |
       |   1002      Serres        San Jose
                          |
        |   1004      Motika        London
                          |
       |   1007      Rifkin        Barcelona
                          |
```

```
|   1003      Axelrod      New York
                                |
==========================================
```

Рис 10.2. Представление Salesown

Другими словами, это представление - такое же как для таблицы Продавцов, за исключением того, что поле comm, не упоминалось в запросе, и, следовательно, не было включено в представление.

Модифицирование представлений

Представление может теперь изменяться командами модификации DML, но модификация не будет воздействовать на само представление. Команды будут на самом деле перенаправлены к базовой таблице:

```
UPDATE Salesown
    SET city = 'Palo Alto'
    WHERE snum = 1004;
```

Его действие идентично выполнению той же команды в таблице Продавцов. Однако, если значение комиссионных продавца будет обработано командой UPDATE

```
UPDATE Salesown
    SET comm = .20
    WHERE snum = 1004;
```

она будет отвергнута, так как поле comm отсутствует в представлении Salesown. Это важное замечание, показывающее, что не все представления могут быть модифицированы.

Именование столбцов.

В нашем примере, пол наших представлений имеют свои имена, полученные прямо из имен полей основной таблицы. Это удобно. Однако, иногда вам нужно снабжать ваши столбцы новыми именами:

- когда некоторые столбцы являются выводимыми, и поэтому не имеющими имен.
- когда два или более столбцов в объединении, имеют те же имена что в их базовой таблице.

Имена, которые могут стать именами полей, даются в круглых скобках ( ), после имени таблиц. Они не будут

запрошены, если совпадают с именами полей запрашиваемой таблицы. Тип данных и размер этих полей будут отличаться от запрашиваемых полей, которые "передаются" в них. Обычно вы не указываете новых имен полей, но если вы все-таки сделали это, вы должны делать это для каждого пол в представлении.

10.3 Комбинирование предикатов представлений и основных запросов в представлениях

Когда вы делаете запрос представления, вы собственно, запрашиваете запрос. Основной способ для SQL обойти это, - объединить предикаты двух запросов в один. Давайте посмотрим еще раз на наше представление с име- нем Londonstaff :

```
CREATE VIEW Londonstaff
   AS SELECT *
   FROM Salespeople
   WHERE city = 'London';
```

Если мы выполняем следующий запрос в этом представлении

```
SELECT *
   FROM Londonstaff
   WHERE comm > .12;
```

он такой же, как если бы мы выполнили следующее в таблице Продавцов:

```
SELECT *
   FROM Salespeople
   WHERE city = 'London'
   AND comm > .12;
```

Это прекрасно, за исключением того, что появляется возможна проблема с представлением. Имеется возможность комбинации из двух полностью допустимых предикатов и получения предиката который не будет работать. Например, предположим, что мы создаем (CREATE) следующее представление:

```
CREATE VIEW Ratingcount (rating, number)
   AS SELECT rating, COUNT (*)
   FROM Customers
   GROUP BY rating;
```

Это дает нам число заказчиков, которые мы имеем для каждого уровня оценки(rating). Вы можете затем сделать запрос этого представления чтобы выяснить, имеется ли какая-нибудь оценка, в настоящее время назначенная для трех заказчиков:

```
SELECT *
    FROM Ratingcount
    WHERE number = 3;
```

Посмотрим, что случится если мы скомбинируем два предиката:

```
 SELECT rating, COUNT (*)
    FROM Customers
    WHERE COUNT (*) = 3
    GROUP BY rating;
```

Это недопустимый запрос. Агрегатные функции, такие как COUNT (СЧЕТ), не могут использоваться в предикате. Правильным способом при формировании вышеупомянутого запроса, конечно же будет следующий:

```
SELECT rating, COUNT (*)
    FROM Customers
    GROUP BY rating;
    HAVING COUNT (*) = 3;
```

Но SQL может не выполнить превращения. Может ли равноценный запрос вместо запроса Ratingcount потерпеть неудачу? Да может! Это - неоднозначна область SQL, где методика использования представлений может дать хорошие результаты. Самое лучшее что можно сделать в случае, когда об этом ничего не сказано в вашей системной документации, так это попытка в ней разобраться. Если команда допустима, вы можете использовать представления чтобы установить некоторые ограничения SQL в синтаксисе запроса.

Групповые представления.

Групповые представления - это представления, наподобие запроса Ratingcount в предыдущем примере, который содержит предложение GROUP BY, или который основывается на других групповых представлениях. Групповые представления могут стать превосходным способом обрабатывать полученную информацию непрерывно. Предположим, что каждый день вы должны следить

за порядком номеров заказчиков, номерами продавцов принимающих порядки, номерами порядков, средним от порядков, и общей суммой приобретений в порядках.

Чем конструировать каждый раз сложный запрос, вы можете просто создать следующее представление:

```
CREATE VIEW Totalforday
    AS SELECT odate, COUNT (DISTINCT cnum),
COUNT
        (DISTINCT snum), COUNT (onum), AVG
        (amt), SUM (amt)
        FROM Orders
        GROUP BY odate;
```

Теперь вы сможете увидеть всю эту информацию с помощью простого запроса:

```
SELECT *
FROM Totalforday;
```

Как мы видели, SQL запросы могут дать вам полный комплекс возможностей, так что представления обеспечивают вас чрезвычайно гибким и мощным инструментом чтобы определить точно, как ваши данные могут быть использованы. Они могут также делать вашу работу более простой, переформатируя данные удобным для вас способом и исключив двойную работу.

Представления и объединения.

Представления не требуют, чтобы их вывод осуществлялся из одной базовой таблицы. Так как почти любой допустимый запрос SQL может быть использован в представлении, он может выводить информацию из любого числа базовых таблиц, или из других представлений. Мы можем, например, создать представление, которое показывало бы, порядки продавца и заказчика по имени:

```
CREATE VIEW Nameorders
    AS SELECT onum, amt, a.snum, sname, cname
    FROM Orders a, Customers b, Salespeople c
    WHERE a.cnum = b.cnum
    AND a.snum = c.snum;
```

Теперь вы можете выбрать (SELECT) все порядки заказчика или продавца (*), или можете увидеть эту информацию для

любого порядка. Например, чтобы увидеть все порядки продавца Rifkin, вы должны ввести следующий запрос (вывод показан в рис. 10.3):

```
SELECT *
FROM Nameorders
WHERE sname = 'Rifkin';
```

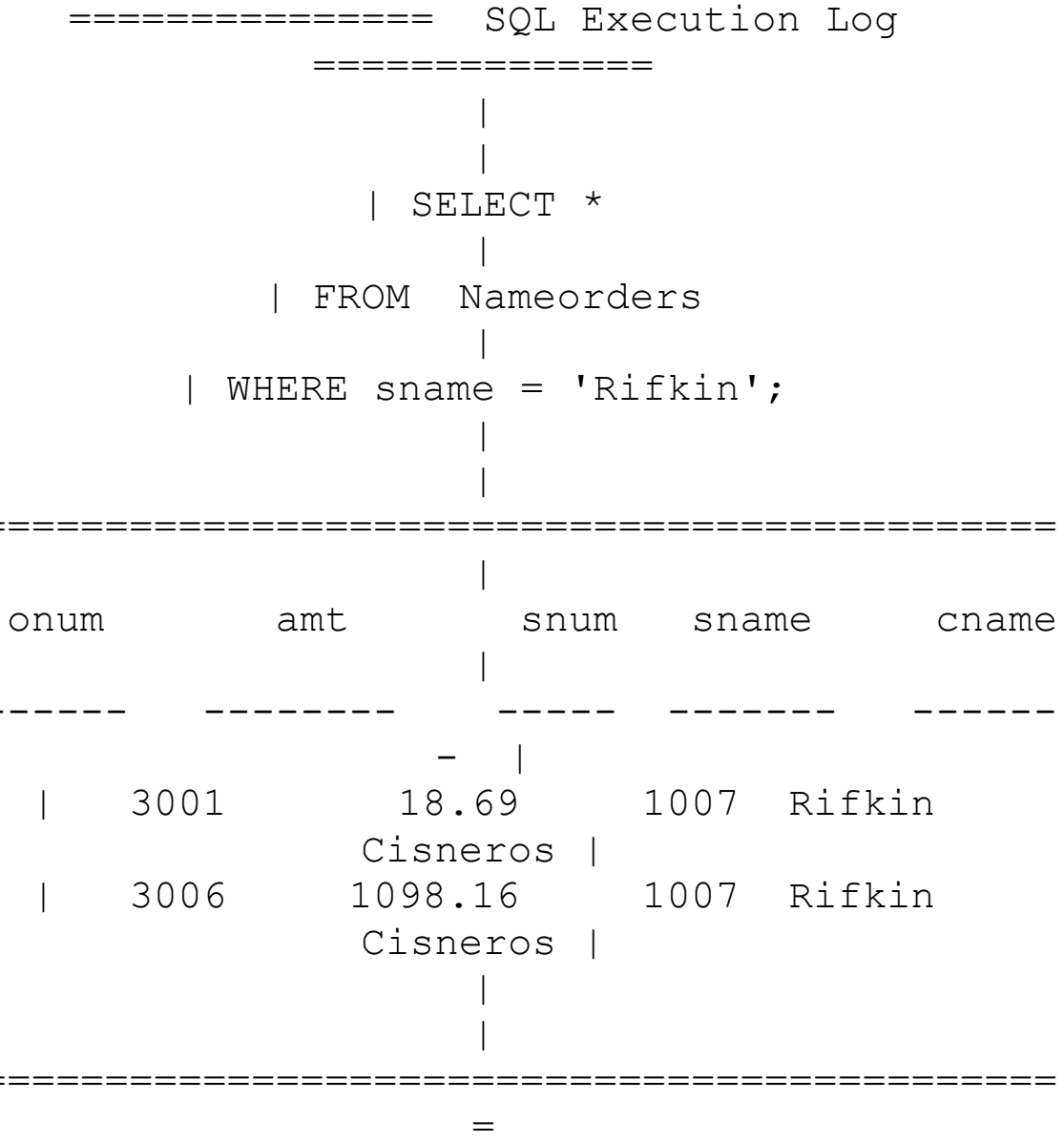


Рис. 10.3. Порядки Rifkin показанные в Nameorders

Вы можете также объединять представления с другими таблицами, или базовыми таблицами или представлениями, поэтому вы можете увидеть все порядки Axelroda и значения его комиссионных в каждом порядке:

```
SELECT a.sname, cname, amt  comm
FROM Nameorders a, Salespeople b
```

```
WHERE a.sname = 'Axelrod'
AND b.snum = a.snum;
```

Вывод для этого запроса показывается в рис.10.4.

В предикате, мы могли бы написать - " WHERE a.sname = 'Axelrod' AND b.sname = 'Axelrod' " , но предикат который мы использовали здесь более общеупотребительный. Кроме того, поле snum - это первичный ключ таблицы Продавцов, и, следовательно, должен по определению быть уникальным.

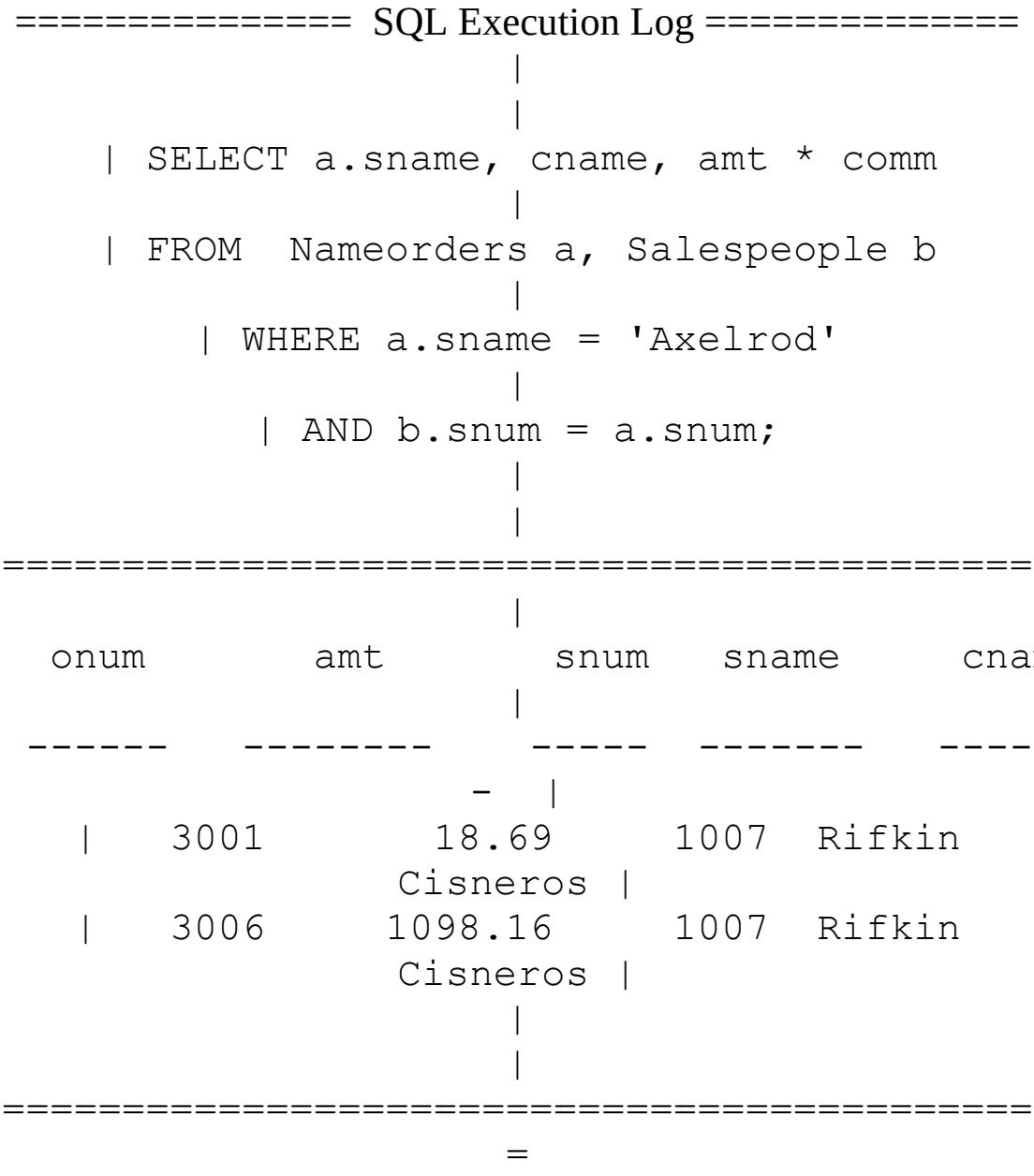


Рис. 10.4. Объединение основной таблицы с представлением

Если бы там, например было два Axelrodf, вариант с именем, будет объединять вместе их данные. Более предпочтительный вариант - использовать поле snum чтобы хранить его отдельно.

Представления и подзапросы.

Представления могут также использовать и подзапросы, включая соотнесенные подзапросы. Предположим ваша компания предусматривает премию для тех продавцов которые имеют заказчика с самым высоким порядком для любой указанной даты. Вы можете проследить эту информацию с помощью представления:

```
CREATE VIEW Elitesalesforce
    AS SELECT b.odate, a.snum, a.sname,
    FROM Salespeople a, Orders b
    WHERE a.snum = b.snum
    AND b.amt =
        (SELECT MAX (amt)
        FROM Orders c
        WHERE c.odate = b.odate);
```

Если, с другой стороны, премия будет назначаться только продавцу, который имел самый высокий порядок за последние десять лет, вам необходимо будет проследить их в другом представлении основанном на первом:

```
CREATE VIEW Bonus
    AS SELECT DISTINCT snum, sname
    FROM Elitesalesforce a
    WHERE 10 < =
        (SELECT COUNT (*)
        FROM Elitesalestorce b
        WHERE a.snum = b.snum);
```

Извлечение из этой таблицы продавца, который будет получать премию - выполняется простым вопросом:

```
SELECT *
FROM Bonus;
```

Теперь мы видим истинную мощность SQL. Извлечение той же полученной информации программами RPG или COBOL будет более длительной процедурой. В SQL, это - только вопрос из двух комплексных команд, сохраненных, как представление совместно с простым запросом. При самостоятельном запросе - мы должны заботится об этом каждый день, потому что информация, которую

извлекает запрос, непрерывно меняется чтобы отражать текущее состояние базы данных.

Что не могут делать представления.

Имеются большое количество типов представлений (включая многие из наших примеров в этой главе) которые являются доступными только для чтения. Это означает, что их можно запрашивать, но они не могут подвергаться действиям команд модификации. Имеются также некоторые виды запросов, которые не допустимы в определениях представлений. Одиночное представление должно основываться на одиночном запросе; ОБЪЕДИНЕНИЕ (UNION) и ОБЪЕДИНЕНИЕ ВСЕГО (UNIOM ALL) не разрешаются. УПОРЯДОЧЕНИЕ ПО (ORDER BY) никогда не используется в определении представлений. Вывод запроса формирует содержание представления, которое напоминает базовую таблицу и является - по определению - неупорядоченным.

Удаление представлений.

Синтаксис удаления представления из базы данных подобен синтаксису удаления базовых таблиц:

```
DROP VIEW < view name >
```

В этом нет необходимости, однако, сначала надо удалить все содержание как это делается с базовой таблицей, потому что содержание представления не является созданным и сохраняется в течении определенной команды. Базовая таблица, из которой представление выводится, не эффективна, когда представление удалено. Помните, вы должны являться владельцем представления чтобы иметь возможность удалить его.

Контрольные вопросы:

1. Что такое язык определения данных?
2. Какие объекты можно создавать с помощью языка определения данных?
3. Как производить модификацию таблиц с помощью DDL?
4. Что такое представление?
5. Как формируется представление?
6. Можно ли обновлять данные представления и почему?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p.
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p.

Глава - XI. Управление транзакциями. Обработка и оптимизация запросов

11.1. Цель управления согласованностью

Согласованность данных (иногда консистентность данных, англ. data consistency) — согласованность данных друг с другом, целостность данных, а также внутренняя непротиворечивость.

Задача аналитика и проектировщика базы данных — возможно более полно выявить все имеющиеся ограничения целостности и задать их в базе данных.

Целостность БД не гарантирует достоверности содержащейся в ней информации, но обеспечивает по крайней мере правдоподобность этой информации, отвергая заведомо невероятные, невозможные значения. Таким образом, не следует путать целостность БД с достоверностью БД. Достоверность (или истинность) есть соответствие фактов, хранящихся в базе данных, реальному миру. Очевидно, что для определения достоверности БД требуется обладание полными знаниями как о содержимом БД, так и о реальном мире. Для определения целостности БД требуется лишь обладание знаниями о содержимом БД и о заданных для неё правилах. Поэтому СУБД может (и должна) контролировать целостность БД, но принципиально не в состоянии контролировать достоверность БД. Контроль достоверности БД может быть возложен только на человека, да и то в ограниченных масштабах, поскольку в ряде случаев люди тоже не обладают полнотой знаний о реальном мире.

Итак, БД может быть целостной, но не достоверной. Возможно и обратное: БД может быть достоверной, но не целостной. Последнее имеет место, если правила (ограничения целостности) заданы неверно.

Транзакция – это логическая единица работы. Например. Предположим сначала, что отношение Students (отношение студентов) включает дополнительный атрибут AvgMark, представляющий собой средний балл студента, по результатам сдачи текущей сессии. Значение AvgMark для любой определенной детали предполагается равным среднему арифметическому всех значений Mark из таблицы Marks для всех оценок полученных в текущем семестре.

В приведенном примере предполагается, что речь идет об одиночной, атомарной операции. На самом деле добавление новой оценки в таблицу Marks – это выполнение двух обновлений в базе данных (под обновлениями здесь, конечно, понимаются операции insert, delete, а также сами по себе операции update). Более того, в базе данных между этими двумя обновлениями временно нарушается требование, что значение AvgMark для студента 1 равно среднему арифметическому всех значений поля Mark для студента 1 в текущем семестре. Таким образом, логическая единица работы (т.е. транзакция) – не просто одиночная операция системы баз данных, а скорее согласование нескольких таких операций. В общем, это преобразование одного согласованного состояния базы данных в другое, причем в промежуточных точках база данных находится в несогласованном состоянии.

Из этого следует, что недопустимо, чтобы одно из обновлений было выполнено, а другое нет, так как база данных останется в несогласованном состоянии. В идеальном случае должны быть выполнены оба обновления. Однако нельзя обеспечить стопроцентную гарантию, что так и будет. Не исключена вероятность того, что, система, например, будет разрушена между двумя обновлениями, или же на втором обновлении произойдет арифметическое переполнение и т.п. Система, поддерживающая транзакции, гарантирует, что если во время выполнения неких обновлений произошла ошибка (по любой причине), то все эти обновления будут аннулированы. Таким образом, транзакция или выполняется полностью, или полностью отменяется (как будто она вообще не выполнялась).

Системный компонент, обеспечивающий атомарность (или ее подобие), называется администратором транзакций (или диспетчером транзакций), а ключами к его выполнению служат

операторы COMMIT TRANSACTION и ROLLBACK TRANSACTION.

Оператор COMMIT TRANSACTION (для краткости commit) сигнализирует об успешном окончании транзакции. Он сообщает администратору транзакций, что логическая единица работы завершена успешно, база данных вновь находится (или будет находиться) в согласованном состоянии, а все обновления, выполненные логической единицей работы, теперь могут быть зафиксированы, т.е. стать постоянными.

Оператор ROLLBACK TRANSACTION (для краткости ROLLBACK) сигнализирует о неудачном окончании транзакции. Он сообщает администратору транзакций, что произошла какая-то ошибка, база данных находится в несогласованном состоянии и все обновления могут быть отменены, т.е. аннулированы.

Для отмены обновлений система поддерживает файл регистрации, или журнал, на диске, где записываются детали всех операций обновления, в частности новое и старое значения модифицированного объекта. Таким образом, при необходимости отмены некоторого обновления система может использовать соответствующий файл регистрации для возвращения объекта в первоначальное состояние.

Еще один важный момент. Система должна гарантировать, что индивидуальные операторы сами по себе атомарные (т.е. выполняются полностью или не выполняются совсем). Это особенно важно для реляционных систем, в которых операторы многоуровневые и обычно оперируют множеством кортежей одновременно; такой оператор просто не может быть нарушен посреди операции и привести систему в несогласованное состояние. Другими словами, если произошла ошибка во время работы такого оператора, база данных должна остаться полностью неизмененной. Более того, это должно быть справедливо даже в том случае, когда действия оператора являются причиной дополнительной, например каскадной, операции.

11.2 Свойства АСИД

Из предыдущих разделов следует, что транзакции обладают четырьмя важными свойствами: атомарность, согласованность, изоляция и долговечность (назовем это свойствами АСИД).

1. ***Атомарность.*** Транзакции атомарны (выполняется все или ничего).
2. ***Согласованность.*** Транзакции защищают базу данных согласованно. Это означает, что транзакции переводят одно согласованное состояние базы данных в другое без обязательной поддержки согласованности во всех промежуточных точках.
3. ***Изоляция.*** Транзакции отделены одна от другой. Это означает, что, если даже будет запущено множество конкурирующих друг с другом транзакций, любое обновление определенной транзакции будет скрыто от остальных до тех пор, пока эта транзакция выполняется. Другими словами, для любых двух отдаленных транзакций T1 и T2 справедливо следующее утверждение: T1 сможет увидеть обновление T2 только после выполнения T2, а T2 сможет увидеть обновление T1 только после выполнения T1.
4. ***Долговечность.*** Когда транзакция выполнена, ее обновления сохраняются, даже если в следующий момент произойдет сбой системы.

Восстановление транзакции.

Транзакция начинается с успешного выполнения оператора BEGIN TRANSACTION) и заканчивается успешным выполнением либо оператора COMMIT, либо ROLLBACK. Оператор COMMIT устанавливает так называемую точку фиксации (которая в коммерческих продуктах также называется точкой синхронизации (syncpoint). Точка фиксации соответствует концу логической единицы работы и, следовательно, точке, в которой база данных находится (или будет находиться) в состоянии согласованности. В противовес этому, выполнение оператора ROLLBACK вновь возвращает базу данных в состояние, в котором она была во время операции BEGIN TRANSACTION, т.е. в предыдущую точку фиксации.

Случаи установки точки фиксации:

1. Все обновления, совершенные программой с тех пор, как установлена предыдущая точка фиксации, выполнены, т.е. стали постоянными. Во время выполнения все такие обновления могут расцениваться только как пробные (в том

смысле, что они могут быть не выполнены, например прокручены назад). Гарантируется, что однажды зафиксированное обновление так и останется зафиксированным (это и есть определение понятия "зафиксировано").

2. Все позиционирование базы данных утеряно, и все блокировки кортежей реализованы. Позиционирование базы данных здесь означает, что в любое конкретное время программа обычно адресована определенным кортежам. Эта адресуемость в точке фиксации теряется.

Следовательно, система может выполнить откат транзакции как явно – например по команде ПО с которым работает пользователь, так и неявно – для любой программы, которая по какой-либо причине не достигла запланированного завершения операций, входящих в транзакцию.

Из этого видно, что транзакции – это не только логические единицы работы, но также и единицы восстановления при неудачном выполнении операций. При успешном завершении транзакции система гарантирует, что обновления постоянно установлены в базе данных, даже если система потерпит крах в следующий момент. Возможно, что в системе произойдет сбой после успешного выполнения COMMIT, но перед тем, как, обновления будут физически записаны в базу данных (они все еще могут оставаться в буфере оперативной памяти и таким образом могут быть утеряны в момент сбоя системы). Даже если подобное случилось, процедура перезагрузки системы все равно должна устанавливать эти обновления в базу данных, исследуя соответствующие записи в файле регистрации. Из этого следует, что файл регистрации должен быть физически записан перед завершением операции COMMIT. Это важное правило ведения файла регистрации известно как протокол предварительной записи в журнал (т.е. запись об операции осуществляется перед ее выполнением). Таким образом, процедура перезагрузки сможет восстановить любые успешно завершенные транзакции, хотя их обновления не были записаны физически до аварийного отказа системы. Следовательно, как указывалось ранее, транзакция действительно является единицей восстановления.

Параллелизм.

Термин параллелизм означает возможность одновременной обработки в СУБД многих транзакций с доступом к одним и тем же данным, причем в одно и то же время. В такой системе для корректной обработки параллельных транзакций без возникновения конфликтных ситуаций необходимо использовать некоторый метод управления параллелизмом.

Каждый метод управления параллелизмом предназначен для решения некоторой конкретной задачи. Тем не менее, при обработке правильно составленных транзакций возникают ситуации, которые могут привести к получению неправильного результата из-за взаимных помех среди некоторых транзакций. (Обратите внимание, что вносящая помеху транзакция сама по себе может быть правильной. Неправильный конечный результат возникает по причине бесконтрольного чередования операций из двух правильных транзакций). Основные проблемы, возникающие при параллельной обработке транзакций следующие:

1. проблема потери результатов обновления;
2. проблема незафиксированной зависимости;
3. проблема несовместимого анализа.

Проблема потери результатов обновления.

Рассмотрим ситуацию, показанную на рис 11.1. В такой интерпретации: транзакция А извлекает некоторый кортеж р в момент времени t1; транзакция В извлекает некоторый кортеж р в момент времени t2; транзакция А обновляет некоторый кортеж р (на основе значений, полученных в момент времени t1) в момент времени t3; транзакция В обновляет тот же кортеж р (на основе значений, полученных в момент времени t2, которые имеют те же значения, что и в момент времени t1) в момент времени t4. Однако результат операции обновления, выполненной транзакцией А, будет утерян, поскольку в момент времени t4 она не будет учтена и потому будет "отменена" операцией обновления, выполненной транзакцией В.

Транзакция A	Время	Транзакция B
Извлечение кортежа p	t1	–
–	t2	Извлечение кортежа p
Обновление кортежа p	t3	–
–	t4	Обновление кортежа p

Рис 11.1. Потеря в момент времени t4 результатов обновления, выполненного транзакцией A.

Проблема незафиксированной зависимости.

Проблема незафиксированной зависимости появляется, если с помощью некоторой транзакции осуществляется извлечение (или, что еще хуже, обновление) некоторого кортежа, который в данный момент обновляется другой транзакцией, но это обновление еще не закончено. Таким образом, если обновление не завершено, существует некоторая вероятность того, что оно не будет завершено никогда. (Более того, в подобном случае может быть выполнен возврат к предыдущему состоянию кортежа с отменой выполнения транзакции.) В таком случае, в первой транзакции будут принимать участие данные, которых больше не существует. Эта ситуация показана на рис. 11.2, рис.11. 3.

В первом примере (рис. 11.3) транзакция A в момент времени t2 встречается с невыполненным обновлением (оно также называется невыполненным изменением). Затем это обновление отменяется в момент времени t3. Таким образом, транзакция A выполняется на основе фальшивого предположения, что кортеж p имеет некоторое значение в момент времени t2, тогда как на самом деле он имеет некоторое значение, существовавшее еще в момент времени t1. В итоге после выполнения транзакции A будет получен неверный результат. Кроме того, обратите внимание, что отмена выполнения транзакции B может произойти не по вине транзакции B, а, например, в результате краха системы. (К этому времени выполнение транзакции A может быть уже завершено, а потому крушение системы не приведет к отмене выполнения транзакции A.)

Транзакция A	Время	Транзакция B
–	t1	Обновление кортежа p
Извлечение кортежа p	t2	–
–	t3	Отмена выполнения транзакции

Рис. 11.2. Транзакция A становится зависимой от невыполненного изменения в момент времени t2.

Второй пример, приведенный на рис. 11/3, иллюстрирует другой случай. Не только транзакция A становится зависимой от изменения, не выполненного в момент времени t2, но также в момент времени t3 фактически утрачивается результат обновления, поскольку отмена выполнения транзакции B в момент времени t3 приводит к восстановлению кортежа p к исходному значению в момент времени t1. Это еще один вариант проблемы потери результатов обновления.

Транзакция A	Время	Транзакция B
–	t1	Обновление кортежа p
Обновление кортежа p	t2	–
–	t3	Отмена выполнения транзакции

Рис. 11.3. Транзакция A обновляет невыполненное изменение в момент времени t2, и результаты этого обновления утрачиваются в момент времени t3.

Проблема несовместимого анализа.

На рис. 11.4 показаны транзакции A и B, которые выполняются для кортежей со счетами (табл. 11.1). При этом транзакция A суммирует балансы, транзакция B производит перевод суммы 10 со счета 3 на счет 1. Полученный в итоге транзакции A результат 110, очевидно, неверен, и если он будет записан в базе данных, то в ней может возникнуть проблема несовместимости. В таком случае говорят, что транзакция A встретилась с несовместимым состоянием и на его основе был выполнен несовместимый анализ. Обратите внимание на следующее различие между этим примером и предыдущим: здесь не идет речь о зависимости транзакции A от транзакции B, так как

транзакция B выполнила все обновления до того, как транзакция A извлекла СЧЕТ 3.

Таблица 11.1. Остатки на счетах до выполнения транзакций.

Счет	СЧЕТ 1	СЧЕТ 2	СЧЕТ 3
Остаток	40	50	30

Транзакция A	Время	Транзакция B
Извлечение кортежа СЧЕТ 1: СУММА = 40	t1	–
Извлечение кортежа СЧЕТ 1: СУММА = 90	t2	–
–	t3	Извлечение кортежа СЧЕТ 3:
–	t4	Обновление кортежа СЧЕТ 3: 30 → 20
–	t5	Извлечение кортежа СЧЕТ 1:
–	t6	Обновление кортежа СЧЕТ 1: 40 → 50
–	t7	Завершение выполнения транзакции
Извлечение кортежа СЧЕТ 3: СУММА = 110 (а не 120)	t8	–

Рис. 11.4. Транзакция A выполнила несовместимый анализ.

11.3. Понятие блокировки

Описанные выше проблемы могут быть разрешены с помощью методики управления параллельным выполнением процессов под названием блокировка. Ее основная идея очень проста: в случае, когда для выполнения некоторой транзакции необходимо, чтобы некоторый объект (обычно это кортеж базы данных) не изменялся непредсказуемо и без ведома этой транзакции (как это обычно бывает), такой объект блокируется. Таким образом, эффект блокировки состоит в том, чтобы "заблокировать доступ к этому объекту со стороны других транзакций", а значит, предотвратить непредсказуемое изменение этого объекта. Следовательно, первая транзакция в состоянии выполнить всю необходимую обработку с учетом того, что обрабатываемый объект остается в стабильном состоянии настолько долго, насколько это нужно.

1. Предположим, что в системе поддерживается два типа блокировок: блокировка без взаимного доступа (монопольная блокировка), называемая X-блокировкой (X locks – exclusive locks), и блокировка с взаимным доступом, называемая S-блокировкой (S locks - Shared locks).

Замечание. X- и S-блокировки иногда называют блокировками записи и чтения соответственно. Предположим, что X- и S-блокировки единственно возможные, хотя в коммерческих системах существуют блокировки других типов. Кроме того, допустим, что в кортежи являются единственным типом "блокируемого объекта", хотя опять же н в коммерческих системах могут блокироваться и другие объекты. Ниже показано функционирование механизма блокировок.

2. Если транзакция A блокирует кортеж p без возможности взаимного доступа (X-блокировка), то запрос другой транзакции B с блокировкой этого кортежа p будет отменен.
3. Если транзакция A блокирует кортеж p с возможностью взаимного доступа (S-блокировка), то:
    a. запрос со стороны некоторой транзакции B на X-блокировку кортежа будет отвергнут;
    b. запрос со стороны некоторой транзакции B на S-блокировку кортежа p будет принят (т.е. транзакция B также будет блокировать кортеж p с помощью S-блокировки).

Эти правила можно наглядно представить в виде матрицы совместимости, показанной на рис. 11.5, и интерпретировать ее следующим образом. Рассмотрим некоторый кортеж p и предположим, что транзакция A блокирую кортеж p различными типами блокировки (это обозначено соответствующими символами S и X, а отсутствие блокировки — прочерком). Предположим также, что некоторая транзакция B запрашивает блокировку кортежа p, что обозначено в первом слева столбце матрицы на рис 5 (для полноты картины в таблице также приведен случай "отсутствия блокировки"). В других ячейках матрицы символ N обозначает конфликтную ситуацию (запрос со стороны транзакции B не может быть удовлетворен, и сама эта транзакция переходит в состояние ожидания), а Y – полную совместимость

(запрос со стороны транзакции B удовлетворен). Очевидно, что эта матрица является симметричной.

	X	S	–
X	N	N	Y
S	N	Y	Y
–	Y	Y	Y

Рис. 11.5. Матрица совместимости для X- и S-блокировки.

Введем протокол доступа к данным, который на основе введения только что описанных X- и S-блокировки позволяет избежать возникновения проблем параллелизма.

1. Транзакция, предназначенная для извлечения кортежа, прежде всего должна наложить S-блокировку на этот кортеж.
2. Транзакция, предназначенная для обновления кортежа, прежде всего должна наложить X–блокировку на этот кортеж. Иначе говоря, если, например, для последовательности действий типа извлечение/обновление для кортежа уже задана S-блокировка, то ее необходимо заменить X-блокировкой.

Замечение: Блокировки в транзакциях обычно задаются неявным образом: например, запрос на "извлечение кортежа" является неявным запросом с S-блокировкой, а запрос на "обновление кортежа" – неявным запросом с X-блокировкой соответствующего кортежа. При этом под термином "обновление" (как и ранее) подразумеваются помимо самих операций обновления также операции вставки и удаления.

1. Если запрашиваемая блокировка со стороны транзакции B отвергается из-за конфликта с некоторой другой блокировкой со стороны транзакции A, то транзакция B переходит в состояние ожидания. Причем транзакция B будет находиться в состоянии ожидания до тех пор, пока не будет снята блокировка, заданная транзакцией A. В системе обязательно должны быть предусмотрены способы устранения бесконечно долгого состояния ожидания транзакции B.
2. X–блокировки сохраняются вплоть до конца выполнения транзакции (до операции "завершение выполнения" или

"отмена выполнения"). S-блокировки также обычно сохраняются вплоть до этого момента.

Решение проблем параллелизма.

Рассмотрим решение проблем параллелизма с помощью механизма блокировок.

Проблема потери результатов обновления.

На рис. 11.6 приведена измененная версия процесса, показанного на рис. 11.1, с учетом применения протокола блокировки для чередующихся операций. Операция обновления для транзакции A в момент времени t3 не будет выполнена, поскольку она является неявным запросом с заданием X-блокировки для кортежа p, а этот запрос вступает в конфликт с S-блокировкой, уже заданной транзакцией B. Таким образом, транзакция A переходит в состояние ожидания. По аналогичным причинам транзакция B переходит в состояние ожидания в момент времени t4.Обновления теперь не утрачиваются, однако возникает новая проблема – бесконечное ожидание или тупиковая ситуация. Способы решения этой проблемы рассматриваются ниже.

Транзакция A	Время	Транзакция B
Извлечение кортежа p (задание S-блокировки для p)	t1	–
–	t2	Извлечение кортежа p (задание S-блокировки для p)
Обновление кортежа p (задание X-блокировки для p)	t3	–
Ожидание	t4	Обновление кортежа p (задание X-блокировки для p)
Ожидание		Ожидание

Рис. 11.6. Хотя обновления не утрачиваются, но в момент времени t4 возникает тупиковая ситуация.

Проблема незафиксированной зависимости.

На рис. 11.7, рис. 11.8 приведены в измененном виде примеры, показанные ранее на рис. 2 и рис. 3 соответственно. Они демонстрируют чередующееся выполнение операций согласно описанному выше протоколу блокировки. Операция для транзакции A в момент времени t2 (извлечение на рис. 11.7 и обновление на рис. 11.8) не будет выполнена. Дело в том, что она является неявным запросом с заданием блокировки для кортежа p, а этот запрос вступает в конфликт с X-блокировкой, уже заданной транзакцией B. Таким образом, транзакция A переходит в состояние ожидания до тех пор, пока не будет прекращено выполнение транзакции B (до операции окончания или отмены выполнения транзакции B). Тогда заданная транзакцией B блокировка будет снята и транзакция A может быть выполнена. Причем транзакция A будет иметь дело с некоторым фиксированным значением (либо существовавшим до выполнения транзакции B при отмене ее выполнения, либо полученным после выполнения транзакции B). В любом случае транзакция A больше не зависит от незафиксированного обновления.

Транзакция A	Время	Транзакция B
–	t1	Обновление кортежа p (задание X-блокировки для p)
Извлечение кортежа p (задание S-блокировки для p)	t2	–
Ожидание	t3	Отмена выполнения транзакции (снятие X-блокировки для p)
Итог: Извлечение кортежа p (задание S-блокировки для p)	t4	

Рис. 11.7. Транзакция A предохраняется от выполнения операций с незафиксированным изменением в момент времени t2.

Транзакция A	Время	Транзакция B
–	t1	Обновление кортежа p (задание X-блокировки для p)
Обновление кортежа p (задание X-блокировки для p)	t2	–
Ожидание	t3	Отмена выполнения транзакции (снятие X-блокировки для p)
Итог: Обновление кортежа p (задание X-блокировки для p)	t4	

Рис. 11.8. Транзакция A предохраняется от выполнения операций с незафиксированным изменением в момент времени t2.

Проблема несовместимого анализа.

На рис. 11.9 приведена измененная версия отношения (рис. 11.4) с перечислением чередующихся транзакций согласно протоколу блокировки. Операция обновления для транзакции B в момент времени t6 не будет выполнена. Дело в том, что она является неявным запросом с заданием X-блокировки для кортежа СЧЕТ 1, а этот запрос вступает в конфликт с S-блокировкой, уже заданной транзакцией A. Таким образом, транзакция B переходит в состояние ожидания. Точно так же операция извлечения для транзакции A в момент времени t7 не будет выполнена. Дело в том, что она является неявным запросом с заданием S-блокировки для кортежа СЧЕТ 3, а этот запрос вступает в конфликт с X-блокировкой, уже заданной транзакцией B. Таким образом, транзакция A переходит в состояние ожидания. Следовательно, блокировка хотя и помогает решить одну проблему (а именно проблему несовместимого анализа), но приводит к необходимости решения другой проблемы (а именно проблемы возникновения тупиковой ситуации).

СЧЕТ 1 40	СЧЕТ 2 50	СЧЕТ 3 30
Транзакция A	Время	Транзакция B
Извлечение кортежа СЧЕТ 1: (задание S-блокировки для СЧЕТ 1) СУММА = 40	t1	–
Извлечение кортежа СЧЕТ 1: (задание S-блокировки для СЧЕТ 2) СУММА = 90	t2	–
–	t3	Извлечение кортежа СЧЕТ 3: (задание S-блокировки для СЧЕТ 3)
–	t4	Обновление кортежа СЧЕТ 3: (задание X-блокировки для СЧЕТ 3) 30 → 20
–	t5	Извлечение кортежа СЧЕТ 1: (задание S-блокировки для СЧЕТ 1)
–	t6	Обновление кортежа СЧЕТ 1: (задание X-блокировки для СЧЕТ 1) 40 → 50
–	t7	Ожидание
Извлечение кортежа СЧЕТ 3: (задание S-блокировки для СЧЕТ 3)	t8	Ожидание
Ожидание		Ожидание

Рис. 11.9. Проблема несовместимого анализа разрешается, но в момент времени t7 возникает тупиковая ситуация.

11.4. Тупиковые ситуации

Как было показано выше, блокировку можно использовать для разрешения трех основных проблем, возникающих при параллельной обработке кортежей. К сожалению, использование блокировок приводит к возникновению другой проблемы – тупиковой ситуации. На рис. 11.10 показан обобщенный пример этой проблемы, в котором p1 и p2 представляют любые блокируемые объекты, необязательно кортежи базы данных, а выражения типа "блокировка ... без взаимного доступа" представляют любые операции с наложением блокировки без взаимного доступа, заданные как явно, так и неявно.

Транзакция A	Время	Транзакция B
Блокировка p1 без взаимного доступа	t1	–
–	t2	Блокировка p2 без взаимного доступа
Блокировка p2 без взаимного доступа	t3	–
Ожидание	t4	Блокировка p1 без взаимного доступа
Ожидание		Ожидание

Рис. 11.10. Пример тупиковой ситуации.

Тупиковая ситуация возникает тогда, когда две или более транзакции одновременно находятся в состоянии ожидания, причем для продолжения работы каждая из транзакций ожидает прекращения выполнения другой транзакции.

Для обнаружения тупиковой ситуации следует обнаружить цикл в диаграмме состояний ожидания, т.е. в перечне "транзакций, которые ожидают окончания выполнения других транзакций" Поиск выхода из тупиковой ситуации состоит в выборе одной из заблокированных транзакций в качестве жертвы и отмене ее выполнения. Таким образом, с нее снимается блокировка, а выполнение другой транзакции может быть возобновлено.

На практике не все системы в состоянии обнаружить тупиковую ситуацию. Например, в некоторых из них используется хронометраж выполнения транзакций, и сообщение о возникновении тупиковой ситуации поступает, если транзакция не выполняется за некоторое предписанное заранее время.

Следует обратить внимание на то, что транзакция-жертва признается "некорректной" и отменяется "не из-за собственной некорректности". В некоторых системах предусмотрен автоматический перезапуск транзакции с самого начала при условии, что обстоятельства, которые привели к тупиковой ситуации, не повторятся вновь. А в других системах в программу, связанную с данной транзакцией, просто посылается сообщение о "вызвавшей тупиковую ситуацию транзакции-жертве" для

обработки этой ситуации в самой программе С точки зрения программирования приложений предпочтительнее первый из этих подходов. Но несмотря на это, всегда рекомендуется решать данную проблему с точки зрения пользователя.

Способность к упорядочению.

Чередующееся выполнение заданного множества транзакций будет верным, если оно упорядочено, т.е. при его выполнении будет получен такой же результат, как и при последовательное выполнении тех же транзакций. Обосновать это утверждение помогут следующие замечания:

1. Отдельные транзакции считаются верными, если при их выполнении база данных переходит из одного непротиворечивого состояния в другое непротиворечивое состояние.
2. Выполнение транзакций одна за другой в любом последовательном порядке также является верным. При этом под выражением "любой последовательный порядок" подразумевается, что используются независимые друг от друга транзакции.
3. Чередующееся выполнение транзакций, следовательно, является верным, если оно эквивалентно некоторому последовательному выполнению, т.е. если оно подлежит упорядочению.

Возвращаясь к приведенным выше примерам (рис. 11.1 – рис. 11.4), можно отметить, что проблема в каждом случае заключалась в том, что чередующееся выполнение транзакций не было упорядочено, т.е. не было эквивалентно выполнению либо сначала транзакции А, а затем транзакции В, либо сначала транзакции В, а затем транзакции А.

Для заданного набора транзакций любой порядок их выполнения (чередующийся или какой-либо другой) называется графиком запуска. Выполнение транзакций по одной без их чередования называется последовательным графиком запуска, а непоследовательное выполнение транзакций – чередующимся графиком запуска или непоследовательным графиком запуска. Два графика называются эквивалентными, если при их выполнении будет получен одинаковый результат, независимо от исходного состояния базы данных. Таким образом, график запуска является

верным (т.е. допускающим возможность упорядочения), если он эквивалентен некоторому последовательному графику запуска.

При выполнении двух различных последовательных графиков запуска, содержащих одинаковый набор транзакций, можно получить совершенно различные результаты. Поэтому выполнение двух различных чередующихся графиков запуска с одинаковыми транзакциями может также привести к различным результатам, которые могут быть восприняты как верные.

Теорема двухфазной блокировки (не имеет отношения к протоколу двухфазной фиксации), которая может быть сформулирована следующим образом:

Если все транзакции подчиняются "протоколу двухфазной блокировки", то для всех возможных чередующихся графиков запуска существует возможность упорядочения.

При этом протокол двухфазной блокировки, в свою очередь, формулируется следующим образом.

1. Перед выполнением каких-либо операций с некоторым объектом (например, с кортежем базы данных) транзакция должна заблокировать этот кортеж.
2. После снятия блокировки транзакция не должна накладывать никаких других блокировок.

Таким образом, транзакция, которая подчиняется этому протоколу, характеризуется двумя фазами: фазой наложения блокировки и фазой снятия блокировки.

Характеристика упорядочения может быть выражена следующим образом. Если A и B являются любыми двумя транзакциями некоторого графика запуска, допускающего возможность упорядочения, то либо A логически предшествует B, либо B логически предшествует A, т.е. либо B использует результаты выполнения транзакции A, либо A использует результаты выполнения транзакции B. (Если транзакция A приводит к обновлению кортежей p, q, ... r и транзакция B использует эти кортежи в качестве входных данных, то используются либо все обновленные с помощью A кортежи, либо полностью не обновленные кортежи до выполнения транзакции A, но никак не их смесь.) Наоборот, график запуска является неверным и не подлежит упорядочению, если результат выполнения транзакций не соответствует либо сначала

выполнению транзакции A, а затем транзакции B, либо сначала выполнению транзакции B, а затем транзакции A.

В настоящее время с целью понижения требований к ресурсам и, следовательно, повышения производительности и пропускной способности в реальных системах обычно предусмотрено использование не двухфазных транзакций, а транзакций с "ранним снятием блокировки" (еще до выполнения операции прекращения транзакции) и наложением нескольких блокировок. Однако следует понимать, что использование таких транзакций сопряжено с большим риском. Действительно, при использовании недвухфазной транзакции A предполагается, что в данной системе не существует никакой другой чередующейся с ней транзакции B (в противном случае в системе возможно получение ошибочных результатов).

Уровни изоляции транзакции.

Термин уровень изоляции, грубо говоря, используется для описания степени вмешательства параллельных транзакций в работу некоторой заданной транзакции. Но при обеспечении возможности упорядочения не допускается никакого вмешательства, иначе говоря, уровень изоляции должен быть максимальным. Однако, как уже отмечалось, в реальных системах по различным причинам обычно допускаются транзакции, которые работают на уровне изоляции ниже максимального.

Уровень изоляции обычно рассматривается как некоторое свойство транзакции. В реальных СУБД может быть реализовано различное количество уровней изоляции.

Кроме того, помимо кортежей могут блокироваться другие единицы данных, например, целое отношение, база данных или (пример противоположного характера) некоторое значение атрибута внутри заданного кортежа.

11.5 Поддержка в языке SQL

SQL поддерживает операции COMMIT и ROLLBACK для фиксации и отката транзакции соответственно.

Специальный оператор SET TRANSACTION используется для определения некоторых характеристик транзакции, которую нужно будет инициировать, такие, как режим доступа и уровень изоляции.

В стандарте языка SQL не предусмотрена поддержка явным образом возможности блокировки (фактически, блокировка в нем вообще не упоминается). Блокировки накладываются неявно, при выполнении операторов SQL.

Оптимизация запросов.

Оптимизация запросов — это 1) функция СУБД, осуществляющая поиск оптимального плана выполнения запросов из всех возможных для заданного запроса, 2) процесс изменения запроса и/или структуры БД с целью уменьшения использования вычислительных ресурсов при выполнении запроса. Один и тот же результат может быть получен СУБД различными способами (планами выполнения запросов), которые могут существенно отличаться как по затратам ресурсов, так и по времени выполнения. Задача оптимизации заключается в нахождении оптимального способа.

В реляционной СУБД оптимальный план выполнения запроса — это такая последовательность применения операторов реляционной алгебры к исходным и промежуточным отношениям, которая для конкретного текущего состояния БД (её структуры и наполнения) может быть выполнена с минимальным использованием вычислительных ресурсов.

В настоящее время известны две стратегии поиска оптимального плана:

- грубой силы путём оценки всех перестановок соединяемых таблиц, используемых способов входа в таблицы и типов соединения (т. е. полный перебор вариантов);
- на основе генетического алгоритма путём оценки ограниченного числа перестановок.

Также некоторые СУБД позволяют программисту вмешиваться в поиск оптимального плана в различной степени, от минимального влияния до полного и чёткого указания какой именно план запроса использовать.

Планы выполнения запроса сравниваются исходя из множества факторов (реализации в различных СУБД отличаются), в том числе:

- потенциальное число строк, извлекаемое из каждой таблицы, получаемое из статистики;
- наличие индексов;
- возможность выполнения слияний (merge-join);
- способ чтения записей/блоков таблиц/индексов.

В общем случае соединение выполняется вложенными циклами. Однако этот алгоритм может оказаться менее эффективен, чем специализированные алгоритмы. Например, если у сливаемых таблиц есть индексы по соединяемым полям, или одна или обе таблицы достаточно малы, чтобы быть отсортированными в памяти, то исследуется возможность выполнения слияний.

Как уже отмечалось, суть оптимизации заключается в поиске минимума функции стоимости от перестановки таблиц. Независимо от стратегии, оптимизатор обязан уметь анализировать стоимость для произвольной перестановки, в то время как сами перестановки для анализа предоставляются другим алгоритмом. Исследуемое множество перестановок может отличаться от всего пространства перестановок. Исходя из этого, обобщённый алгоритм работы оптимизатора можно записать так:

```
Перебор всех планов в поисках наилучшего:
ТекущийПорядокТаблиц :=
НайтиИсходныйПорядокТаблиц;
ЛучшийПорядокТаблиц := ТекущийПорядокТаблиц;
НаименьшаяСтоимость :=
МаксимальноВозможнаяСтоимость;
Выполнять
  Стоимость :=
ОценитьСтоимость(ТекущийПорядокТаблиц);
  Если Стоимость < НаименьшаяСтоимость То
    ЛучшийПорядокТаблиц :=
ТекущийПорядокТаблиц;
    НаименьшаяСтоимость := Стоимость;
  КонецЕсли;

  ТекущийПорядокТаблиц :=
НайтиСледующийПорядокТаблиц;
Пока (ДоступенСледующийПорядокТаблиц);
```

Стратегия грубой силы.

В теории, при использовании стратегии грубой силы оптимизатор запросов исследует все пространство перестановок всех исходных выбираемых таблиц и сравнивает суммарные оценки стоимости выполнения соединения для каждой перестановки. На практике, при разработке System R было предложено ограничить пространство исследования только левосторонними соединениями, чтобы при выполнении запроса одна из таблиц всегда была представлена образом на диске. Исследование нелевосторонних соединений имеет смысл если таблицы, входящие в соединения, расположены на более чем одном узле.

Стратегия на основе генетического алгоритма.

С ростом числа таблиц в запросе количество возможных перестановок растет как n!, следовательно, пропорционально растет и время оценки для каждой из них. Это делает проблематичным оптимизацию запросов на основе большого числа таблиц. В поисках решения этой проблемы в 1991 году Kristin Bennett, Michael Ferris, Yannis Ioannidis предложили использовать генетический алгоритм для оптимизации запросов, который дает субоптимальное решение за линейное время.

При использовании генетического алгоритма исследуется только часть пространства перестановок. Таблицы, участвующие в запросе, кодируются в хромосомы. Над ними выполняются мутации и скрещивания. На каждой итерации выполняется восстановление хромосом для получения осмысленной перестановки таблиц и отбор хромосом, которые дают минимальные оценки стоимости. В результате отбора остаются только те хромосомы, которые дают меньшее, по сравнению с предыдущей итерацией, значение функции стоимости. Таким образом происходит исследование и нахождение локальных минимумов функции стоимости. Предполагается, что глобальный минимум не дает существенных преимуществ, по сравнению с лучшим локальным минимумом. Алгоритм повторяется несколько итераций, после чего выбирается наиболее эффективное решение.

Для каждой таблицы в каждой из перестановок по статистике оценивается возможность использования индексов. Перестановка с минимальной оценкой и есть итоговый план выполнения запроса.

Оценка стоимости выполнения запроса.

Различные реализации СУБД могут учитывать затраты ресурсов различных компонент системы. В Oracle 9i рассматривают количество операций чтения блоков данных, количество тактов процессора и объемы дополнительной дисковой памяти. Далее эти показатели нормируются и приводятся к единицам измерения количества одноблочных чтений [7]. Формула вычисления стоимости Cost выглядит следующим образом:

$$Cost = (\#SRds * sreadtim + \#MRds * mreadtim + \#CPUCycles / CPUSpeed) / sreadtim$$

где

#SRds - оценочное количество одноблочных дисковых чтений;

#MRds - оценочное количество многоблочных дисковых чтений;

#CPUCycles - асимптотически точные оценки количества операций, выполняемых процессором;

#sreadtim - среднее время одного одноблочного дискового чтения;

#mreadtim - среднее время одного многоблочного дискового чтения;

#CPUSpeed - количество операций, выполняемых процессором в единицу времени.

Контрольные вопросы:

1. Что такое транзакция?
2. Перечислите основные свойства ACID.
3. Как реализуется механизм синхронизации транзакций?
4. Что такое S-блокировка?
5. Что такое тупиковые ситуации?
6. Как производится оценка времени выполнения запроса?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p.
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p.

Глава – XII. Администрирование и безопасность базы данных.

12.1. Восстановление базы данных

Компьютерные системы порою дают сбои. Причин тому множество: поломки аппаратного обеспечения, дефекты в программах, неточности в описаниях ручных процедур и ошибочные человеческие действия. Все перечисленные виды ошибок могут возникать и возникают в приложениях баз данных. Поскольку база данных совместно используется множеством людей и зачастую является ключевым элементом функционирования организации, важно как можно быстрее ее восстановить.

Это ставит перед нами несколько задач. Во-первых, с точки зрения бизнеса, работа должна продолжаться. Например, выполнение заказов, осуществление финансовых транзакций и составление упаковочных листов должно быть организовано вручную. Позже, когда приложение базы данных вновь заработает, можно будет ввести новые данные. Во-вторых, персонал, ответственный за компьютерную систему, должен восстановить ее как можно быстрее и привести как можно ближе к тому состоянию, которое было до сбоя. В-третьих, пользователи должны знать, что требуется сделать, когда система вновь начнет функционировать. Возможно, придется повторно ввести какие-то данные, и пользователям нужно знать, насколько далеко им следует возвращаться назад.

Когда происходит сбой, невозможно просто устранить проблему и продолжать обработку. Если даже при этом не были потеряны никакие данные (для чего необходимо, чтобы все виды памяти были энергонезависимыми, – совершенно нереалистичное предположение), то синхронизация и планирование обработки слишком сложны, чтобы состояние системы можно было воспроизвести в точности. Чтобы продолжить работу точно с того места, где она была прервана, операционной системе потребовалось бы громадное количество избыточных данных и времени на их обработку. Невозможно прокрутить время назад и вернуть все электроны в те же конфигурации, в которых они находились на момент возникновения ошибки. Возможны два подхода к восстановлению базы данных: повторная обработка и откат-накат.

Восстановление путем повторной обработки.

Поскольку обработка не может быть возобновлена точно с того места, где она была прервана, следующая возможность – отойти назад до некоторой известной точки и возобновить обработку с нее. Простейший способ такого восстановления – это периодически делать копию базы данных (называемую снимком базы данных (database save)) и хранить записи обо всех транзакциях, которые были выполнены со времени последнего копирования. Затем при возникновении сбоя персонал может восстановить базу данных по ее снимку и заново произвести все транзакции.

К сожалению, эта простая стратегия, как правило, нереализуема. Во-первых, повторное выполнение транзакций занимает столько же времени, сколько оно заняло до возникновения ошибки. Если компьютер имеет напряженный график работы, система может так никогда и не «догнать» свое исходное состояние. Во-вторых, при параллельной обработке транзакций события происходят асинхронно. Небольшие вариации в действиях человека – например, когда пользователь чуть медленнее вставляет дискету или читает сообщение электронной почты, прежде чем ответить на запрос приложения, – могут изменить порядок обработки параллельных транзакций. Поэтому может случиться, что хотя изначально последний билет на данный рейс достался клиенту А, при повторной обработке его получит клиент В. В связи с этим повторная обработка обычно не является реально осуществимой формой восстановления от сбоев в системах параллельной обработки.

Восстановление через откат-накат.

Второй подход к восстановлению заключается в том, чтобы периодически делать копии (снимки) базы данных и вести журнал всех изменений, произведенных транзакциями в базе данных со времени последнего копирования. В этом случае, когда происходит сбой, можно использовать один из двух методов. При первом методе, который называется накатом (rollforward), база данных восстанавливается до сохраненного состояния, после чего выполняются все правильные транзакции. (Мы не обрабатываем транзакции повторно, поскольку прикладные программы не участвуют в процессе восстановления. Все, что мы делаем – это

производим изменения в базе данных согласно записям в журнале.)

Второй метод называется откатом (rollback). При этом методе мы отменяем изменения, произведенные в базе данных ошибочными или частично выполненными транзакциями. Затем повторно запускаются правильные транзакции, которые выполнялись в момент возникновения сбоя.

Оба эти метода требуют ведения журнала (log) результатов транзакций. Этот журнал содержит записи изменений, произведенных с данными, в хронологическом порядке. Прежде чем транзакция будет выполнена, ее необходимо записать и журнал. Тогда, если крах системы произойдет в интервале между записью транзакции в журнал и выполнением ее, то в худшем случае у нас будет иметься запись о невыполненной транзакции. Если, напротив, транзакции сначала выполняются, а затем уже записываются в журнал, то возможен нежелательный вариант, когда изменения в базе данных произведены, но запись об этих изменениях отсутствует. В такой ситуации неискушенный пользователь может повторно ввести транзакцию, которая уже выполнена.

В случае сбоя журнал используется как для отмены, так и для повторного выполнения транзакций (рис. 12.1). Чтобы была возможна отмена транзакций, журнал должен содержать копию каждой записи (или страницы) базы данных, сделанную перед ее изменением. Такие записи называются исходными образами (before-images). Отмена транзакции производится путем последовательной записи в базу данных исходных образов всех произведенных ею изменений. Чтобы было возможно повторное выполнение транзакций, журнал должен содержать копию каждой записи (или страницы) базы данных, сделанную после ее изменения. Такие записи называются конечными образами (after-images). Транзакция выполняется повторно путем последовательной записи в базу данных конечных образов всех произведенных ею изменений.

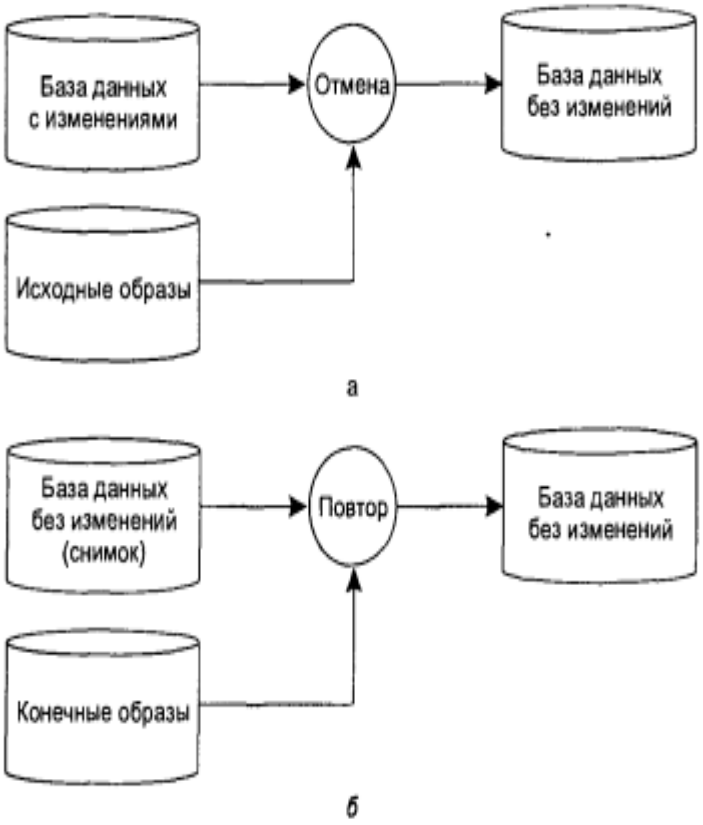


Рис. 12.1. Откат и накат транзакций: а – отмена изменений в базе данных (откат); б – повторное выполнение изменений (накат).

Если есть журнал с исходными и конечными образами, то отмена и повторное выполнение транзакций происходит элементарно (но мы все равно опишем этот процесс). Чтобы отменить транзакцию, программа восстановления просто заменяет каждую измененную запись ее исходным образом. Когда все исходные образы будут восстановлены, транзакция будет отменена. Чтобы повторно выполнить транзакцию, программа восстановления начинает с версии базы данных, существовавшей на момент начала данной транзакции, и записывает в базу все конечные образы. Как уже говорилось, это действие подразумевает, что имеется снимок базы данных с более ранней ее версией.

Восстановление базы данных до ее последнего снимка и повторное выполнение всех транзакций может потребовать значительного времени. Чтобы уменьшить задержку, в СУБД иногда используются контрольные точки. ***Контрольная точка (checkpoint)*** – это точка синхронизации между базой данных и журналом транзакций. Для вставки контрольной точки СУБД

отклоняет новые запросы, завершает обработку текущих запросов и сбрасывает свои буферы на диск. Затем СУБД ждет, пока операционная система не сообщит, что все текущие операции записи на диск и в журнал успешно выполнены. Теперь база данных и журнал синхронизированы. После этого в журнале делается запись о контрольной точке. Позже база данных может быть восстановлена с контрольной точки, при этом нужно будет записать конечные образы только тех транзакций, которые начались после контрольной точки.

Вставка контрольной точки – недорогая операция, и можно выполнять три или четыре (а можно и больше) таких операции в час. Таким образом, для восстановления потребуется не более 15-20 минут работы. Большинство СУБД вставляют контрольные точки автоматически, делая человеческое вмешательство ненужным.

12.2. Управление структурой базы данных

Управление структурой базы данных включает участие в первоначальном проектировании и реализации базы данных, а также руководство и контроль в процессе внесения в нее изменений. В идеальном случае отдел администрирования привлекается к работе на ранней стадии разработки базы данных и ее приложений и принимает участие в изучении требований, оценке альтернатив, включая то, какую СУБД предпочтительнее использовать, и разработке структуры базы данных. Для больших организационных приложений администратор базы данных – это обычно менеджер, который руководит работой технически ориентированного персонала по проектированию базы данных.

Как говорилось ранее, при создании базы данных приходится решать несколько задач. Прежде всего, создается база данных и выделяется место на физическом носителе под саму базу и ее журналы. Затем создаются таблицы, индексы, хранимые процедуры и триггеры. Примеры этого вы увидите в следующих двух главах. Когда структуры базы данных сформированы, база заполняется информацией. В большинстве СУБД предусмотрены утилиты для записи больших объемов данных.

Конфигурирование.

После того как база данных и ее приложения будут реализованы, неизбежно будут меняться требования. Это может

быть обусловлено новыми потребностями, изменениями в бизнес-окружении, сменой политики и т. д. Когда изменение требований вызывает необходимость изменения структуры базы данных, следует действовать с большой осторожностью, потому что структурные изменения редко затрагивают только одно приложение.

Следовательно, эффективное администрирование базы данных должно включать в себя процедуры и политику, с помощью которых пользователи могли бы регистрировать свои потребности в изменениях, а все сообщество пользователей базы данных имело бы возможность обсуждать эффект от этих изменений, чтобы затем можно было принять глобальное решение о том, стоит ли их воплощать. Из-за больших размеров и сложности базы данных и ее приложений изменения иногда приводят к неожиданным результатам. Поэтому администратор базы данных должен быть готов к тому, что базу данных придется восстанавливать, и должен иметь достаточное количество информации для диагностики и устранения проблемы, вызвавшей сбой. База данных наиболее подвержена сбоям после внесения изменений в ее структуру.

Документирование.

В обязанности администратора по управлению структурой базы данных входит также ведение документации. Исключительно важно знать, какие модификации были произведены, когда и каким образом. Изменение в структуре базы данных может повлечь за собой ошибку, которая не будет проявляться в течение шести месяцев; при отсутствии должного документирования изменений диагностика такой проблемы становится почти невозможной. Могут потребоваться десятки повторных прогонов, чтобы выявить момент, когда начали появляться первые симптомы, и по этой причине важно также вести запись тестовых процедур и прогонов, сделанных для проверки произведенной модификации. Если используются стандартизированные тестовые процедуры, тестовые формы и методы ведения записей, запись тестовых результатов не должна занимать много времени.

Хотя ведение документации – процесс утомительный и бесконечный, затраченные на него усилия окупаются, когда приходит беда и документация оказывается тем, без чего невозможно решить серьезную (и дорогостоящую) проблему. В

настоящее время появляется много продуктов, облегчающих бремя ведения документации. Многие CASE-средства, например, можно использовать для документирования логической структуры базы данных. Для отслеживания изменений можно использовать программы контроля версий. Словари данпых обеспечивают составление отчетов и применение других средств для чтения и интерпретации структуры информации в базе данных.

Другая причина для тщательного документирования изменений в структуре базы данных состоит в том, чтобы должным образом использовать исторические данные. Если, например, маркетологи захотят проанализировать данные о продажах трехлетней давности, находившиеся в архивах в течение двух лет, им необходимо будет знать, какова была структура базы данных перед тем, как данные были отправлены в архив. В ответе на этот вопрос могут помочь записи, отражающие изменения в структуре базы данных. Аналогичная ситуация возникает, когда для восстановления поврежденной базы данных приходится использовать резервную копию шестимесячной давности (хотя такого происходить не должно, иногда это случается). Резервную копию можно использовать для реконструкции базы данных до того состояния, в котором она находилась на момент снятия этой копии. Затем можно в хронологическом порядке выполнить транзакции к произвести структурные изменения, чтобы восстановить базу данных до ее текущего состояния. В списке приведен перечень обязанностей администратора по управлению структурой базы данных.

Участие в разработке базы данных и приложений.

- Помощь на стадии определения требований и оценки альтернатив.
- Активная роль в проектировании и создании базы данных.
- Помощь в изменении структуры базы данных
- Поиск решений во взаимодействии с пользователями.
- Оценка того, как планируемое изменение отразится на каждом пользователе.
- Организация форума по вопросам конфигурирования.
- Готовность к устранению проблем, возникающих после внесения изменений.
- Ведение документации.

12.3. Безопасность базы данных

Защита безопасности базы данных заключается в том, что право выполнять некоторые действия дается только определенным пользователям и в определенное время. Эта цель труднодостижима, и чтобы хоть в какой-то степени к ней приблизиться, команда разработчиков базы данных должна на стадии определения требований к проекту установить для всех пользователей права и обязанности по обработке (processing rights and responsibilities). Реализация этих требований безопасности может обеспечиваться соответствующими возможностями СУБД, а при их недостаточности – логикой прикладных программ.

Права и обязанности по обработке.

В задачи администратора базы данных входит управление правами и обязанностями по обработке. Это подразумевает, что права и обязанности могут меняться с течением времени. В ходе работы с базой данных, по мере внесения изменений в приложения и в структуру СУБД возникает потребность во введении новых или изменении существующих прав и обязанностей. Администратор базы данных играет ведущую роль в обсуждении и реализации таких изменений.

Когда права по обработке определены, необходимо их реализовать. Эта задача может быть возложена на различные элементы: операционную систему, сеть, web-сервер, СУБД или приложение. В последующих двух разделах мы рассмотрим реализацию прав по обработке средствами СУБД и приложения. Описание остальных возможностей выходит за рамки данной книги.

Обеспечение безопасности средствами СУБД.

Терминология, возможности и функции безопасности СУБД варьируются от продукта к продукту. В принципе, во всех СУБД существует возможность ограничить выполнение определенных действий определенным временем и кругом пользователей. Общая модель безопасности СУБД представлена на рис. 12.2

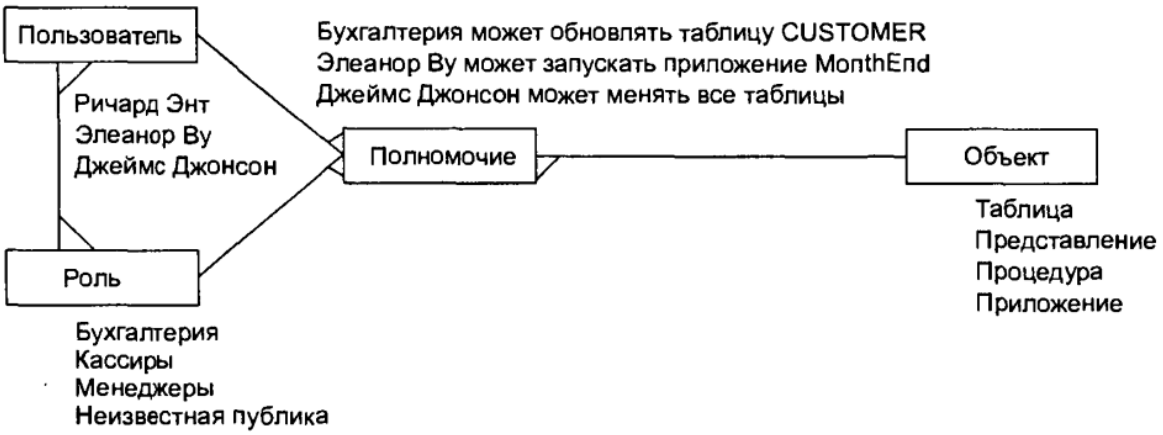


Рис. 12.2. Модель безопасности СУБД.

Пользователю может быть назначена одна или несколько ролей, а роль может принадлежать одному или многим пользователям. Как пользователи, так и роли могут иметь много различных полномочий. С каждым объектом (в широком смысле этого слова) связаны определенные полномочия. Каждое полномочие относится к одному пользователю или группе и одному объекту.

Когда пользователь входит в систему базы данных, СУБД ограничивает его действия полномочиями, определенными индивидуально для данного пользователя, а также для роли, назначенной данному пользователю. Определить, является ли пользователь тем, за кого он себя выдает, — задача, вообще говоря, сложная. Во всех коммерческих СУБД используется тот или иной вариант парольной защиты, даже при том, что такой метод обеспечения безопасности можно легко обойти, если пользователи небрежно относятся к своим личным идентификаторам.

Язык управления данными (DCL) используется для управления правами доступа к данным и выполнением процедур в многопользовательской среде. Более точно его можно назвать "язык управления доступом". Он состоит из двух основных команд:

- **GRANT** – дать права;
- **REVOKE** – забрать права.

Как правило, пользователи самостоятельно вводят свое имя и пароль, а в некоторых приложениях имя пользователя и пароль вводятся системой от лица пользователя. Например, Windows 2000

может непосредственно передать имя пользователя и пароль для входа в систему СУБД SQL Server. В других случаях имя пользователя и пароль предоставляются прикладной программой.

В интернет - приложениях обычно определяется группа под названием вроде «Unknown Public» (неизвестная публика), и в эту группу записываются анонимные пользователи, входящие в систему. Таким образом, компании типа Dell Computer избегают необходимости вводить каждого пользователя в систему под собственным именем и паролем.

Модели систем безопасности, используемые во множестве СУБД, изображены на рис. 12.3. В соответствии с этой моделью, у каждого пользователя есть профиль, который указывает, какие системные ресурсы может задействовать данный пользователь. Пользователю может быть назначено много ролей, и одна и та же роль может быть назначена многим пользователям. Каждый пользователь или роль имеет множество привилегий. Есть два вида привилегий. Объектные привилегии определяют действия, которые могут быть предприняты по отношению к объектам базы данных – таблицам, представлениям и индексам. Системные привилегии определяют возможные действия с использованием команд.

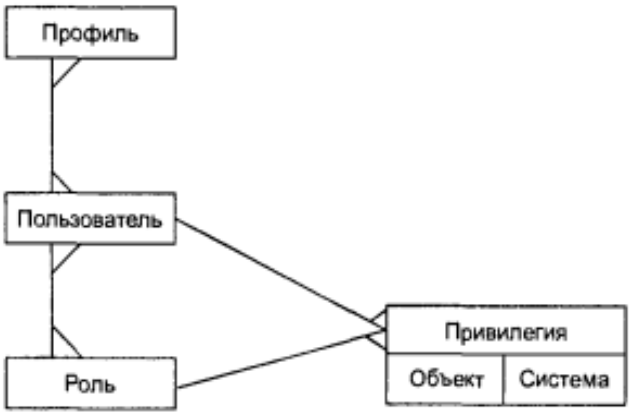


Рис. 12.3. Модель безопасности.

Обеспечение безопасности средствами приложения.

Хотя такие СУБД, как Oracle или SQL Server, предоставляют значительные возможности по обеспечению безопасности, эти возможности по своей натуре являются весьма общими. Если приложению требуются специфические меры безопасности, например «пользователь не может просматривать строки таблицы или соединения таблиц, имя сотрудника в которых отличается от

его собственного», то СУБД будет бессильна это сделать. В таких случаях система безопасности должна быть дополнена функциями, реализованными в самом приложении.

Можно организовать хранение информации, относящейся к безопасности, в специализированной базе данных, доступ к которой могут осуществлять приложение или хранимые процедуры и триггеры.

Есть много других способов расширения возможностей СУБД по обеспечению безопасности за счет приложения. Однако в первую очередь следует задействовать систему безопасности СУБД. Только если ее функций оказывается недостаточно, стоит дополнять их за счет прикладных программ. Чем ниже уровень, на котором идет обеспечение безопасности, тем меньше возможностей для несанкционированного вторжения. Кроме того, система безопасности СУБД работает быстрее, ее использование приводит к меньшим затратам и, возможно, дает лучшие результаты, чем разработка собственной системы.

Шифрование.

Шифрование – преобразование данных с использованием специального алгоритма, в результате чего данные становятся недоступными для чтения любой программой, не имеющей ключа дешифрования,

Если в системе с базой данных содержится весьма важная конфиденциальная информация, то имеет смысл зашифровать ее с целью предупреждения возможной угрозы несанкционированного доступа с внешней стороны (по отношению к СУБД). Некоторые СУБД включают средства шифрования, предназначенные для использования в подобных целях. Подпрограммы таких СУБД обеспечивают санкционированный доступ к данным (после их дешифрования), хотя это будет связано с некоторым снижением производительности, вызванным необходимостью двойного преобразования. Шифрование также может использоваться для защиты данных при их передаче по линиям связи. Существует множество различных технологий шифрования данных с целью сокрытия передаваемой информации, причем одни из них называют необратимыми, а другие – обратимыми. Необратимые методы, как и следует из их названия, не позволяют установить исходные данные, хотя последние могут использоваться для сбора достоверной статистической информации. Обратимые технологии

используются чаще. Для организации защищенной передачи данных по незащищенным сетям должны использоваться системы шифрования, включающие следующие компоненты:

- ключ шифрования, предназначенный для шифрования исходных данных (обычного текста);
- алгоритм шифрования, который описывает, как с помощью ключа шифрвания преобразовать обычный текст в зашифрованный;
- ключ дешифрования, предназначенный для дешифрования зашифрованного текста;
- алгоритм дешифрования, который описывает, как с помощью ключа дешифрования преобразовать зашифрованный текст в обычный исходный.

Некоторые системы шифрования, называемые симметричными, используют один и тот же ключ как для шифрования, так и для дешифрования, при этом предполагается наличие защищенных линий связи, предназначенных для обмена ключами. Однако большинство пользователей не имеют доступа к защищенным линиям связи, поэтому для получения надежной защиты длина ключа должна быть не меньше длины самого сообщения. Тем не менее, большинство эксплуатируемых систем построено на использовании ключей, которые короче самих сообщений.

Другой тип систем шифрования предусматривает использование различных ключей для шифровки и дешифровки сообщений – подобные системы принято называть асимметричными. Примером такой системы является система с открытым ключом, предусматривающая использование двух ключей, один из которых является открытым, а другой хранится в секрете. Алгоритм шифрования также может быть открытым, поэтому любой пользователь, желающий направить владельцу ключей зашифрованное сообщение, может использовать его открытый ключ и соответствующий алгоритм шифрования. Однако дешифровать данное сообщение сможет только тот, кто имеет парный закрытый ключ шифрования. Системы шифрования с открытым ключом могут также использоваться для отправки вместе с сообщением "цифровой подписи", подтверждающей, что

данное сообщение было действительно отправлено владельцем открытого ключа.

12.4. Управление СУБД

Кроме управления работой с данными и структурой базы данных, администратор обязан управлять самой СУБД. Он должен собирать и анализировать статистику производительности системы и идентифицировать области, чреватые возникновением проблем. Вспомните, что база данных обслуживает множество пользовательских групп. Администратор базы данных должен рассматривать все жалобы на медленный отклик системы, неточность данных, сложность в использовании и т. п. Если требуются какие-то изменения, администратор должен составить план этих изменений и реализовать их.

Администратор должен периодически осуществлять мониторинг пользовательской активности при работе с базой данных. Отчеты могут содержать информацию о том, какие пользователи были наиболее активны, какие файлы и, возможно, элементы данных использовались, какие методы доступа были при этом задействованы. Также в отчетах может присутствовать информация о типах и частоте возникновения ошибок. Администратор анализирует эту информацию с целью определить, нужны ли какие-либо изменения в структуре базы данных для повышения производительности или упрощения работы пользователей. Если такие изменения необходимы, администратор обеспечивает их реализацию.

Администратор базы данных должен анализировать текущую статистику производительности базы данных и активности пользователей. При обнаружении каких-либо проблем с производительностью (в ходе анализа отчета или по жалобе пользователя) администратор должен определить, требуется ли модификация структуры базы данных или системы. Примерами возможных структурных модификаций являются введение новых ключей, чистка данных, удаление ключей и установление новых связей между объектами.

Когда производитель используемой СУБД объявляет о новых возможностях продукта, администратор базы данных должен рассмотреть их в свете потребностей сообщества пользователей. Если он решит внедрить эти новые возможности, разработчиков

следует уведомить об этом и обучить их использованию. Соответственно, помимо управления структурой базы данных администратор должен осуществлять управление и контроль над изменениями в СУБД.

В обязанности администратора базы данных могут входить и другие изменения в системе; какие именно – это зависит от используемой СУБД и другого программного и аппаратного обеспечения. Например, изменения в операционной системе или web-сервере могут повлечь за собой необходимость модификации некоторых возможностей, функций или параметров СУБД. Поэтому администратор базы данных должен также настраивать СУБД для совместной работы с другим используемым программным обеспечением.

Первоначальный выбор параметров СУБД (таких как уровень изоляции транзакций) происходит в момент, когда еще мало известно о том, как система будет работать в конкретном пользовательском окружении. Следовательно, опыт работы с системой и результаты анализа ее производительности могут указать на необходимость изменений. Даже если производительность кажется приемлемой, возможно, администратор базы данных захочет исследовать, как изменение ее параметров будет влиять на производительность. Этот процесс называется настройкой (tuning), или оптимизацией (optimizing), системы. В списке приведен перечень обязанностей администратора базы данных по управлению СУБД.

- Подготовка отчетов о работе базы данных.
- Рассмотрение жалоб пользователей на работу системы.
- Оценка необходимости изменений в структуре базы данных или приложений.
- Модификация структуры базы данных.
- Оценка и внедрение новых возможностей и функций СУБД.
- Настройка СУБД.

Поддержание репозитория данных

Представьте себе большое и активное интернет-приложение базы данных, подобное тем, что используются компаниями, занимающимися электронной коммерцией, – например, продажей музыки в сети Интернет. Информацию для такой системы могут

предоставлять несколько различных баз данных, десятки web-страниц и сотни, если не тысячи, пользователей.

Предположим, что компания, использующая это приложение, желает расширить ассортимент предлагаемых товаров, включив в него спортивные товары. Высшее руководство компании может попросить администратора базы данных оценить время и другие ресурсы, необходимые для того, чтобы настроить базу данных на поддержку новой линии продуктов.

Чтобы администратор базы данных смог ответить на этот запрос, ему потребуются подробные метаданные, описывающие базу данных, ее приложения и компоненты этих приложений, пользователей и их права и привилегии, а также другие элементы системы. Часть этих метаданных хранится в системных таблицах базы данных, но одной только этой части будет недостаточно, чтобы ответить на вопросы, задаваемые высшим руководством. Администратору базы данных требуются дополнительные данные об объектах COM и ActiveX, сценарных процедурах и функциях, ASP-страницах, таблицах стилей, определениях типов документов и т. п. Кроме того, хотя механизмы безопасности СУБД обеспечивают документирование данных о пользователях, группах и привилегиях, они делают это в высокоструктурированной и зачастую неудобной форме.

По этим причинам многие организации разрабатывают и поддерживают репозитории данных (data repositories), которые представляют собой коллекции метаданных, описывающих базу данных, ее приложения, web-страницы, пользователей и другие компоненты приложений. Репозиторий может быть виртуальным в том смысле, что составляющие его метаданные собраны из различных источников, в числе которых может быть СУБД, программное обеспечение контроля версий, библиотеки кода, средства генерации и редактирования web-страниц и т. д. Репозиторий данных также может быть интегрированным продуктом, который поставляется производителем CASE-средств или другими компаниями, например Microsoft или Oracle.

В любом из этих случаев администратор базы данных должен задуматься о построении репозитория данных задолго до того, как высшее руководство начнет задавать вопросы. На самом деле репозиторий должен строиться при разработке системы, и его следует рассматривать как важную составляющую часть системы.

В противном случае администратор базы данных будет «вечно догоняющим», пытаясь поддерживать существующие приложения, адаптировать их к новым потребностям и каким-то образом собирать метаданные для репозитория.

Лучший вид репозиториев – это активные репозитории (active repositories). Они формируются в процессе разработки системы за счет того, что при создании компонентов системы автоматически создаются метаданные. Менее желательным, но все же эффективным вариантом являются пассивные репозитории (passive repositories): заполнение таких репозиториев и создание метаданных для них происходит вручную.

Интернет создал невиданные возможности для расширения клиентской базы и увеличения продаж и рентабельности в бизнесе. Базы данных и приложения, используемые компаниями в своей работе, составляют ключевой элемент этого успеха. К сожалению, найдутся такие организации, росту которых будет препятствовать неспособность наращивать свои приложения или адаптировать их к меняющимся потребностям. Зачастую построить новую систему оказывается проще, чем адаптировать уже существующую; определенно, построение новой системы, которая интегрировалась бы со старой, в то же время заменяя ее, может быть весьма сложной задачей.

Контрольные вопросы:

1. Какие способы восстановления данных существуют в СУБД?
2. Как обеспечивается управление конфигурацией БД?
3. Как обеспечивается управление безопасностью БД?
4. Каким способом администратор участвует в процессе поддержки базы данных?
5. Как обеспечивается шифрование?
6. Что такое репозиторий?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p.
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p.

Глава XIII. Интерфейс ODBC

13.1. Технологии доступа к данным.

Одной из основных задач, необходимых решить проектировщику или программисту ИС - это выбор технологии доступа к БД. Выбор технологии доступа к данным является одной из стратегических задач, от решения которой зависит как производительность будущей системы и способность реализовывать дополнительные функции, так и совместимость ее с другими программными платформами и технологиями, переносимость с одной платформы на другую.

Существует несколько способов решения задачи обеспечения доступа к данным.

Во многих современных СУБД имеются библиотеки, содержащие специальный интерфейс прикладного программирования (API), который представляет собой набор функций для манипулирования данными. В СУБД для настольных систем API осуществляет лишь чтение и запись данных в БД. В СУБД типа клиент/сервер API инициирует отправку по сети запроса к серверу и получение результатов или кодов ошибок для дальнейшей их обработки клиентским приложением.

Один из способов доступа к данным заключается в непосредственном использовании API, однако это означает полную зависимость приложения от используемой СУБД. Таким образом, необходим некий универсальный механизм доступа к данным, обеспечивающий для клиентского приложения стандартный набор общих функций, классов, сервисов, служб, необходимых для работы с различными СУБД. Эти стандартные функции (классы или сервисы) должны размещаться в библиотеках, именуемых драйверами или провайдерами баз данных (data base drivers (providers)). Каждая такая библиотека реализует набор стандартных функций, классов или сервисов, используя обращения API к конкретной системе управления базами данных.

13.2. Интерфейс Open Database Connectivity

Интерфейс ODBC (Open Database Connectivity) был разработан фирмой Microsoft как открытый интерфейс доступа к базам данных. Он предоставляет унифицированные средства

взаимодействия прикладной программы, называемой клиентом (или приложением-клиентом), с сервером - базой данных.

В основу интерфейса ODBC были положены спецификация CLI-интерфейса (Call-Level Interface), разработанная X/Open, и ISO/IEC для API баз данных, а также язык SQL (Structured Query Language) как стандарт языка доступа к базам данных.

Интерфейс ODBC проектировался для поддержки максимальной интероперабельности приложений, которая обеспечивает унифицированный доступ любого приложения, использующего ODBC, к различным источникам данных. Так, если приложение, соответствующее стандарту ODBC и SQL, первоначально разрабатывалось для работы с базой данных Microsoft Access, а затем таблицы этой базы были перенесены в базу данных Microsoft SQL Server или базу данных Oracle, то приложение сможет и дальше обрабатывать эти данные без внесения дополнительных изменений.

Для взаимодействия с базой данных приложение-клиент вызывает функции интерфейса ODBC, которые реализованы в специальных модулях, называемых ODBC-драйверами. Как правило, ODBC-драйверы - это DLL-библиотеки, при этом одна DLL-библиотека может поддерживать несколько ODBC-драйверов. При установке на компьютер любого SQL-сервера (базы данных, поддерживающей один из стандартов языка SQL, например, SQL-92) автоматически выполняется регистрация в реестре Windows и соответствующего ODBC-драйвера.

Архитектура ODBC

Архитектура ODBC представлена четырьмя компонентами (рис. 13.1):

- Приложение-клиент, выполняющее вызов функций ODBC.
- Менеджер драйверов, загружающий и освобождающий ODBC-драйверы, которые требуются для приложений-клиентов. Менеджер драйверов обрабатывает вызовы ODBC-функций или передает их драйверу.
- ODBC-драйвер, обрабатывающий вызовы SQL-функций, передавая SQL-серверу выполняемый SQL-оператор, а приложению-клиенту - результат выполнения вызванной функции.

- Источник данных, определяемый как конкретная локальная или удаленная база данных.

Рис. 13.1. Архитектура ODBC.

Основное назначение менеджера драйверов - загрузка драйвера, соответствующего подключаемому источнику данных, и инкапсуляция взаимодействия с различными типами источников данных посредством применения различных ODBC-драйверов.

ODBC-драйверы, принимая вызовы функций, взаимодействуют с приложением-клиентом, выполняя следующие задачи:

- управление коммуникационными протоколами между приложением-клиентом и источником данных;
- управление запросами к СУБД;
- выполнение передачи данных от приложения-клиента в СУБД и из базы данных в приложение-клиент;
- возвращение приложению-клиенту стандартной информации о выполненном вызове ODBC-функции в виде кода возврата;
- поддерживает работу с курсорами и управляет транзакциями.

Приложение-клиент одновременно может устанавливать соединения с несколькими различными источниками данных, используя разные ODBC-драйверы, а также несколько соединений с одним и тем же источником данных, используя один и тот же ODBC-драйвер.

Функции ODBC API.

Все функции ODBC API условно можно разделить на четыре группы:

1. основные функции ODBC, обеспечивающие взаимодействие с источником данных;
2. функции установки (setup DLL);
3. функции инсталляции (installer DLL) ODBC и источников данных;
4. функции преобразования данных (translation DLL).

Объявления всех функций и используемых ими типов данных содержатся в заголовочных файлах. Группа основных функций ODBC API разбита на три уровня:

1. функции ядра ODBC;
2. функции 1 уровня;
3. функции 2 уровня.

Интерфейс ODBC API реализован как набор расслоенных DLL-функций для Windows. Динамическая библиотека ODBC.DLL – это основная библиотека управления драйверами ODBC, которая содержит функции вызовов специализированных драйверов для разных поддерживаемых системой баз данных. Каждый драйвер совместим со своим уровнем и относится к одной из двух категорий: одноуровневые или многоуровневые драйверы.

Одноуровневые драйверы предназначены для использования при работе с теми источниками данных, которые не могут быть прямо обработаны с использованием ANSI SQL. Обычно это локальные базы данных на персональных компьютерах, такие как dBase, Paradox, FoxPro и Excel. Драйверы, соответствующие этим базам данных, производят компиляцию ANSI SQL в наборы инструкций более низкого уровня, которые непосредственно обрабатывают составляющие базу данных файлы.

Многоуровневые драйверы используют сервер СУБД для обработки SQL-предложений и предназначены для работы в среде клиент-сервер. Помимо обработки ANSI SQL, они также могут поддерживать и собственные конструкции конкретной РСУБД, поскольку ODBC может без трансляции передавать SQL-операторы источникам данных (механизм "passthrough"). Драйверы ODBC для баз данных, поддерживаемым в технологии клиент-сервер реализованы для практически для всех промышленных серверов БД.

Существует 4 важных этапа (шага) процедуры запроса данных через ODBC API.

- *Шаг 1* - установление соединения. Первый шаг состоит в размещении указателей (handle) среды ODBC, которые выделяют оперативную память под ODBC драйверы и библиотеки. Затем происходит выделение памяти для указателей соединения, и соединение устанавливается.
- *Шаг 2* - выполнение оператора SQL. Выделяется указатель оператора, локальные переменные связываются со столбцами в SQL-выражении (это необязательное действие), и выражение представляется главному ODBC-драйверу для обработки.
- *Шаг 3* - извлечение данных. Перед извлечением данных возвращается информация о результирующем наборе, в частности, число столбцов в наборе. Исходя из этого числа, результирующий набор помещается в буфер записей, выполняется цикл его просмотра и содержимое каждого столбца помещается в соответствующую локальную переменную. Этот шаг необязателен, если используется связывание столбцов с локальными переменными.
- *Шаг 4* - освобождение ресурсов. После того, как данные получены, ресурсы освобождаются путем вызова функций освобождения указателей оператора, соединения и среды. Указатели оператора и соединения могут быть использованы в процессе обработки.

Технология ODBC разрабатывалась как общий, независимый от источников данных, способ доступа к данным. Применение технологии должно было также обеспечить переносимость приложений в среду различных баз данных без потребности переработки самих приложений. В этом смысле технология ODBC уже стала промышленным стандартом, ее поддерживают практически все производители СУБД и средств разработки.

Однако универсальность стоит дорого. Если при разработке приложений одним из основных критериев является переносимость на различные СУБД, то использование ODBC является оправданным. Для увеличения производительности и эффективности приложения активно применяют специфические для данной СУБД расширения языка SQL, используют хранимые на сервере процедуры и функции. В этом случае теряется роль

ODBC как общего метода доступа к данным. Тем более, что для разных СУБД драйверы ODBC поддерживают разные уровни совместимости. Поэтому многие производители средств разработки, помимо поддержки ODBC, поставляют "прямые" драйверы к основным СУБД.

Объектная модель OLE DB.

OLE DB представляет собой набор COM-интерфейсов (Component Object Model), которые предоставляют приложению-клиенту унифицированный доступ к различным источникам данных.

Можно сказать, что OLE DB - это метод доступа к любым данным через стандартные COM-интерфейсы, вне зависимости от типа данных и места их расположения. В качестве данных могут выступать базы данных, простые документы, таблицы Excel и любые другие источники данных. В отличие от доступа, предоставляемого посредством драйверов OBDC, OLE DB позволяет реализовывать доступ к источникам данных, как с применением языка SQL (к SQL-серверам), так и к любым другим произвольным источникам данных.

Средства, предоставляющие доступ к источнику данных с использованием технологии OLE DB, называются OLE DB провайдерами. Программы-клиенты, использующие для доступа OLE DB провайдеры, называются потребителями данных.

В том случае, если существует только ODBC-драйвер для доступа к конкретному источнику данных, то для применения технологии OLE DB можно использовать OLE DB провайдер, предназначенный для доступа к ODBC-источнику данных.

Так как архитектура OLE DB основана на COM, то механизм создания результирующих наборов состоит из последовательностей шагов типа: 1. создание объекта -> 2. запрос указателя на интерфейс созданного объекта -> 3. вызов метода интерфейса.

Аналогично комплексу действий, который производится после создания результирующего набора при применении технологии ODBC - выполнению связывания, в технологии OLE DB используется механизм асессоров. Асессоры описывают, каким образом данные записываются в область памяти потребителя данных, устанавливая адресное соответствие между областью памяти в буфере потребителя данных и столбцами

данных в результирующем наборе. Иногда такой набор связей называют картой столбцов (column map).

Спецификация OLE DB описывает набор интерфейсов, реализуемых объектами OLE DB. Каждый объектный тип определен как набор интерфейсов. Спецификация OLE DB определяет набор интерфейсов базового уровня, которые должны реализовываться любыми OLE DB провайдерами.

В базовую модель OLE DB входят следующие объекты:

- объект Data Source (источник данных), используемый для соединения с источником данных и создания одного или нескольких сеансов. Этот объект управляет соединением, использует информацию о полномочиях и аутентификации пользователя;
- объект Session (сеанс) управляет взаимодействием с источником данных - выполняет запросы и создает результирующие наборы. Сеанс также может возвращать метаданные. В сеансе может создаваться одна или несколько команд;
- объект Rowset (результирующий набор) представляет собой данные, извлекаемые в результате выполнения команды или создаваемые в сеансе.

OLE DB представляет собой интерфейс системного уровня, обеспечивающий доступ к различным источникам данных – реляционным и нереляционным, содержащим текст, графические и географические данные, к файлам электронной почты, содержимому файловых систем и создаваемым пользователями бизнес - объектам. OLE DB определяет набор интерфейсов компонентной объектной модели (Component Object Model, COM), включающих в себя службы различных систем управления базами данных для обеспечения универсального доступа к данным. С помощью этих интерфейсов программисты могут создавать дополнительные сервисы баз данных.

Любой компонент программного обеспечения, который использует интерфейсы OLE DB, является потребителем OLE DB. Это может быть бизнес-приложение, инструментальное средство разработки программного обеспечения, например, Borland Delphi, сложные приложения или же объектная модель ActiveX Data Objects, использующая интерфейсы OLE DB. Потребители

используют либо те ActiveX Data Objects (ADO), которые являются интерфейсом прикладного уровня для обеспечения косвенного (indirect) доступа к данным с применением OLE DB, либо непосредственно OLE DB – для прямого доступа к данным с помощью провайдера OLE DB.

С точки зрения OLE DB, может быть два вида провайдеров: OLE DB - провайдеры данных и провайдеры сервисов (служб).

Провайдер данных (data provider) представляет собой компонент программного обеспечения, "владеющий" данными. Он находится между потребителем и непосредственным массивом данных. В OLE DB все провайдеры представляют данные в табличном формате (с которым мы уже знакомы по реляционным базам данных и электронным таблицам), в виде виртуальных таблиц. Провайдер данных выполняет следующие задачи.

- Принимает запросы, поступающие от потребителя, на доступ к данным.
- Выполняет выборку или обновление данных из массива данных.
- Возвращает эти данные потребителю.

Одним из примеров провайдера данных служит Microsoft Jet 4.0 OLE DB Provider. Он используется совместно с механизмом доступа к базам данных Microsoft Jet, применяемым для обработки информации в базах данных Microsoft Access, а также для доступа как к базам данных к информации, упорядоченной с помощью так называемого инсталлируемого индексно-последовательного метода доступа (Indexed Sequential Access Method, I-ISAM), который поддерживается в Jet. К таким данным относятся таблицы, хранимые в рабочих книгах Excel, почтовые файлы Outlook и Microsoft Exchange, таблицы dBase и Paradox, текстовые и HTML-файлы и т. д. Другим провайдером OLE DB является Microsoft OLE DB Provider for SQL Serwer, используемый для работы с базами данных Microsoft SQL Server 6.5, 7.0, 2000.

Провайдер сервисов (служб) реализует расширенные функциональные возможности, которые не поддерживаются обычными провайдерами данных, и сам не "владеет" данными. Этот провайдер, например, обеспечивает сортировку, фильтрацию, управление транзакциями, обработку SQL-запросов, функции указателя (курсора) и т. д. Провайдер сервисов может напрямую

работать с массивами данных или же через соответствующий провайдер данных; в этом случае он выступает в роли потребителя и провайдера.

Например, такие провайдеры сервисов, как Microsoft Cursor Service for OLE DB и Microsoft Data Shaping Service for OLE DB могут интегрироваться с базовыми провайдерами данных OLE DB для расширения их функциональных возможностей. Такие провайдеры сервисов, как Microsoft Cursor Service for OLE DB и Microsoft Data Shaping Service for OLE DB могут интегрироваться с базовыми провайдерами данных OLE DB для расширения их функциональных возможностей.

13.3. ADO

OLE DB обеспечивает связывание для программистов на C и C++, а также программистов, использующих языки с C-подобными вызовами функций. Такие языки, как VB и VBScript, не поддерживают тип данных «указатель» (адресных переменных). Следовательно, они не могут использовать связывание в стиле C и прямое обращение к OLE DB.

Вероятно, для большей путаницы разработчики Microsoft ввели еще одну объектную модель доступа к данным: ADO. ADO работает с объектами DAO и RDO, а также поддерживает более простые модели, чем DAO и RDO (хотя с избыточной функциональностью, так что можно выполнить операцию несколькими способами). Объектная иерархия в ADO более однородная, чем в DAO. ADO содержит несколько встроенных объектов, которые упрощают доступ к данным из информационных хранилищ.

На Рисунке 13.2 показано несколько способов, с помощью которых приложение связывается с базой данных. Например, VB-программист может использовать ADO для соединения приложения с провайдером OLE DB. Если база данных не поддерживает OLE DB, приложение может задействовать ODBC. Программист на Visual C++ может применять ADO или соединяться напрямую через OLE DB.

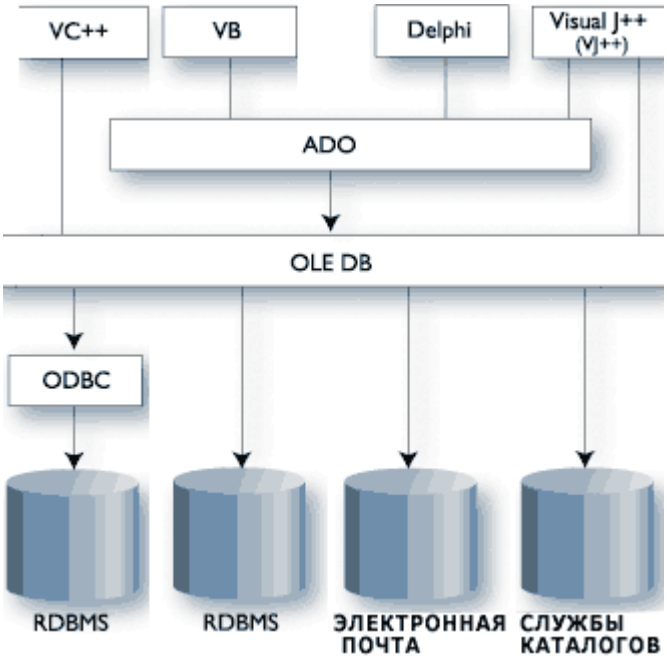


Рис.13.2. Различие маршрутов приложений в ADO.

Объект Recordset представляет собой набор записей (таблицу) и поддерживает типы курсоров adOpenForwardOnly, adOpenKeyset, adOpenDynamic и adOpenStatic. Курсор может быть как на стороне сервера (по умолчанию), так и на стороне клиента.

Для доступа к записи ADO требуется просканировать набор строк последовательно. Для доступа к нескольким таблицам необходимо выполнить запрос на объединение JOIN, чтобы получить результат в виде набора строк. Хотя объект Recordset поддерживает доступ к данным без соединения с ними, ADO изначально был спроектирован для данных, с которыми установлено соединение. Такой метод доступа заставляет хранить важные ресурсы на стороне сервера. Вдобавок для передачи набора строк следует использовать метод упорядочивания, названный COM marshalling. COM marshalling — это процесс преобразования типов данных, который, естественно, занимает полезные ресурсы системы.

Начиная с ADO 2.1, Microsoft добавляет поддержку XML в объектную модель ADO, что позволяет хранить набор строк Recordset как XML-документ. Однако только при появлении ADO 2.5 ряд ограничений XML, который сохранялся в версии ADO 2.1 (например, жесткая иерархия объектов Recordset), был устранен. Хотя ADO может преобразовать документ XML в набор Recordset, он в состоянии читать только документы в собственной схеме, известной как Advanced Data TableGram (ADTG).

В поисках механизма доступа к несвязанным данным Microsoft расширяет ADO и вводит службу Remote Data Services (RDS). RDS создана после ADO и разрешает передачу объекта Recordset клиенту (например, в Web-браузер) при отсутствии активного соединения. Однако RDS, как и ADO, использует упорядочивание COM marshaling для передачи набора строк от сервера клиенту.

Эра .NET

Когда Microsoft начала разрабатывать .NET Framework, она имела хорошую возможность пересмотреть модель доступа к данным. Решив не продолжать разработку технологии ADO, специалисты Microsoft приступили к созданию новой структуры доступа к данным, при этом сохранив акроним. Microsoft разрабатывает ADO.NET на базе уже зарекомендовавшей себя объектной технологии ADO. Но ADO.NET ориентируется на три важные возможности, которые не поддерживаются ADO: поддержка модели доступа к несвязанным данным, что является ключевым элементом для работы в Web; поддержка тесной интеграции с XML; интеграция с .NET Framework (например, совместимость с базовой библиотекой классов типичной системы).

Архитектура ADO.NET. На Рисунке 11.3 представлена архитектура ADO.NET. Объект Recordset, который выполняет так много функций в ADO, здесь отсутствует. Вместо него в ADO.NET предусмотрено несколько особых объектов, выполняющих специфические задачи. В Таблице 1 описаны три из них: DataAdapter, DataReader и DataSet.

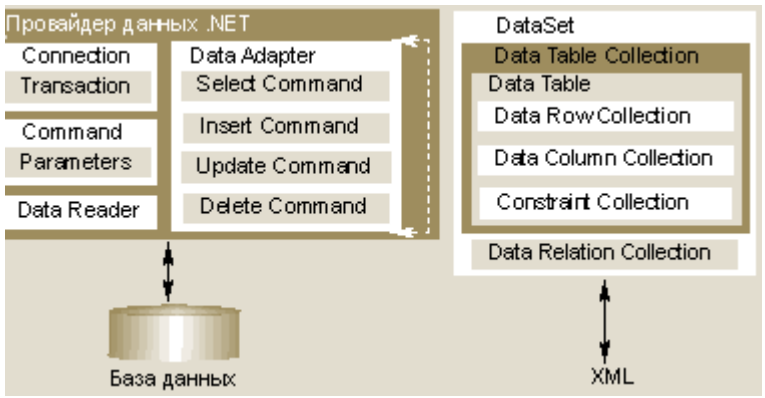


Рис. 11.3. Архитектура ADO.NET.

Поставщики данных .NET. Очень важный компонент ADO.NET, провайдер данных .NET, реализует интерфейсы

ADO.NET. В частности, он реализует объект DataReader так, что его могут использовать и приложение, и объект DataSet.

Поставщик данных .NET состоит из четырех основных компонентов: Connection — для связи с источником данных; Command выполняет команды над источником данных; DataReader читает данные из источника данных в однонаправленном режиме «только чтение», и DataAdapter, который читает данные из источника данных и использует их для заполнения объекта DataSet.

Visual Studio .NET содержит два поставщика данных .NET. Поставщик данных SQL Server .NET обеспечивает связь с SQL Server 7.0 и более поздними версиями. Этот метод доступа наиболее эффективен для SQL Server 7.0 и выше, потому что поставщик данных SQL Server .NET связывается напрямую с SQL Server через протокол Tabular Data Stream (TDS). Поставщик данных OLE DB .NET необходим для соединения с отличными от SQL Server базами данных, такими, как Oracle или IBM DB2. Этот поставщик данных использует OLE DB для соответствующих баз данных.

Во время написания статьи разработчики Microsoft реализовали третий тип поставщика данных .NET — ODBC .NET Data Provider — Release Candidate Beta. Его можно получить на сайте Microsoft *http://www.microsoft.com/data/download_odbcnetrc.htm*.

На Рисунке 11.4 показаны различные пути, по которым приложение может связываться с базой данных через ADO.NET. При выборе пути сначала определяется, какой поставщик данных .NET будет использоваться. Если это SQL Server 7.0 или более поздняя версия, то подключается поставщик данных SQL Server.NET. Если база данных SQL Server 6.5 или отличная от SQL Server (например, Oracle), понадобится поставщик данных OLE DB .NET. Заметим, что можно задействовать поставщик данных OLE DB .NET для баз данных SQL 7.0 и выше, но тогда потеряется выигрыш в производительности, который дает прямое подключение к SQL Server через протокол TDS. Однако в этом неспецифическом способе есть свой плюс — мобильность, т. е. можно менять базы данных без модификации кода.

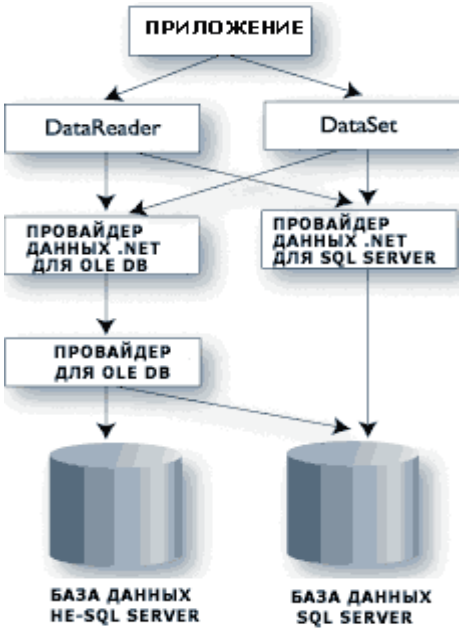


Рис. 11.4. Различие маршрутов в ADO.NET.

Далее необходимо определить, какую задачу требуется выполнить. Если надо просто прочитать и отобразить данные из источника данных, объекта Data Reader вполне достаточно. Но если предстоит манипулировать данными (например, редактировать или удалять), нужно использовать объект Data Set. Хотя задействовать этот объект следует только в случае необходимости, потому что он работает медленнее, чем Data Reader (Data Set использует Data Reader для заполнения таблиц).

JDBC.

JDBC очень проста — это API доступа к табличным данным. Внимательный читатель заметит, что речь не идет о базах данных, а о «табличных данных». Разница на первый взгляд может и не очень существенная, но на самом деле критичная. Например существуют JDBC драйверы доступа к текстовым файлам, таблицам Microsoft Excel, то есть к таким данным, которые ну никак нельзя отнести к базам данных: с поддержкой транзакций, индексов, отношений и проч.

JDBC, важно отметить, поддерживает работу не только с SQL совместимыми СУБД, но также и с практически любыми данными табличного типа. Хотя справедливо все же будет отметить, что мощь JDBC состоит конечно же не в том, чтобы прочитать данные из текстового файла — это как говорится «из пушки по воробьям» — без использования механизма транзакций, доступа к хранимым

процедурам и проч. «прелестям» присущим SQL базам данных JDBC не стал бы тем чем он сейчас является.

Технологию JDBC ни в коей мере нельзя называть технологией одной компании Sun Microsystems — родного отца Java. На данный момент технологию JDBC официально поддерживают следующие организации:

- Oracle
- DataDirect Technologies
- BEA
- Fujitsu
- MySQL
- INET Software
- Novell
- Borland
- Pointbase Inc.
- Macromedia
- SAP

В API JDBC интенсивно эксплуатируется концепция интерфейсов — набора методов, которые должны быть реализованы поставщиком того или иного сервиса, в данном случае поставщиком т.н. драйвера JDBC. С программной точки зрения драйвер JDBC есть нечто иное как реализация интерфейсов предусмотренных API JDBC. По способу реализации драйверы подразделяются на 4 типа:

Тип 1

К этому типу относятся драйверы реализованные поверх ODBC драйверов (что такое ODBC мы объяснять здесь не будем, если кто-то не знает отсылаем к первоисточнику). То есть фактически все вызовы API JDBC транслируются в вызовы ODBC, а дальше обработку вызова ведет API ODBC. Иногда еще 1-й тип драйверов называется "JDBC-ODBC bridge". Преимуществом драйверов этого типа, является то что все источники данных доступные с помощью ODBC становятся доступными Java приложению, недостатки такого драйвера: низкая скорость работы, трудности конфигурирования и невозможность поддержки всех возможностей API JDBC.

Тип 2

Ко второму типу относятся драйверы использующие программные части написанные на других языках (как правило на Си). Обычно в этом случае для доступа к базе данных используются библиотеки разработанные производителем, а для их вызова используется JNI — Java интерфейс вызова нативных функций. Примером такого драйвера является т.н. «толстый» OCI-JDBC драйвер для Oracle. Такие драйверы обычно очень быстрые, но опять же так же как и в случае JDBC-ODBC драйверов требуют установки специального ПО на клиентской машине. Для OCI-JDBC драйвера Oracle например требуется установка клиента SQL*NET.

Тип 3

В отличие от предыдущих типов драйверов данный тип драйвера полностью реализуется на Java, но при этом вызовы JDBC транслируются в сетевой протокол (RMI, HTTP и т.д.), который далее транслируется в специфичный протокол базы данных. В чем-то этот драйвер схож с драйверами JDBC-ODBC, отличие в том, что реализуется полностью на Java, за счет чего отсутствует необходимость в установке клиентского ПО.

Тип 4

Также как и драйверы 3-го типа реализуется полностью на Java, но вызовы реализуются напрямую с использованием протокола базы данных, минуя сетевой протокол.

Необходимо здесь также отметить, что несмотря на то что производители драйверов часто декларируют тот или иной тип, полную совместимость и т.д. тем не менее в реальной жизни только около четверти драйверов имеют сертификат соответствия спецификации JDBC. Наиболее полную информацию о существующих драйверах JDBC можно получить посетив корневой ресурс JDBC: http://java.sun.com/products/jdbc/. Здесь помещен список драйверов JDBC с указанием их типа, производителя и наличия сертификатов: http://servlet.java.sun.com/products/jdbc/drivers. База содержит около 200 драйверов и снабжена удобной системой поиска и навигации.

JDBC API содержит два основных типа интерфейсов: первый — для разработчиков приложений и второй (более низкого уровня) — для разработчиков драйверов.

Соединение с базой данных описывается классом, реализующим интерфейс java.sql.Connection. Имея соединение с

базой данных, можно создавать объекты типа Statement, служащие для исполнения запросов к базе данных на языке SQL.

Существуют следующие виды типов Statement, различающихся по назначению:

- java.sql.Statement — Statement общего назначения;
- java.sql.PreparedStatement — Statement, служащий для выполнения запросов, содержащих подставляемые параметры (обозначаются символом '?' в теле запроса);
- java.sql.CallableStatement — Statement, предназначенный для вызова хранимых процедур.
- java.sql.ResultSet позволяет легко обрабатывать результаты запроса.

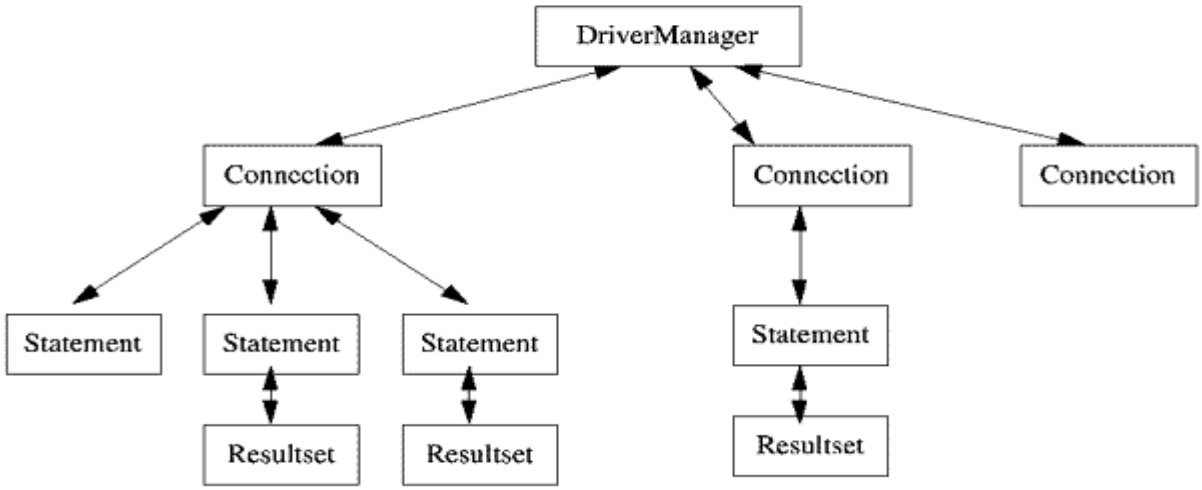


Рис 11.5. Основные интерфейсы JDBC.

Интерфейс выражения java.sql.Statement выступает в качестве предка для других двух важных интерфейсов: java.sql.PreparedStatement и java.sql.CallableStatement, первый из которых предназначен для выполнения прекомпилированных SQL-выражений, второй - для выполнения вызовов хранимых процедур. Соответственно Statement выполняет обычные (статические) SQL-запросы, а указанные два наследника работают с параметризированными SQL-выражениями.

Контрольные вопросы:

1. Перечислите основные технологии доступа к данным.
2. Что такое open database connectivity?
3. Что такое OLEDB?

4. Как работает OLEDB?
5. Что такое JDBC?
6. Как работает JDBC?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p.
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p.

Глава XIV. Добавление, обновление и удаление записей с использованием C++ и SQL

14.1. Класс TDataBase.

Объект типа TDataBase не является обязательным при работе с базами данных, однако он предоставляет ряд дополнительных возможностей по управлению соединением с базой данных. TDataBase служит для:

- Создания постоянного соединения с базой данных
- Определения собственного диалога при соединении с базой данных (опрос пароля)
- Создания локального псевдонима базы данных
- Изменения параметров при соединении
- Управления транзакциями

TDataBase является невидимым во время выполнения объектом. Он находится на странице "Data Access" Палитры Компонент. Для включения в проект TDataBase нужно "положить" его на главное окно вашей программы.

Создание постоянного соединения с базой данных

Если вы работаете с базой данных, то перед началом работы выполняется процедура соединения с этой базой. В процедуру соединения, кроме прочего, входит опрос имени и пароля пользователя (кроме случая работы с локальными таблицами Paradox и dBase через IDAPI). Если в программе не используется TDataBase, то процедура соединения выполняется при открытии первой таблицы из базы данных. Соединение с базой данных

обрывается, когда в программе закрывается последняя таблицы из этой базы (это происходит в том случае, если свойство *KeepConnections* объекта *Session* установлено в False, но об этом чуть позже). Теперь, если снова открыть таблицу, то процедура установки соединения повторится и это может быть достаточно неудобно для пользователя. Чтобы соединение не обрывалось даже в том случае, когда нет открытых таблиц данной базы, можно использовать компонент типа TDataBase. В свойстве AliasName укажите псевдоним базы данных, с которой работает программа; в свойстве DatabaseName - любое имя (псевдоним БД), на которое будут ссылаться таблицы вместо старого псевдонима базы. Свойство Connected установите в True - процедура соединения с базой будет выполняться при запуске программы. И, наконец, свойство KeepConnection нужно установить в True (см. рис.14.1).

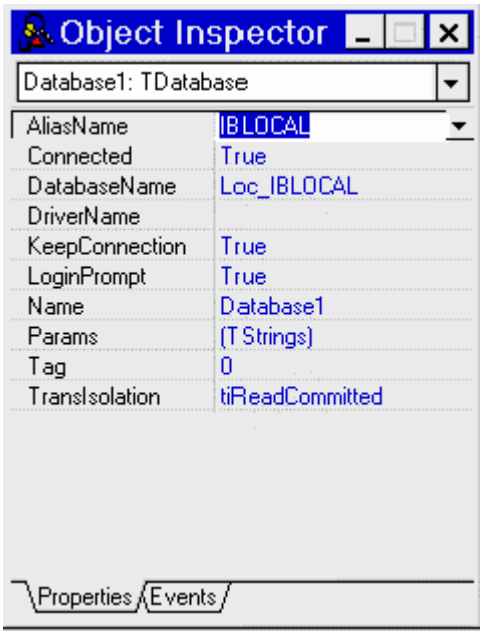


Рис.14.1. Свойства TDataBase в Инспекторе объектов

В нашем примере, после задания свойств DataBase1 нужно у всех таблиц, работающих с IBLOCAL в свойстве DatabaseName поставить Loc_IBLOCAL.

Определение собственного диалога при соединении с базой данных

По умолчанию при соединении с базой данных используется диалог опроса имени и пароля пользователя, показанный на рис.14.2

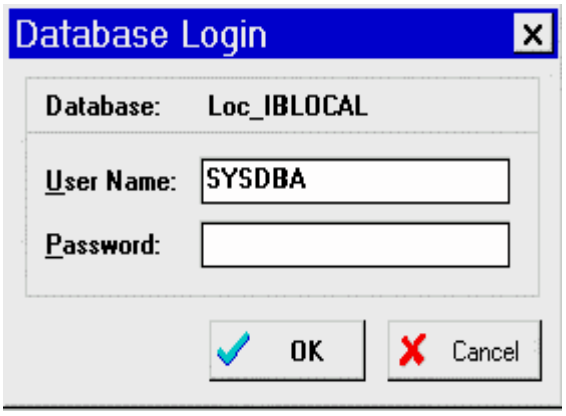


Рис.14.2. Диалог авторизации пользователя

При желании можно изменить внешний вид диалога или вообще его отменить. Для этого используются свойства и события класса TDataBase - *LoginPrompt, Params* и *OnLogin.*

Чтобы отключить опрос имени и пароля установите свойство Login Prompt в False. При этом в свойстве Params требуется в явном виде (во время дизайна либо во время выполнения) указать имя и пароль пользователя. Например, в программе можно написать (до момента соединения с базой, например в событии для Form1 OnCreate) :

```
DataBase1->LoginPrompt=False;
DataBase1->Params->Clear;
DataBase1->Params->Add('USER NAME=SYSDBA');
DataBase1->Params->Add('PASSWORD=masterkey');
DataBase1->Connected=True;
```

14.2. Управление транзакциями

TDataBase позволяет начать в БД транзакцию (метод StartTransaction), закончить (Commit) или откатить ее (RollBack). Кроме того, можно изменять уровень изоляции транзакций (свойство TransIsoltion).

TransIsolation	Oracle	Sybase and Microsoft SQL	Informix	InterBase
Dirty read	Read committed	Read committed	Dirty Read	Read committed
Read committed(Default)	Read committed	Read committed	Read committed	Read committed

Repeatable read	Repeatable read	Read committed	Repeatable Read	Repeatable Read

"Dirty Read" - внутри вашей текущей транзакции видны все изменения, сделанные другими транзакциями, даже если они еще не завершились по Commit. "Read Committed" - видны только "закоммитченные" изменения, внесенные в базу. "Repeatable Read" - внутри транзакции видны те данные, что были в базе на момент начала транзакции, даже если там на самом деле уже имеются изменения.

Объект Session.

Объект Session, имеющий тип TSession создается автоматически в программе, работающей с базами данных (в этом случае Delphi подключает в программу модуль DB). Вам не нужно заботиться о создании и уничтожении данного объекта, но его методы и свойства могут быть полезны в некоторых случаях. В этом компоненте содержится информация обо всех базах данных, с которыми работает программа. Ее можно найти в свойстве *DataBase*s. Со свойством *KeepConnections* данного объекта мы уже знакомы. Это свойство определяет, нужно ли сохранять соединение с базой, если в программе нет ни одной открытой таблицы из этой базы. *NetDir* - директория, в которой лежит общий сетевой файл PDOXUSRS.NET, необходимый BDE. *PrivateDir* - директория для хранения временных файлов.

С помощью методов объекта Session можно получить информацию о настройках BDE, например, список всех псевдонимов, драйверов баз данных или список всех таблиц в базе.

Еще одно важное назначение объекта Session - доступ с его помощью к таблицам Paradox, защищенным паролем. Прежде, чем открыть такую таблицу, требуется выполнить метод *AddPassword* :

```
Session->AddPassword('my_pass');
```

Удалить пароль можно с помощью метода RemovePassword или RemoveAllPasswords.

Класс TDataSet.

TDataSet класс - один из наиболее важных объектов БД. Чтобы начать работать с ним, Вы должны взглянуть на следующую иерархию:

```
TDataSet
        |
    TDBDataSet
        |
        |-- TTable
        |-- TQuery
        |-- TStoredProc
```

TDataSet содержит абстрактные методы там, где должно быть непосредственное управление данными. TDBDataSet знает, как обращаться с паролями и то, что нужно сделать, чтобы присоединить Вас к определенной таблице. TTable знает (т.е. уже все абстрактные методы переписаны), как обращаться с таблицей, ее индексами и т.д.

Как Вы увидите в далее, TQuery имеет определенные методы для обработки SQL запросов.

TDataSet - инструмент, который Вы будете использовать чтобы открыть таблицу, и перемещаться по ней. Конечно, Вы никогда не будете непосредственно создавать объект типа TDataSet. Вместо этого, Вы будете использовать TTable, TQuery или других потомков TDataSet (например, TQBE). Полное понимание работы системы, и точное значение TDataSet, будут становиться все более ясными по мере прочтения этой главы.

На наиболее фундаментальном уровне, Dataset это просто набор записей, как изображено на рис.14.3

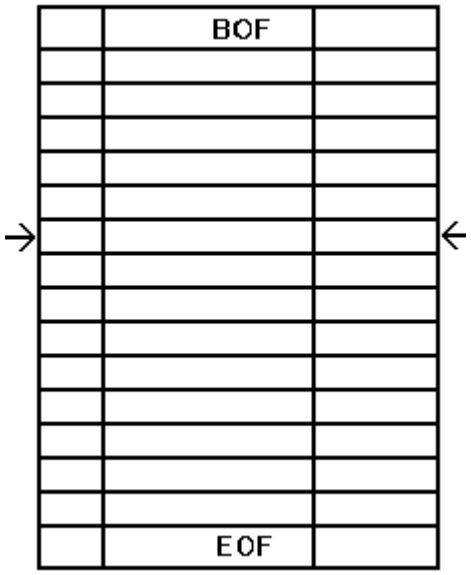


Рис.14.3. Любой dataset состоит из ряда записей (каждая содержит N полей) и указатель на текущую запись.

В большинстве случаев dataset будет иметь а прямое, один к одному, соответствие с физической таблицей, которая существует на диске. Однако, в других случаях Вы можете исполнять запрос или другое действие, возвращающие dataset, который содержит либо любое подмножество записей одной таблицы, либо объединение (join) между несколькими таблицами. В тексте будут иногда использоваться термины DataSet и TTable как синонимы.

Обычно в программе используются объекты типа TTable или TQuery, поэтому в следующих нескольких главах будет предполагаться существование объекта типа TTable называемого Table1.

Итак, самое время начать исследование TDataSet. Как только Вы познакомитесь с его возможностями, Вы начнете понимать, какие методы использует Delphi для доступа к данным, хранящимся на диске в виде БД. Ключевой момент здесь - не забывать, что почти всякий раз, когда программист на Delphi открывает таблицу, он будет использовать TTable или TQuery, которые являются просто некоторой надстройкой над TDataSet.

14.3. Открытие и закрытие DataSet

Если Вы используете TTable для доступа к таблице, то при открытии данной таблицы заполняются некоторые свойства TTable (количество записей RecordCount, описание структуры таблицы и т.д.).

Прежде всего, Вы должны поместить во время дизайна на форму объект TTable и указать, с какой таблицей хотите работать. Для этого нужно заполнить в Инспекторе объектов свойства DatabaseName и TableName. В DatabaseName можно либо указать директорию, в которой лежат таблицы в формате dBase или Paradox (например, C:\DELPHI\DEMOS\DATA), либо выбрать из списка псевдоним базы данных (DBDEMOS). Псевдоним базы данных (Alias) определяется в утилите Database Engine Configuration. Теперь, если свойство Active установить в True, то при запуске приложения таблица будет открываться автоматически.

Имеются два различных способа открыть таблицу во время выполнения программы. Вы можете написать следующую строку кода:

```
Table1->Open();
```

Или, если Вы предпочитаете, то можете установить свойство Active равное True:

```
Table1->Active = True;
```

Нет никакого различия между результатом производимым этими двумя операциями. Метод Open, однако, сам заканчивается установкой свойства Active в True, так что может быть даже чуть более эффективно использовать свойство Active напрямую.

Также, как имеются два способа открыть а таблицу, так и есть два способа закрыть ее. Самый простой способ просто вызывать Close:

```
Table1->Close();
```

Или, если Вы желаете, Вы можете написать:

```
Table1->Active = False;
```

Еще раз повторим, что нет никакой существенной разницы между двумя этими способами. Вы должны только помнить, что Open и Close это методы (процедуры), а Active - свойство.

Навигация (Перемещение по записям).

После открытия а таблицы, следующим шагом Вы должны узнать? как перемещаться по записям внутри него.

Следующий обширный набор методов и свойства TDataSet обеспечивает все, что Вам нужно для доступа к любой конкретной записи внутри таблицы:

- Вызов `Table1->First` перемещает Вас к первой записи в таблице.
- `Table1->Last` перемещает Вас к последней записи.
- `Table1->Next` перемещает Вас на одну запись вперед.
- `Table1->Prior` перемещает Вас на одну запись Назад.
- Вы можете проверять свойства `BOF` или `EOF`, чтобы понять, находитесь ли Вы в начале или в конце таблицы.
- Процедура `MoveBy` перемещает Вас на N записей вперед или назад в таблице. Нет никакого функционального различия между запросом `Table1->Next` и вызовом `Table1->MoveBy(1)`. Аналогично, вызов `Table1-`

`>Prior` имеет тот же самый результат, что и вызов `Table1->MoveBy(-1).`

Чтобы начать использовать эти навигационные методы, Вы должны поместить TTable, TDataSource и TDBGrid на форму, также, как Вы делали это в предыдущем уроке. Присоедините DBGrid1 к DataSource1, и DataSource1 к Table1. Затем установите свойства таблицы:

- в DatabaseName имя подкаталога, где находятся демонстрационные таблицы (или псевдоним DBDEMOS);
- в TableName установите имя таблицы CUSTOMER.

Если Вы запустили программу, которая содержит видимый элемент TDBGrid, то увидите, что можно перемещаться по записям таблицы с помощью полос прокрутки (scrollbar) на нижней и правой сторонах DBGrid.

14.4. Основные понятия о TQuery

При использовании TTable, возможен доступ ко всему набору записей из одной таблицы. В отличие от TTable, TQuery позволяет произвольным образом (в рамках SQL) выбрать набор данных для работы с ним. Во многом, методика работы с объектом TQuery похожа на методику работы с TTable, однако есть свои особенности.

Вы может создать SQL запрос используя компонент TQuery следующим способом:

1. Назначите Псевдоним (Alias) DatabaseName.
2. Используйте свойство SQL чтобы ввести SQL запрос типа "Select * from Country".
3. Установите свойство Active в True

Если обращение идет к локальным данным, то вместо псевдонима можно указать полный путь к каталогу, где находятся таблицы.

Свойство SQL

Свойство SQL - вероятно, самая важная часть TQuery. Доступ к этому свойству происходит либо через Инспектор Объектов во время конструирования проекта (design time), или программно во время выполнения программы (run time).

Интересней, конечно, получить доступ к свойству SQL во время выполнения, чтобы динамически изменять запрос. Например, если требуется выполнить три SQL запроса, то не надо размещать три компонента TQuery на форме. Вместо этого можно разместить один и просто изменять свойство SQL три раза. Наиболее эффективный, простой и мощный способ - сделать это через параметризованные запросы, которые будут объяснены в следующей части. Однако, сначала исследуем основные особенности свойства SQL, а потом рассмотрим более сложные темы, типа запросов с параметрами.

Свойство SQL имеет тип TStrings, который означает что это ряд строк, сохраняемых в списке. Список действует также, как и массив, но, фактически, это специальный класс с собственными уникальными возможностями. В следующих нескольких абзацах будут рассмотрены наиболее часто используемые свойства.

При программном использовании TQuery, рекомендуется сначала закрыть текущий запрос и очистить список строк в свойстве SQL:

```
Query1->Close();
Query1->SQL.Clear();
```

Обратите внимание, что всегда можно "безопасно" вызвать Close. Даже в том случае, если запрос уже закрыт, исключительная ситуация генерироваться не будет.

Следующий шаг - добавление новых строк в запрос:

```
Query1->SQL->Add('Select * from Country');
Query1->SQL->Add('where Name = ''Argentina''');
```

Метод Add используется для добавления одной или нескольких строк к запросу SQL. Общий объем ограничен только количеством памяти на вашей машине.

Чтобы Delphi отработал запрос и возвратил курсор, содержащий результат в виде таблицы, можно вызвать метод:

```
Query1->Open();
```

Демонстрационная программа THREESQL показывает этот процесс (см Рис.14.4)

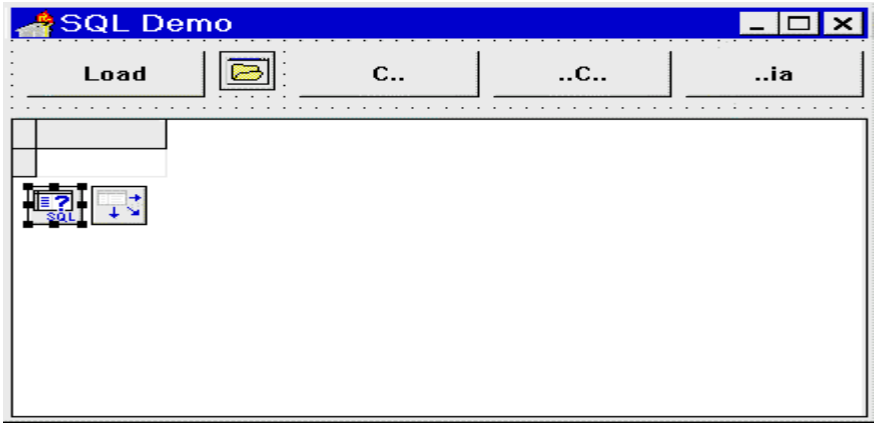


Рис.14.4. Программа THREESQL показывает, как сделать несколько запросов с помощью единственного объекта TQuery.

Программа THREESQL использует особенность локального SQL, который позволяет использовать шаблоны поиска без учета регистра (case insensitive). Например, следующий SQL запрос:

```
Select * form Country where Name like 'C%'
```

возвращает DataSet, содержащий все записи, где поле Name начинается с буквы 'C'. Следующий запрос позволит увидеть все страны, в названии которых встречается буква 'C':

```
Select * from Country where Name like '%C%';
```

Вот запрос, которое находит все страны, название которых заканчивается на 'ia':

```
Select * from Country where Name like '%ia';
```

Одна из полезных особенностей свойства SQL - это способность читать файлы, содержащие текст запроса непосредственно с диска. Эта особенность показана в программе THREESQL.

TQuery и Параметры.

C++ Builder позволяет составить "гибкую" форму запроса, называемую параметризованным запросом. Такие запросы позволяют подставить значение переменной вместо отдельных слов в выражениях "where" или "insert". Эта переменная может быть изменена практически в любое время. (Если используется локальный SQL, то можно сделать замену почти любого слова в утверждении SQL, но при этом та же самая возможность не поддерживается большинством серверов.)

Перед тем, как начать использовать параметризованные запросы, рассмотрим снова одно из простых вышеупомянутых предложений SQL:

```
Select * from Country where Name like 'C%'
```

Можно превратить это утверждение в параметризованный запрос заменив правую часть переменной NameStr:

```
select * from County where Name like :NameStr
```

В этом предложении SQL, NameStr не является предопределенной константой и может изменяться либо во время дизайна, либо во время выполнения. SQL parser (программа, которая разбирает текст запроса) понимает, что он имеет дело с параметром, а не константой потому, что параметру предшествует двоеточие ":NameStr". Это двоеточие сообщает Delphi о необходимости заменить переменную NameStr некоторой величиной, которая будет известна позже.

Обратите внимание, слово NameStr было выбрано абсолютно случайно. Использовать можно любое допустимое имя переменной, точно также, как выбирается идентификатор переменной в программе.

Есть два пути присвоить значение переменной в параметризованном запросе SQL. Один способ состоит в том, чтобы использовать свойство Params объекта TQuery. Второй - использовать свойство DataSource для получения информации из другого DataSet.

Если подставлять значение параметра в параметризованный запрос через свойство Params, то обычно нужно сделать четыре шага:

1. Закрыть TQuery
2. Подготовить объект TQuery, вызвав метод Prepare
3. Присвоить необходимые значения свойству Params
4. Открыть TQuery

Второй шаг выполняется в том случае, если данный текст запроса выполняется впервые, в дальнейшем его можно опустить.

Вот фрагмент кода, показывающий как это может быть выполнено практически:

```
Query1->Close();
Query1->Prepare();
```

```
Query1->Params[0]->AsString = 'Argentina';
Query1->Open();
```

Этот код может показаться немного таинственным. Чтобы понять его, требуется внимательный построчный анализ. Проще всего начать с третьей строки, так как свойство Params является "сердцем" этого процесса.

Params - это индексированное свойство, которое имеет синтаксис как у свойства Fields для TDataSet. Например, можно получить доступ к первой переменной в SQL запросе, адресуя нулевой элемент в массиве Params:

```
Params[0]->AsString := '"Argentina"';
```

Если параметризованный SQL запрос выглядит так:

```
select * from Country where Name = :NameStr
```

то конечный результат (т.е. то, что выполнится на самом деле) - это следующее предложение SQL:

```
select * from Country where Name = "Argentina"
```

Все, что произошло, это переменной :NameStr было присвоено значение "Аргентина" через свойство Params. Таким образом, Вы закончили построение простого утверждения SQL.
Если в запросе содержится более одного параметра, то доступаться к ним можно изменяя индекс у свойства Params

```
Params[1]->AsString =  'SomeValue';
```

либо используя доступ по имени параметра

```
ParamByName('NameStr')-
>AsString:='"Argentina"';
```

Итак, параметризованные SQL запросы используют переменные, которые всегда начинаются с двоеточия, определяя места, куда будут переданы значения параметров.

Прежде, чем использовать переменную Params, сначала можно вызвать Prepare. Этот вызов заставляет Delphi разобрать ваш SQL запрос и подготовить свойство Params так, чтобы оно "было готово принять" соответствующее количество переменных. Можно присвоить значение переменной Params без предварительного вызова Prepare, но это будет работать несколько медленнее.

После того, как Вы вызывали Prepare, и после того, как присвоили необходимые значения переменной Params, Вы должны вызвать Open, чтобы закончить привязку переменных и получить желаемый DataSet. В нашем случае, DataSet должен включать записи где в поле "Name" стоит "Argentina".

Контрольные вопросы:

1. Как управлять соединением при помощи TDatabase?
2. Что такое объек TSession и зачем он нужен?
3. Какие парамены можн задавать в TDataset?
4. Что такое TQuery?
5. Как раотать с TQuery?
6. Как использовать параметризированные запросы в TQuery?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p.
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p.
3. Послед Б.С. Borland C++ Builder 6. Разработка приложений баз данных СПб.: ООО «ДиаСофтЮП», 2003 — 320 с.

Глава XV. Использование технологий ADO. Разработка приложений, использующих ADO и C++

15.1. Технология ADO

Технология Microsoft ActiveX Data Objects (ADO) представляет собой универсальный механизм доступа к различным источникам данных из приложений баз данных. Основу технологии ADO составляет использование набора интерфейсов общей модели объектов COM, описанных в спецификации OLE DB. Достоинством этой технологии является то, что базовый набор интерфейсов OLE DB имеется в каждой современной операционной системе Microsoft. Отсюда следует простота обеспечения доступа приложения к данным. При использовании технологии ADO приложение БД может использовать данные из электронных таблиц, таблиц локальных и серверных баз данных, XML-файлов и т.д.

В соответствии с терминологией ADO любой источник данных (базу данных, файл, электронную таблицу) называют хранилищем данных. Приложение взаимодействует с хранилищем данных с помощью провайдера. Для каждого типа хранилища данных используется свой провайдер ADO. Провайдер обеспечивает обращение к данным хранилища с запросами, интерпретацию возвращаемой служебной информации и результатов выполнения запросов для передачи их приложению.

Все объекты и интерфейсы ADO являются объектами и интерфейсами COM. Согласно спецификации OLE DB в состав COM входит следующий набор объектов:

- Command (команда) - служит для обработки команд (обычно SQL-запросов);
- Data Source (источник данных) - используется для связи с провайдером данных;
- Enumerator (перечислитель) - служит для перечисления провайдеров ADO;
- Error (ошибка) - содержит информацию об исключениях;
- Rowset (набор рядов) - строки данных, являющиеся результатом выполнения команды;
- Session (Сессия) - совокупность объектов, обращающихся к одному хранилищу данных;
- Transaction (транзакция) - управление транзакциями в OLE DB.

В системе программирования **C++ Builder** компоненты, используемые для создания приложений по технологии **ADO,** расположены на странице ADO Палитры компонентов. Охарактеризуем кратко назначение этих компонентов:

- ADOConnection - ADO-соединение, используется для установки соединения с ADO-источником данных и обеспечивает поддержку транзакций;
- ADOCommand - ADO-команды, используются для выполнения SQL-команд доступа к ADO-источнику данных без возвращения результирующего набора данных;
- ADODataSet - набор данных ADO, обеспечивает доступ к одной или более таблице ADO-источника данных и позволяет другим компонентам управлять этими данными,

связываясь с компонентом через компонент Datasource аналогично тому, как используется компонент Dataset. Может использоваться в компонентах ADOTabie, ADOQuery, ADOStoredProc;

- ADOTabie - таблица ADO, обеспечивает доступ к одной таблице ADO-источника данных и позволяет другим компонентам управлять этими данными, связываясь с компонентом ADOTable через компонент DataSource;
- ADOQuery - запрос ADO, позволяет выполнять SQL-команды для получения информации из ADO-источника данных и позволяет другим компонентам управлять этими данными, связываясь с компонентом ADOTable через компонент DataSource;
- ADOStoredProc - хранимая процедура ADO, позволяет приложениям получать доступ к хранимым процедурам, используя интерфейс ADO;
- RDSConnection - RDS-соединение, служит для управления передачей объекта Recordset от одного процесса (компьютера) к другому. Компонент используется для создания серверных приложений.

Компонент **ADOconnection** может использоваться как посредник между данными и другими компонентами **ADO**, что позволяет более гибко управлять соединением. Возможен вариант использования других компонентов **ADO** путем установления соединения с источником данных напрямую. Для этого компоненты **ADO** имеют свойство *ConnectionString*, с помощью которого могут создавать свой собственный канал доступа к данным.

Создадим в программе Access какую-нибудь простенькую базу данных с одной или несколько таблиц. Надеюсь вы знаете как это делать. Если нет, то советую почитать книгу "Microsoft Office Access для чайников", очень хорошая книга.

Все компоненты для работы с ActiveX Data Object (ADO) находятся на закладке под одноименным названием. Для начала нам понадобится поместить на форму компонент TADOConnection - именно он будут отвечать за связь нашего приложения с физической базой данных. Далее не забудьте поместить и TDataSource - без него мы не сможем графически отобразить

данные, он необходим для связи Data Controls компонентов с компонентом TADOTable, который тоже поместите на форму (он нужен для связи с конкретной таблицей БД, аналогично TTable из BDE).

Рис 15.1. Панель ADO.

Для связи всех трех компонентов между собой нужно проделать следующие операции:

1. У компонента TADOTable установить свойство Connection в положенный на форму компонент TADOConnection (По-умолчанию это ADOConnection1);
2. Свойство DataSet компонента TDataSource установите в ADOTable1;

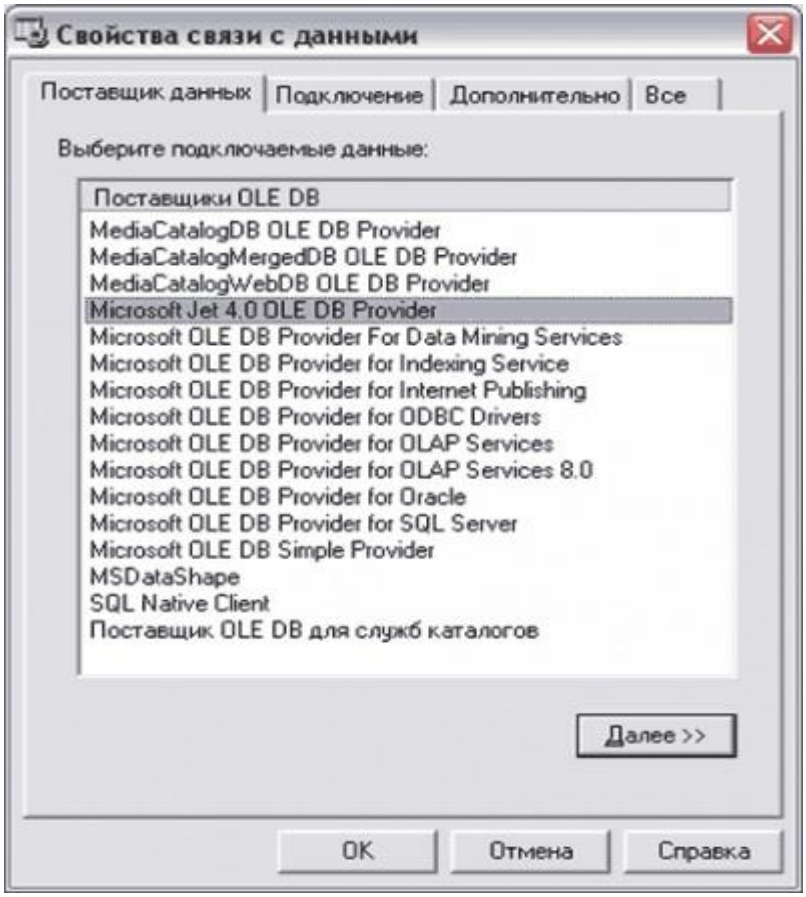


Рис 15.2. Настройка параметров соединения TADOConnection.

Настало время установить соединение с БД, но для начала его нужно настроить. Выберите компонент TADOConnection, дважды щелкните по нему. В появившемся окне выберите "Use

connection string" и нажмите Build. В ответ вам предложат выбрать драйвер для работы с базой. Хочу сразу вас обрадовать: в стандартную поставку Windows входят все необходимые драйвера, поэтому вам не нужно будет таскать с собой дистрибутив как в случае с BDE.

Так как мы будем работать с БД стандарта MDB от Access, то выберите драйвер Microsoft Jet 4.0 OLE DB и нажмите далее. Укажите путь к вашей базе данных и если вы установили контроль доступа к файлу, то введите имя пользователя и пароль.

На вкладке "Дополнительно" устанавливаются режим доступа к файлам и параметры сети, но в нашем случае последнее недоступно.

Теперь можете нажать OK и все настройки сохраняться в свойстве компонента TADOConnection под названием ConnectionString типа String.

Проверить есть ли соединение с базой данных можно установив свойство Connected (этого же компонента) в true.

Если IDE не выкинет никакой ошибки значит вы все настроили правильно и можно продолжать работу, если нет, то повторите все вышеописанные шаги.

Если вы не хотите, чтобы при соединении программа автоматически спрашивала пароль и пользователя доступа к файлу БД, то для этого есть свойство TADOConnection.LoginPrompt. Установите его в false.

Теперь выберите компонент TADOTable и укажите в свойстве TableName имя желаемой для работы таблицы из созданной вами базы данных (проверьте наличие true в свойстве TADOConnection.Connected). И напоследок, чтобы компонент получил доступ к базе данных, используя ADOConnection установите Active=true (TADOTable).

В принципе на этом настройка нашего приложения на работу с mdb-файлами закончена. Теперь займемся графическим интерфейсом, обработкой и отображением данных.

15.2. Программирование компонента TADOTable

Проделав вышеописанные операции, мы получили небольшое, но уже довольно серьезное приложение, позволяющее в полной мере работать с нашей базой. Однако анализировать имеющиеся данные несколько затруднительно. Неплохо было бы

добавить в программу поиск записей и фильтрацию по определенным критериями.

Для начала поиск. Для этого у компонента TADOTable предусмотрено множество функций:

- TADOTable.Locate(const AnsiString KeyFields, const System::Variant &KeyValues, TLocateOptions Options);
  Ищет в ключевом поле, заданном в переменной KeyFields, значение переменной KeyValues. В случае если совпадение найдено, найденная запись становится текущей, т.е. курсор устанавливается на нее. В переменной Options указывается как преобразовывать значения для поиска может быть: loCaseInsensitive или loPartialKey.
- TADOTable.Seek(const Variant &KeyValues, TSeekOption SeekOption =soFirstEQ); Используется при поиске с помощью индексов. SeekOption определяет как поступать если запись найдена: soFirstEQ, soLastEQ, soAfterEQ , soAfter, soBeforeEQ, soBefore. Подробное описание их вы найдете в мануале.
- TADOTable.LookUp (const AnsiString KeyFields, const Variant &KeyValues, const AnsiString ResultFields);
  Возвращает в качестве результата набор типа Variant со значениями всех найденных значений, удовлетворяющих запросу.

Существует еще несколько способов поиска информации в таблице: перебор всех элементов вручную, использование FindFirst, Next и т.д. Но мы перечислили только основные и необходимые, на мой взгляд, методы.

Приведем пример использования метода Locate. Поместите на форму компонент TComboBox, TButton и TEdit, занесите в КомбоБокс названия всех полей вашей таблицы (для этого используется свойство Items). В Edit мы будем вводить значение необходимое для поиска, а из ComboBox выбирать поле, по которому осуществлять поиск.

Теперь дважды щелкните по Button или выберите событие OnClick в инспекторе объектов.

Впишете в него следующий код:

```
void __fastcall TForm1::Button1Click(TObject
*Sender)
{
 ADOTable1->Locate(ComboBox1->Text,Edit1-
>Text,[loCaseInsensitive]);
//здесь по указанному в Combo полю ищем
значении из Edit. Все просто
//loCaseInsensitive - это поиск без ориентации
на строчные и прописные буквы
}
```

Видите, для организации простого поиска достаточного всего одной строчки кода.

Настало время добавить в наш проект фильтрацию. Добавить еще один TEdit и TButton на форму. Для понимания принципа этого будет достаточно. И разберите вот этот код:

void __fastcall TForm1::Button2Click(TObject *Sender)

```
{
 ADOTable1->Filtered=false;       // Отменяем
предыдущую фильтрацию если была
 ADOTable1->Filter=Edit2->Text;   // Выставляем
критерий фильтрации
 ADOTable1->Filtered=true;        // И снова
включает фильтрацию
}
```

Объясню, что может вводиться в Edit2. Мы создали некий очень далекий аналог командной строки. Вот шаблон, по которому работает фильтрация:

```
<имя поля> |параметр (=,>,<,>=,<= и т.д.)|
<значение>
```

Вот пример:

```
ФИО = Иванов Иван Петрович
```

В результате в таблице останутся только записи где, поле ФИО имеет значение "Иванов Иван Петрович" без внесения физических изменений в таблицу.

Контрольные вопросы:

1. Какие компоненты необходмы для создания подключения к базе данных через ADO?
2. Как производится настройка подключения в базе данных через ADO?
3. Какие методы предусмотрены для работы с TADOTable?
4. Как производится фильтрация данных в TADOTable?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p.
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p.
3. Послед Б.С. Borland C++ Builder 6. Разработка приложений баз данных СПб.: ООО «ДиаСофтЮП», 2003 — 320 с.

Глава XVI. Навигация по множеству строк, доступ к значениям полей, обновление строк, добавление и удаление через компоненты ADO

16.1. Язык манипуляции данными (DML).

Попробуем применить полученные сведения для создания приложения, использующего компоненты TTable, TQuery, TDataSource, TDBGrid. Воспользуемся для этой цели таблицами Customer.db и Orders.db, имеющимися в базе данных BCDEMOS, входящей в комплект поставки Borland C++ Builder. Данное приложение должно будет выводить списки клиентов из таблицы Customer, их заказов из таблицы Orders, а также позволять выбирать диапазон номеров клиентов.

Создадим новый проект и сохраним его главную форму как CUST1.CPP, а проект как CUST.MAK.

Изменим заголовок заголовок формы на "Контроль заказов". Разместим на форме компонент TDBGrid, два компонента TGroupBox, один компонент TTable, два компонента TQuery, три компонента TDataSource. На компонент GroupBox1 поместим три компонента TRadioButton и два компонента TButton. На компонент GroupBox2 поместим два компонента TEdit, два компонента TEdit и один компонент TButton.

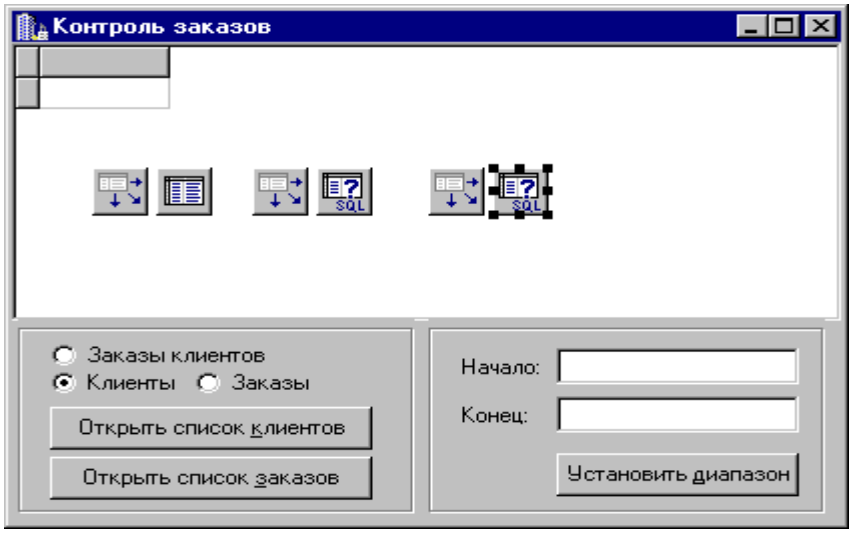


Рис. 16.1. Расположение компонентов на форме приложения CUST.

Установим следующие свойства для этих компонентов:

Имя компонента	**Свойство**	**Значение**
Table1	DatabaseName	BCDEMOS
	TableName	CUSTOMER.DB
	Active	false
DataSource1	DataSet	Table1
DBGrid1	DataSource	DataSource1
Query1	Database Name	BCDEMOS
	SQL	select * from orders
	Active	false
DataSource2	DataSet	Query1
Query2	DatabaseName	BCDEMOS
DataSource3	DataSet	Query2
Button1	Caption	Открыть список &клиентов
Button2	Caption	Открыть список &заказов
RadioButton1	Caption	Клиенты
	Checked	true
RadioButton2	Caption	Заказы
GroupBox1	Caption	
GroupBox2	Caption	
Button3	Caption	Установить &диапазон
Edit1	Text	
Edit2	Text	
Label1	Caption	Начало:

Label2	Caption	Конец:
RadioButton3	Caption	Заказы клиентов

Создадим обработчик события OnClick для кнопки Button1:

```
    void __fastcall TForm1::Button1Click(TObject
*Sender)
    {
    if (Table1->Active)
    {
    Table1->Close();
    Button1->Caption = "Ioe?uou nienie
&eeeaioia";
    }
    else
    {
    Table1->Open();
    Button1->Caption= "Cae?uou nienie
&eeeaioia";
    }
    }
```

Теперь при нажатии на эту кнопку таблица Customer будет то открываться, то закрываться, при этом будет изменяться и надпись на кнопке.

Создадим обработчик события OnClick для кнопки Button2:

```
    void __fastcall TForm1::Button2Click(TObject
*Sender)
    {
    if (Query1->Active)
    {
    Query1->Active = false;
    Button2->Caption = "Ioe?uou список
&caeacia";
    }
    else
    {
    Query1->Active = true;
    Button2->Caption = "Зае?uou список
&caeacia";
    }
```

```
    }
```

При нажатии на кнопку Button2 будет то открываться, то закрываться запрос Query1, содержащий список заказов:

Создадим обработчики событий OnClick для радиокнопок RadioButton1 и RadioButton2:

```
    void __fastcall
TForm1::RadioButton1Click(TObject *Sender)
    {
    DBGrid1->DataSource = DataSource1;
    }
    //-------------------------------------
-----------------------------
    void __fastcall
TForm1::RadioButton2Click(TObject *Sender)
    {
    DBGrid1->DataSource =DataSource2;
    }
```

Теперь с помощью этих радиокнопок можно переключаться между списком клиентов и списком заказов.

Создадим обработчик события OnClick для кнопки Button3:

```
    void __fastcall TForm1::Button3Click(TObject
*Sender)
    {
    if (Table1->Active)
    {
    Table1->SetRangeStart();
    Table1->Fields[0]->AsString = Edit1->Text;
    Table1->SetRangeEnd();
    Table1->Fields[0]->AsString = Edit2->Text;
    Table1->ApplyRange();
    }
    }
```

Теперь с помощью полей редактирования Edit1 и Edit2 и кнопки Button3 можно выбрать диапазон номеров клиентов для отображения сведений о них в DBGrid1.

Затем с помощью Visual Query Builder установим свойство SQL компонента Query2. Выберем в качестве имени базы данных

BCDEMOS и внесем в запрос таблицы CUSTOMER и ORDERS. Затем установим связь между таблицами, проведя линию от поля CustNo в таблице CUSTOMER к полю CustNo в таблице ORDERS.

Внесем в запрос следующие поля:

- Customer.CustNo
- Customer.Company
- Orders.OrderNo
- Orders.AmountPaid

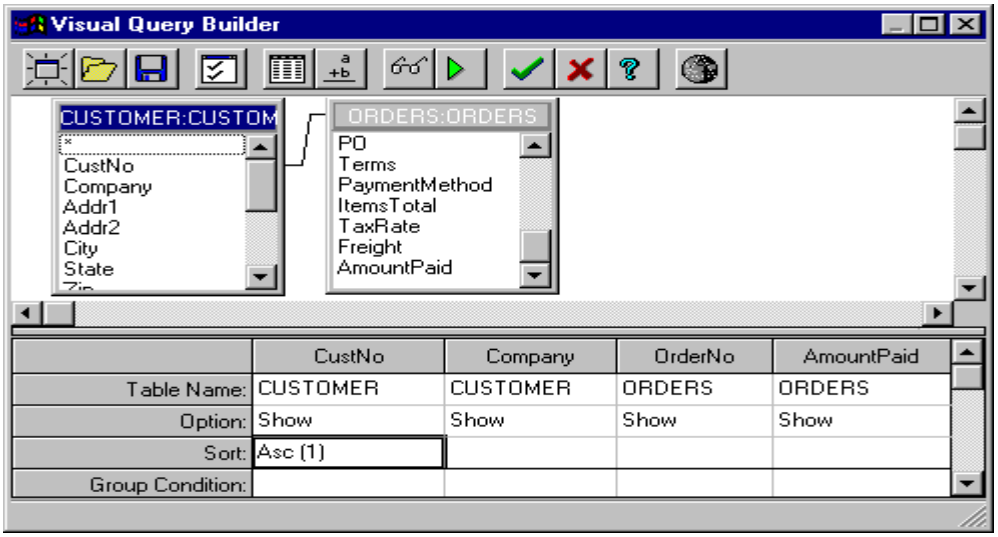


Рис. 16.2. Использование Visual Query Builder для создания комбинированного запроса.

Теперь отсортируем результаты запроса по номеру покупателя и выйдем из Visual Query Builder.

Используя инспектор объектов, выберем компонент Query2 и установим его свойство Active равным true.

Создадим обработчик события OnClick для RadioButton3:.

```
    void __fastcall
TForm1::RadioButton3Click(TObject *Sender)
    {
    DBGrid1->DataSource= DataSource3;
    }
```

Скомпилируем приложение. Щелкнем кнопками и "Открыть список заказов", чтобы открыть оба набора данных. Попробуем, используя радиокнопки, переключаться между ними.

Рис. 16.3. Так выглядит готовое приложение.

Нажмем кнопку "Открыть список клиентов". Введем значения полей "Начало" и "Конец" (например, 1200 и 1700 соответственно) и затем нажмем кнопку "Установить диапазон". Убедимся, что значения номеров заказов действительно находятся в пределах этого диапазона.

Щелкнем на радиокнопке "Заказы клиентов" и убедимся, что результирующий набор данных действительно содержит сведения из обеих таблиц.

Контрольные вопросы:

1. Какие действия необходимы для создания приложения с TADOQuery?
2. Какие стойства необходимо настроить в компоненах?
3. Как передать ифнормацию с формы в код?
4. Как обработать запрос с соединением?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p.
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p.
3. Послед Б.С. Borland C++ Builder 6. Разработка приложений баз данных СПб.: ООО «ДиаСофтЮП», 2003 — 320 с.

Глава XVII. XML и базы данных

17.1. Слабоструктурированные данные

Слабоструктурированными называются данные, обладающие определенной структурой, но эта структура может оказаться непостоянной, недостаточно изученной или неполной. Как правило, такие данные не могут быть описаны с помощью какой-либо неизменной схемы, поэтому иногда их называют не имеющими схемы (schema-less) или описывающими сами себя (self-describing). Характерной особенностью слабоструктурированных данных является то, что описательная информация, которая обычно выделяется в отдельную схему, присутствует в самих данных. В некоторых формах представления слабоструктурированных данных не предусмотрено применение отдельной схемы, а в других она существует, но налагает на представленные в ней данные очень слабые ограничения. В отличие от этого, для реляционных СУБД требуется заранее определенная схема, позволяющая распределить данные по таблицам, а все данные, управляемые этой системой, должны соответствовать такой структуре. Объектно-ориентированные СУБД допускают создание более развитой структуры по сравнению с реляционными СУБД, но также требуют, чтобы все данные укладывались в заранее заданную (объектно-ориентированную) схему. Но если в СУБД должны храниться слабоструктурированные данные, она должна формировать схему на основе этих данных, а не налагать на данные априорно заданную схему.

В последнее время обнаруживается значительный интерес к слабоструктурированным данным по многим причинам. Наиболее важные из них перечислены ниже.

- Для дальнейшей автоматизации обработки информации необходимо иметь возможность обращаться к источникам данных, представленным в Web, как к базам данных, но на эти источники невозможно наложить какую-либо заранее заданную схему.
- Количество разнотипных баз данных постоянно возрастает, поэтому задача создания гибкого формата,

обеспечивающего обмен данными между базами различных типов, становится все более важной.

- В связи со становлением языка XML (extensible Markup Language – расширяемый язык разметки) как стандарта представления и обмена данными в Web появились перспективы создания новых методов обработки слабоструктурированных данных, поскольку язык XML хорошо подходит для их описания.

Большинство подходов к управлению слабоструктурированными данными основано на использовании языков запросов, обеспечивающих прохождение по древовидному размеченному графу, который служит для представления таких данных. Если данные невозможно описать с помощью схемы, то единственный способ их идентификации состоит в указании позиции элемента данных в коллекции, а не формализации его структурных свойств. Это означает, что способы выполнения запросов к данным теряют свой традиционный декларативный характер и становятся в большей степени навигационными. Начнем знакомство со слабоструктурированными данными с рассмотрения примера, который показывает, для обработки данных какого типа может быть предназначена слабоструктурированная система.

Рассмотрим структуру, показанную в листинге 1, где изображена часть данных, представленных в учебном проекте DreamHome. Эти данные можно изобразить графически, как показано на рис. 1. Они описывают одно отделение компании (находящееся по адресу 22 Deer Rd), двух сотрудников компании (John White и Ann Beech) и два объекта недвижимости, предназначенные для сдачи в аренду (которые находятся, соответственно, по адресу 2 Manor Rd и 18 Dale Rd); на этом рисунке показаны также некоторые связи между данными. В частности, следует отметить, что данные не совсем правильно формализованы:

- для сотрудника John White отдельно представлены имя и фамилия, а для сотрудника Ann Beech имя и фамилия хранятся как один компонент, а также приведены данные о зарплате;

- для объекта недвижимости, находящегося по адресу 2 Manor Rd, хранится месячная арендная плата, а для объекта недвижимости, который находится по адресу 18 Dale Rd, хранится годовая арендная плата;
- для объекта недвижимости по адресу 2 Manor Rd обозначение типа объекта недвижимости (flat) хранится в виде строки, а для объекта недвижимости по адресу 16 Dale Rd обозначение типа (house) представлено в виде целого числа.

```
DreamHome (&1)
Branch (&2)
    street (&7) "22 Deer Rd"
    Manager &3
Staff (&3)
    name (&8)
        fName (&17) "John"
        IName (&8) "White
    ManagerOf &2
Staff (&4)
    name (&9) "Ann Beech"
    salary (&10) 12000
    Oversees &5
    Oversees &6
PropertyForRent (&5)
    street (&11) "2 Manor Rd"
    type (&2) "Flat"
    monthlyRent (&3) 375
    OverseenBy &4
PropertyForRent (&6)
    Street (&14) "16 Dale Rd"
    type (&15) 1
    annualRent (&6) 7200
    OverseenBy &4
```

Листинг 1. представления слабоструктурированных данных в базе данных проекта DreamHome.

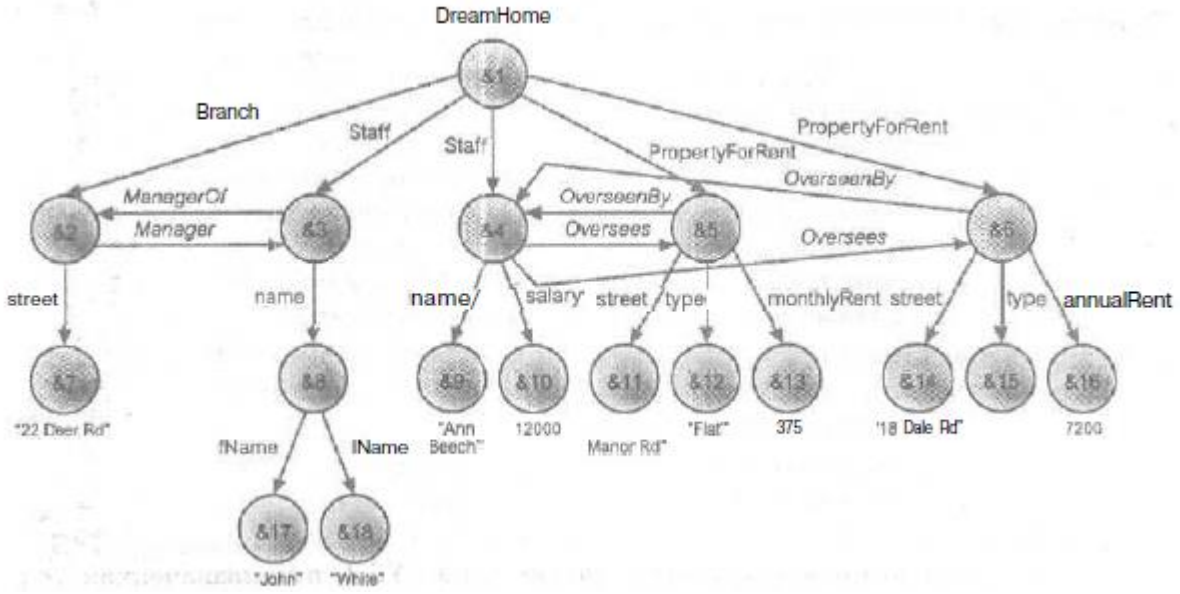


Рисунок 17.1. Графическое изображение данных, приведенных в листинге 1.

1. Предоставлять функциональные средства базы данных.
2. Поддерживать идентичность объектов.
3. Обеспечивать инкапсуляцию.
4. Поддерживать объекты со сложным состоянием

17.2. Модель обмена объектными данными (OEM)

Одной из первых для описания слабоструктурированных данных была предложена модель обмена объектными данными ***(Object Exchange Model – OEM)***. Это – модель представления вложенных объектов, которая была первоначально разработана для проекта TSIMMIS (The Stanford-IBM Manager of Multiple Information Sources) и должна была обеспечить интеграцию данных, полученных из различных источников. Модели OEM не имеют схемы и описывают сами себя. Такую модель можно рассматривать как размеченный ориентированный граф, узлы которого представляют собой объекты (как показано на рис. 17.1).

Для каждого объекта OEM задаются уникальный идентификатор объекта (например, &7), описательная текстовая метка (street), тип (строка) и значение ("22 Deer Rd"). Объекты подразделяются на элементарные и составные. Элементарные объекты содержат значение базового типа (такого как целое число или строка). Они характеризуются тем, что на графе изображаются без исходящих ребер. Все остальные объекты называются составными. Тип этих объектов определен как множество

идентификаторов объектов, а на графе они изображаются как узлы, имеющие одно или несколько исходящих ребер. Составной объект OEM рассматривается как родительский по отношению к своим дочерним объектам OEM. Каждый отдельный объект OEM может иметь произвольное количество родительских объектов. Это позволяет создавать сети любой сложности для моделирования всех необходимых связей между данными.

Метка (label) служит для обозначения объекта и указывает, что собой представляет объект (именно поэтому данные, представленные в модели OEM, называют описывающими самих себя); это означает, что текст метки должен нести максимально возможный объем описательной информации. Метки могут изменяться динамически. Имя – это специальная метка, которая применяется в качестве псевдонима для одного из объектов и может служить точкой входа в базу данных (например, DreamHome – это имя, обозначающее объект &1).

Любой объект OEM можно рассматривать как четверку {label, old, type, value}. Например, ниже показано, как может быть представлен объект Staff (узел &4), содержащий объекты name и salary, объект name (узел &9), который включает строку "Ann Beech", и объект salary (узел &10), который включает десятичное значение 12000.

```
{Staff, &4, множество, {&9, &10}}
{name, &9, строка, "Ann Beech"}
{salary, &10, десятичное значение, 12000}
```

Модель OEM была создана специально для обработки неполных и неформализованных данных, а приведенный выше пример показывает, как в этой модели отображаются данные с нерегулярной структурой и неопределенным типом.

17.3. Основные сведения о языке XML

В настоящее время большинство документов, которые хранятся и передаются в Web, представлены в коде HTML. Как указано выше, одним из преимуществ языка HTML является его простота, что позволяет применять этот язык для самых разных категорий пользователей. Но можно также обоснованно утверждать, что простота этого языка является не только его достоинством, но и недостатком. В частности, язык HTML не

допускает расширения набора дескрипторов, а необходимость в этом возникает все чаще, поскольку новые дескрипторы могли бы позволить упростить решение некоторых задач и сделать документы HTML более привлекательными и динамичными. Пытаясь удовлетворить такую потребность, разработчики браузеров предпринимают попытки ввести некоторые дескрипторы HTML, предназначенные только для конкретных браузеров. Но такие попытки приводят к тому, что создание развитых, общедоступных документов Web еще более усложняется и появляются разные трактовки HTML. Для предотвращения развития тенденции к разделению языка HTML на диалекты W3C предложил новый стандарт – ***XML (Extensible Markup Language – расширяемый язык разметки)***. Этот язык позволяет обеспечить такую же независимость приложений от данных, благодаря которой в свое время HTML стал таким переносимым и мощным языком, обеспечившим независимость средств представления от данных.

XML. Метаязык (язык для описания других языков), который позволяет, проектировщикам создавать специализированные дескрипторы для реализации функциональных возможностей; не достижимых с помощью HTML.

Язык XML представляет собой подмножество языка ***SGML (Standard Generalized Markup Language – стандартный обобщенный язык разметки)***; он предназначен специально для оформления документов Web. XML – это мета-язык, позволяющий проектировщикам создавать собственные специализированные дескрипторы для реализации функциональных возможностей, недостижимых с помощью HTML. Например, XML обеспечивает возможность применять ссылки, которые указывают сразу на несколько документов, тогда как ссылка HTML может указывать только на один целевой документ.

SGML – это система определения структурированных типов документов и языков разметки, представляющих экземпляры документов таких типов. Язык SGML свыше десяти лет применяется как стандартный, независимый от программного обеспечения способ создания репозитариев для структурированной документации. Этот язык позволяет разделить любой документ на две логически независимые части; одна из них определяет структуру документа, а другая содержит сам текст.

Определение структуры называется определением типа документа (Document Type Definition – DTD). Язык SGML позволяет назначить для любого документа отдельно определенную структуру и дает возможность авторам документов создавать собственные, специализированные структуры, поэтому может стать основой чрезвычайно мощной системы управления документами. Но этот язык отличается значительной сложностью, поэтому не получил достаточно широкого распространения.

Язык XML предназначен для выполнения аналогичных функций, но является менее сложным и вместе с тем более приспособленным для использования в сети. При этом очень важно то, что язык XML сохранил основные преимущества SGML: расширяемость, структурируем ость и возможность проверки правильности документов. Поскольку XML является подмножеством SGML, любая система, полностью совместимая с SGML, обладает способностью обрабатывать документы XML (но обратное утверждение не является истинным). Тем не менее, язык XML не предназначен для замены SGML. К тому же он не может заменить и язык HTML, который также основан на SGML и является приложением этого языка. В действительности XML предназначен для использования в качестве дополнения к языку HTML и должен обеспечить передачу через Web данных различных типов. Благодаря своим широким возможностям язык XML фактически может применяться не только для разметки текста, но и для разметки звука или изображения, т.е. мультимедийных данных. В качестве примеров широко применяемых языков, созданных на основе XML, можно указать ***MathML (Mathematics Markup Language – язык математической разметки)***, ***SMIL (Synchronized Multimedia Integration Language – язык интеграции синхронизированных источников мультимедийной информации)*** и ***CML (Chemistry Markup Language – язык химической разметки)***.

Несмотря на то что широкое применение языка XML началось примерно пять лет назад (с тех пор, как спецификация XML 1.0 была формально утверждена W3C в конце 1998 года), этот язык уже оказал значительное влияние на многие сферы информационной технологии, включая графические интерфейсы, встроенные системы, распределенные системы и управление базами данных. Например, язык XML позволяет наглядно

описывать структуру данных, поэтому может стать удобным механизмом для определения структуры разнородных баз данных и источников данных. А поскольку XML позволяет определить всю схему базы данных, то он в принципе может использоваться для выборки содержимого, например, всей схемы СУБД Oracle, и преобразования ее в схему Informix или Sybase.

Язык XML уже фактически стал стандартным средством обмена данными в индустрии программного обеспечения и быстро заменяет системы ***EDI (Electronic Data Interchange – электронный обмен данными)***, которые в свое время служили основным механизмом обмена данными между предприятиями. Некоторые аналитики предполагают, что со временем на языке XML будет создаваться и храниться большинство документов как в Internet, так и за ее пределами. В настоящем разделе подробно рассматривается язык XML и показаны способы определения схем для языка XML. В следующем разделе описаны языки запросов для XML, вначале рассмотрим преимущества XML.

Преимущества XML

Некоторые преимущества использования XML в среде Web:

- Простота. XML – относительно простой язык, и его стандарт не превышает по объему 50 страниц. Он предназначен для использования в качестве языка, основанного на применении текста, обеспечивающего восприятие человеком и достаточно понятного.
- Открытый стандарт; независимость от платформы и программного обеспечения. XML не зависит от платформы и программного обеспечения. Он представляет собой подмножество языка SGML, который определен стандартом ISO. Язык XML также основан на стандарте (ISO 10646), в нем предусмотрена поддержка набора символов Unicode, и поэтому он может служить для представления текста на всех алфавитах, в частности, предоставляет возможность указать применяемый национальный язык и кодировку.
- Способность к расширению. В отличие от HTML, язык XML является расширяемым; он позволяет пользователям

определять собственные дескрипторы в соответствии с требованиями к конкретному приложению.

- Возможность повторного использования. Расширяемость языка XML позволяет также создавать библиотеки дескрипторов XML и повторно использовать их во многих приложениях.
- Разделение информационного наполнения и средств представления. Язык XML позволяет хранить содержимое документа и независимо от этого описывать способ его представления (например, в браузере). Такая возможность упрощает создание специализированных средств отображения данных. При этом данные могут доставляться к пользователю с помощью браузера, а для вывода их на внешнее устройство может применяться способ, выбранный на месте, возможно, с учетом таких факторов, как пользовательские предпочтения или конфигурация. Таким образом, реализуется во многом такой же принцип, как в языке Java, который иногда описывают как язык, который позволяет "один раз разработать программу и применять ее где угодно". Поэтому XML называют языком, позволяющим "один раз подготовить документ и публиковать его где угодно", благодаря использованию таких средств, как таблицы стилей, с помощью которых один и тот же документ XML может публиковаться разными способами, с применением различных форматов и носителей.
- Улучшенное распределение нагрузки. Язык XML позволяет доставлять данные в браузер (на клиентский компьютер) для проведения некоторых вычислений на месте; при этом сервер частично освобождается от вычислительной нагрузки и обеспечивается более равномерное распределение нагрузки.
- Поддержка интеграции данных из нескольких источников. Задача интеграции данных из нескольких разнородных источников является чрезвычайно сложной и требует больших затрат времени. Но язык XML позволяет намного упростить объединение данных из различных источников. Для совместного применения данных, поступающих из

серверных баз данных и других приложений, могут использоваться специальные программные агенты, которые затем передают данные другим клиентам или серверам для дальнейшей обработки или презентации.

- Способность описывать данные, которые относятся к приложениям многих разных типов. Поскольку XML является расширяемым, он может также использоваться для описания данных, содержащихся в самых различных приложениях. К тому же, поскольку язык XML позволяет создать формат представления данных, в котором данные описывают сами себя, последующая передача и обработка данных может происходить без использования каких-либо встроенных дополнительных описаний данных.
- Усовершенствованные механизмы поиска. В настоящее время в процессе работы поисковых машин применяется информация, содержащаяся в метадескрипторах HTML, или анализируется взаимное расположение ключевых слов. А при использовании XML поисковые машины могут применяться просто для интерпретации дескрипторов с описаниями.
- Новые возможности. По-видимому, одним из наибольших преимуществ XML являются те безграничные возможности, которые открываются с введением этой новой технологии.

Краткий обзор языка XML.

В этом разделе представлен краткий обзор языка XML с использованием простого примера, показанного в листинге 1, который содержит сведения о сотрудниках компании.

```
<?xml version= "1.0" encoding= "UTF-8"
standalone= "yes"?>
<?xml:stylesheet type = "text/xsl" href =
"staff_list.xsl"?>
<!DOCTYPE STAFFLIST SYSTEM "staff_iist.dtd">
<STAFFLIST>
    <STAFF branchNo = "B005">
        <STAFFNO>SL2K/STAFFNO>
            <NAME>
```

```
                <FNAME>John</FNAME><LNAME>White</
            LNAME>
            </NAME>
        <POSITION>Manager</POSITION>
        <DOB>l-Oct-45</DOB>
        <SALARY>3OOOO</SALARY>
    </STAFF>
    <STAFF branchNo = "B003">
        <STAFFNO>SG37</STAFFNO>
            <NAME>
                <FNAME>Ann</FMAMExLNAME>Beech</LN
            AME>
            </NAME>
        <POSITION>Assistant</POSITION>
        <SALARY>12000</SALARY>
    </STAFF>
</STAFFLIST>
```

Листинг 2. Пример документа XML, в котором представлены сведения о сотрудниках компании.

Объявление XML.

Документы XML начинаются с необязательного объявления XML, которое в данном примере содержит обозначение версии XML, применяемой автором документа (1.0), и системы кодировки (UTF-8 соответствует стандарту Unicode), а также содержит сведения о том, имеется ли в документе ссылка на внешние объявления разметки (standalone = 'yes' указывает, что в документе отсутствуют внешние объявления разметки). Вторая и третья строки документа XML, показанного в листинге 2, относятся к таблицам стилей и определениям DTD, которые кратко рассматриваются ниже.

Элементы.

Элементы XML, называемые также дескрипторами, представляют собой наиболее широко применяемую форму разметки. Первый элемент документа должен быть так называемым корневым элементом, который может содержать другие элементы (субэлементы). Каждый документ XML должен иметь один корневой элемент, которым в данном примере является <STAFFLIST>. Любой элемент начинается с начального

дескриптора (например, <STAFF>) и оканчивается конечным дескриптором (например, </STAFF>). Элементы XML чувствительны к регистру, поэтому элемент <STAFF> рассматривается как отличный от элемента <staff> (следует отметить, что в языке HTML такое условие не соблюдается). Элемент может быть пустым, и в этом случае его можно сокращенно представить одним дескриптором, например <EMPTYELEMENT/>. Элементы должны быть правильно вложенными, как показывает следующий фрагмент листинга2:

```
<STAFF>
    <NAME>
        <FNAME>John</FNAMExLNAME>White</LNAME>
    </NAME>
</STAFF>
```

В этом случае элемент NAME полностью вложен в элемент STAFF, а элементы FNAME и LNAME вложены в элемент NAME.

Атрибуты

Атрибуты представляют собой пары "имя-значение", которые содержат описательную информацию об элементе. Атрибуты помещаются внутри начального дескриптора после соответствующего имени элемента, а значение атрибута заключаются в кавычки. Например, при подготовке рассматриваемого листинга было решено представить номер отделения, в котором работает данный сотрудник, с помощью атрибута branchNo элемента STAFF:

```
<STAFF branchNo = "B005">
```

Следует отметить, что данные об отделении можно было бы с таким же успехом представить в виде субэлемента элемента STAFF. А для представления данных о поле сотрудника компании можно использовать атрибут пустого элемента, например следующим образом:

```
<SEX gender = "M"/>
```

Каждый конкретный атрибут может присутствовать в дескрипторе только в одном экземпляре, тогда как субэлементы в одном и том же дескрипторе могут повторяться. Следует отметить, что в данном случае может возникнуть неопределенность: должна ли быть представлена информация о

номере отделения или поле сотрудника с помощью элемента или атрибута?

Ссылки на сущности

Сущностями называются элементы документа, которые выполняют следующие основные задачи:

- служат в качестве сокращений для обозначения часто повторяющегося текста или позволяют включить в документ содержимое внешних файлов;
- используются для вставки в текст произвольных символов Unicode (например, для представления символов, которые нельзя ввести непосредственно на клавиатуре);
- позволяют обозначить различие между зарезервированными символами и информационным наполнением. Например, левая угловая скобка (<) обозначает начало начального или конечного дескриптора элемента. Чтобы можно было отличить этот символ разметки от символа, содержащегося в самом информационным наполнении, в язык XML введена сущность It, которая служит для замены символа <.

Каждая сущность должна иметь уникальное имя, и ее применение в документе XML называется ссылкой на сущность. Любая ссылка на сущность начинается с символа амперсанда (&) и оканчивается точкой с запятой (;), например <.

Комментарии

Комментарии заключаются в дескрипторы <!-- и --> и могут содержать любые данные, кроме литеральной строки "--". Комментарии могут помещаться между дескрипторами разметки в любом месте документа XML, но процессор XML не обязательно должен передавать комментарии приложению.

17.4. Разделы CDATA и команды обработки

Раздел CDATA является для процессора XML указанием, что процессор должен игнорировать содержащиеся в этом разделе символы разметки и передавать заключенный в нем текст непосредственно приложению без интерпретации. Для передачи дополнительной информации приложению могут также применяться команды обработки. Команда обработки имеет форму `<?name pidata?>`, где `name` служит для приложения

идентификатором команды обработки. Поскольку команды зависят от приложения, любой документ XML может содержать несколько команд обработки, позволяющих передать разным приложениям команду на выполнение одинаковых действий, но, возможно, различными способами.

Упорядочение

В слабоструктурированной модели данных, описанной в разделе 1, предполагается, что коллекции являются неупорядоченными, тогда как в языке XML элементы рассматриваются как упорядоченные. Таким образом, в языке XML следующие два фрагмента с переставленными элементами FNAME и LNAME считаются разными:

```
<NAME>                    <NAME>
    <FNAME>John</FNAME>     <LNAME>White</LNAME>
    <LNAME>White</LNAME>    <FNAME>John</FNAME>
</NAME>                   </NAME>
```

В отличие от этого, атрибуты в XML не являются упорядоченными, поэтому следующие два элемента XML считаются одинаковыми:

```
<NAME FNAME = "John" LNAME * "White"/>
<NAME LNAME = "White" FNAME = "John"/>
```

Определения типов документов (DTD)

Для обозначения элементов, применяемых в конкретном приложении, иногда используется термин словарь (vocabulary). Синтаксическая структура, называемая также грамматикой, определяется с помощью формы ***EBNF (Extended Backus Naur Form – расширенная форма Бэкуса-Наура)***, а не синтаксиса XML. Хотя определение DTD является необязательным, рекомендуется предусматривать его для каждого документа в целях проверки его соответствия определенным требованиям, как описано ниже.

Продолжая пример с данными о сотрудниках, рассмотрим приведенное в листинге 3 определение DTD для документа XML, показанного в листинге 2. В данном случае определение DTD находится в отдельном внешнем файле, но может быть также вложено непосредственно в документ XML, Объявления DTD подразделяются на следующие четыре типа, которые

рассматриваются ниже: объявления типов элементов, объявления списков атрибутов, объявления сущностей и объявления обозначений.

Объявления типов элементов

Объявления типов элементов определяют правила применения элементов, которые могут встретиться в документе XML. Например, в листинге 3 задано следующее правило (называемое также моделью информационного наполнения) для элемента `STAFFLIST`:

```
<!ELEMENT STAFFLIST (STAFF)*>
```

Листинг 3. Определение типа документа, соответствующее документу XMЦ приведенному в листинге 2

```
<!ELEMENT STAFFLIST (STAFF)*>
<!ELEMENT STAFF (NAME, POSITION, DOB?, SALARY)>
<!ELEMENT NAME (FNAME, LNAME}>
<!ELEMENT FNAME (#PCDATA)>
<!ELEMENT LNAME (#PCDATA)>
<!ELEMENT POSITION (ftPCDATA)>
<!ELEMENT DOB (#PCDATA)>
<!ELEMENT SALARY {#PCDATA)>
<!ATTLIST STAFF branchNo CDATA ft IMPLIED>
```

Это правило указывает, что элемент `STAFFLIST` состоит из элементов `STAFF`, количество которых может быть равно нулю или большему числу. Для обозначения кратности повторения могут примеряться следующие символы:

- звездочка (*) обозначает, что количество вхождений субэлемента в элемент может быть равно нулю или большему числу;
- знак "плюс" (+) указывает, что количество вхождений для данного элемента может составлять от одного и больше;
- вопросительный знак (?) указывает, что количество вхождений может быть равно либо нулю, либо одному.

Если имя элемента приведено без символа, обозначающего количество вхождений, то элемент должен появиться в документе точно один раз. Запятые между именами элементов указывают, что элементы должны присутствовать в документе в указанной

последовательности, а если запятые между именами элементов отсутствуют, элементы могут появляться в документе в любом порядке.

Например, для элемента STAFF задано следующее правило:

```
<!ELEMENT STAFF (NAME, POSITION, DOB?, SALARY)>
```

Это правило содержит информацию о том, что элемент `STAFF` состоит из элементов `NAME` и `POSITION`, необязательного элемента `DOB` и элемента `SALARY` в указанном порядке. В определении должны также быть предусмотрены объявления для элементов `FNAME`, `LNAME`, `POSITION`, `DOB` и `SALARY` и всех прочих элементов, применяемых в модели информационного наполнения. Это позволяет использовать процессор XML для проверки допустимости документа. Все эти основные элементы объявлены с помощью специального символа `ftPCDATA`, который указывает на присутствие в них интерпретируемых символьных данных. Следует отметить, что любой элемент может содержать только элементы, но предусмотрена также возможность включить в элемент другие элементы и фрагмент `#PCDATA` (и в этом случае к нему применяется термин элемент, содержащий смешанное информационное наполнение).

Объявления списков атрибутов.

Объявления списков атрибутов указывают, какие элементы могут иметь атрибуты, какие в них могут содержаться атрибуты, какие значения могут иметь атрибуты и каковыми являются необязательные значения атрибутов, применяемые по умолчанию,, Каждое объявление атрибута состоит из трех частей: имени, типа и необязательного значения по умолчанию. Ниже перечислены шесть возможных типов атрибутов.

`CDATA`. Символьные данные, содержащие любой текст. Строка, которая относится к этому типу, не анализируется процессором XML и просто передается непосредственно в приложение.

`ID`. Идентификатор, применяемый для обозначения отдельных элементов в документе. Идентификаторы `ID` должны соответствовать имени элемента, и все значения `ID`, применяемые в документе, должны быть разными.

IDREF или IDREFS. Одна или несколько ссылок на идентификаторы, которые должны соответствовать значению одного атрибута ID для некоторого элемента в документе. Атрибут IDREFS может содержать несколько значений IDREF, разделенных пробелами.

ENTITY или ENTITIES, Одно или несколько обозначений сущностей, которые должны соответствовать имени одной сущности. Атрибут ENTITIES может содержать несколько значений ENTITY, разделенных пробелами.

NMTOKEN или NMTOKENS. Строка ограниченного формата, как правило, состоящая из одного слова. Атрибут NMTOKENS может содержать несколько значений NMTOKEN, разделенных пробелами.

Список имен. Значения, которые может иметь данный атрибут (иными словами, перечислимый тип).

Например, следующее объявление атрибута используется для определения атрибута branchNo элемента STAFF:

```
< LATTLIST STAFF branchNo CDATA #IMPLIED>
```

В этом объявлении указано, что значение • атрибута branchNo представляет собой строку (CDATA – character data) и является необязательным (!IMPLIED), а применяемое по умолчанию значение для него не предусмотрено. Кроме ключевого слова !IMPLIED, предусмотрено ключевое слово #REQUIRED, которое указывает, что данный атрибут должен быть всегда задан в элементе. Если ни одно из этих уточняющих ключевых слов не задано, то атрибут может содержать объявленное значение по умолчанию. Для указания на то, что атрибут должен всегда иметь значение, предусмотренное по умолчанию, применяется ключевое слово ttFIXED. В качестве примера укажем, что элемент SEX можно определить как имеющий атрибут gender (пол), содержащий либо значение M (по умолчанию), либо значение F, следующим образом:

```
<!ATTLIST SEX gender (M | F } "M">
```

Объявления сущностей и обозначений.

Объявления сущностей позволяют связать имя с некоторым фрагментом ин формационного наполнения, таким как отрывок из обычного текста, часть определения DTD или ссылка на внешний

файл, содержащий текст или двоичные данные. Объявления обозначений указывают на внешние двоичные данные, которые просто передаются процессором XML в приложение. Например, можно объявить сущность для текста "DreamHome Estate Agents" следующим образом:

```
<!ENTITY DH "DreamHome Estate Agents">
```

Обязанности по обработке внешних неинтерпретированных сущностей возлагаются на приложение. После идентификатора, обозначающего местонахождение сущности, должна быть объявлена некоторая информация о внутреннем формате этой сущности, например, следующим образом:

```
<!ENTITY dreamHomeLogo SYSTEM "dreamhome.jpg"
NDATA JPEGFormat>
<!NOTATION JPEGFormat SYSTEM
"http://www.jpeg.org">
```

Здесь наличие ключевого слова `NDATA` указывает на то, что сущность является неинтерпретируемой; произвольное имя, которое следует за этим ключевым словом, является просто ключом для следующего объявления обозначения. В объявлении обозначения это имя сопоставляется с идентификатором, который используется в приложении для определения способа обработки сущности.

17.5. Идентификаторы элементов, идентификаторы ID и ссылки на идентификаторы ID

Как указано выше, язык XML позволяет зарезервировать обозначение типа атрибута `ID`, с помощью которого можно связать с элементом уникальный ключ. Кроме того, тип атрибута `IDREF` позволяет включить в элемент ссылку на другой элемент с указанным ключом, а тип атрибута `IDREFS` позволяет оформить в элементе ссылку на несколько других элементов. Например, чтобы промоделировать неформальную связь `Branch Has Staff`, можно определить следующие два атрибута для элементов `BRANCH` и `STAFF`:

```
<!ATTLIST STAFF staffNo ID #REQUIRED>
<!ATTLIST BRANCH staff IDREFS #IMPLIED>
```

После этого можно применить эти атрибуты, как показано в листинге 4.

```
<STAFF staffNo = "SL21">
<NAME>
<FNAME>John</FNAME><LNAME>White</LNAME>
</NAME>
</STAFF>
<STAFF staffNo = "SL41">
<NAME>
<FNAME>Julie</FNAMExLNAME>Lee</LNAME>
</NAME>
</STAFF>
«BRANCH staff = "SL21 SL41">
<BRANCHNO>B005</BRANCHNO>
</BRANCH>
```

Листинг 4. Пример применения типов атрибутов ID и IDREFS

Проверка допустимости документа

В спецификации XML предусмотрены два этапа проверки правильности документа в процессе его обработки: проверка формальной правильности и допустимости. Процессор, не обеспечивающий проверку допустимости, перед передачей информации документа XML приложению проверяет только, является ли документ формально правильным. Таковым считается любой документ XML, который соответствует правилам, определяющим структуру и систему обозначений документа XML. Кроме всего прочего, формально правильные документы XML должны соответствовать следующим требованиям:

- документ должен начинаться с объявления XML – <?xml version "l.0"?>;
- все элементы должны быть включены в один корневой элемент;
- элементы должны быть вложены друг в друга в виде древовидной структуры, без каких-либо перекрытий элементов;
- все непустые элементы должны иметь начальный и конечный дескрипторы.

Процессор, обеспечивающий проверку допустимости, должен не только проверить, является ли документ XML формально правильным, но и установить, соответствует ли он также определению DTD; в случае положительного результата этой проверки документ XML считается допустимым. Как указано выше, определение DTD может содержаться в документе XML или может быть указано в нем с помощью ссылки. В настоящее время W3C рекомендует использовать способ определения допустимости документов, обладающий большими выразительными возможностями по сравнению с DTD и известный как XML Schema. Прежде чем перейти к описанию спецификации XML Schema, рассмотрим некоторые другие технологии, связанные с XML, которые используются в определениях XML Schema.

17.6. Технологии XML

Интерфейсы DOM и SAX.

API-интерфейсы XML подразделяются главным образом на две категории: основанные на древовидном представлении и основанные на обработке событий.

Интерфейс ***DOM (Document Object Model – объектная модель документа)*** представляет собой API-интерфейс для XML, основанный на древовидном представлении, который обеспечивает создание объектно-ориентированного представления данных. Этот API-интерфейс был создан W3C и описывает ряд независимых (от платформы и языка) интерфейсов, позволяющих преобразовывать во внутреннюю форму любые формально правильные документы XML или HTML. Интерфейс DOM формирует древовидное представление документа XML в оперативной памяти и предоставляет доступ к классам и методам, позволяющим перемещаться и обрабатывать элементы этого дерева в приложении. В частности, модель DOM определяет интерфейс Node с подклассами Element, Attribute и Character-Data. Интерфейс Node содержит методы для доступа к таким компонентам узла, как parentNode { } , который возвращает родительский узел, и childNodes ( ) , который возвращает множество дочерних узлов. В целом интерфейс DOM в наибольшей степени подходит для выполнения таких структурных

манипуляций с деревом XML, как добавление или удаление элементов, а также изменение порядка следования элементов.

На рис. 17.2 показан в виде древовидной структуры документ XML, приведенный в листинге 2. Следует отметить важное различие между представлением этих данных в виде графа OEM (см. рис. 1) и в виде дерева элементов XML. В представлении OEM граф имеет метки на ребрах, а в представлении XML – метки на узлах. Если данные имеют иерархическую структуру, их можно легко преобразовать из одного представления в другое, а если данные представлены в виде графа, такое преобразование становится намного сложнее.

Интерфейс SAX (Simple API for XML – простой API-интерфейс для XML) – это основанный на обработке событий API-интерфейс последовательного доступа для XML, в котором используются функции обратного вызова для передачи приложению сообщений о событиях, возникающих в ходе синтаксического анализа документа. Например, события могут активизироваться при обнаружении начальных и конечных дескрипторов элементов. В приложении для обработки этих событий применяются специализированные обработчики событий. В отличие от API-интерфейсов, основанных на использовании древовидного представления, API-интерфейсы, основанные на обработке событий, не предусматривают формирования в оперативной памяти древовидного представления документа XML.

Следует отметить, что API-интерфейс SAX создан в результате совместных усилий разработчиков, участвующих в деятельности списка рассылки XML-DEV, а не является программным продуктом W3C.

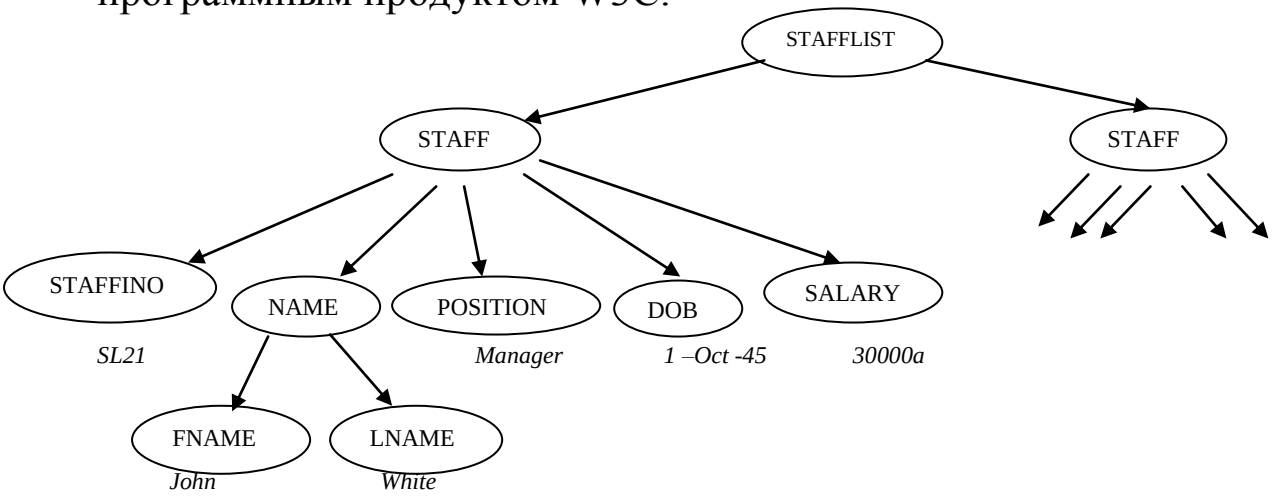


Рисунок 17.2. Представление документа XML, приведенного в листинге 2. в виде древовидной структуры.

Спецификация Namespaces.

Спецификация Namespaces обеспечивает возможность применять специальные уточняющие обозначения к именам элементов и ссылкам, содержащимся в документах XML. Это позволяет избежать коллизий имен при использовании таких элементов, которые имеют одно и то же имя, но определены в разных словарях. Благодаря этому появляется возможность применять в документе дескрипторы, которые определены в разных пространствах имен. Это необходимо, если данные в документе получены из нескольких источников. Пространства имен определены также в рекомендации W3C. Для обеспечения их уникальности элементам и атрибутам присваиваются глобально уникальные имена с помощью ссылки URI. Например, в следующем фрагменте документа используются два разных пространства имен, которые объявлены в корневом элементе. Первое из них ("http: //www. dreamhome. co.uk/branch5/") служит в качестве пространства имен, применяемого по умолчанию, поэтому подразумевается, что все элементы, не имеющие уточнителей, принадлежат к этому пространству имен. Второму пространству имен ("http://www.dreamhome.co.uk/HQ/") присвоено имя (hq), которое в дальнейшем используется в качестве префикса элемента SALARY для обозначения пространства имен, к которому относится данный элемент:

```
<STAFFLIST xmlns =
"http://www.dreamhome.co.uk/branch5/1"
        xmlns:hq =
    "http://www.dreamhome.co.uk/HQ/">
    <STAFF branchNo = "B005">
        <STAFFNO>SL21</STAFFNO>
        ...
        <hq:SALARY>30000</hq:SALARY>
    </STAFF>
</STAFFLIST>
```

Языки XSL и XSLT.

Стилем называется формализованное описание способа отображения информационных элементов в браузере. При обработке данных в коде HTML по умолчанию применяется информация о стиле, встроенная в программное обеспечение браузера. Такая возможность обусловлена тем, что набор дескрипторов для HTML является заранее определенным и постоянным. Разработчики могут также предусмотреть иной способ обработки дескрипторов HTML. Для этого применяется спецификация каскадных таблиц стилей (***Cascading Stylesheet Specification – CSS***). В отличие от HTML, для документов XML не предусмотрены стили, применяемые по умолчанию. Для представления документа XML в браузере также может применяться спецификация CSS, но она не позволяет вносить в документ структурные изменения. Поэтому W3C определил формальную рекомендацию по использованию, языка расширяемых таблиц стилей (***extensible Stylesheet Language – XSL***), который создан специально для определения способа отображения в браузере данных документа XML, а также способа преобразования одного документа XML в другой. Этот язык по своему назначению аналогичен спецификации CSS, но является более мощным.

Расширяемый язык таблиц стилей для поддержки преобразований (***extensible Stylesheet Language for Transformations – XSLT***) представляет собой подмножество языка XSL. Он одновременно является и языком разметки, и языком программирования, поскольку в ним предусмотрены механизмы преобразования структуры XML в любую другую структуру XML, в формат HTML или в любой другой текстовый формат (такой как SQL). Хотя язык XSLT может применяться для описания формата вывода Web-страницы на дисплей, его основным достоинством является способность преобразовывать базовые структуры данных, а не просто обеспечивать представление этих структур на устройствах вывода, как в случае CSS.

Язык XSLT имеет большое значение, поскольку он предоставляет механизмы динамического изменения способа отображения документа и фильтрации данных. Этот язык является также достаточно надежным для того, чтобы на нем можно было кодировать прикладные алгоритмы. К тому же он позволяет формировать из полученных данных графические изображения (а

не только документы). Язык XSLT позволяет даже обеспечить связь с серверами (особенно в сочетании с модулями сценариев, которые могут быть встроены в код XSLT) и дает возможность формировать соответствующие сообщения непосредственно в коде XSLT. В качестве иллюстрации в листинге 5 приведена таблица стилей XSL, применяемая для вывода на экран документа XML, приведенного в листинге 2.

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:template match="/">
    <html>
    <body background ="sky.jpg">
    <center><h2><i>DreamHome</i>Estate
Agents</h2></center>
    <table border-"Iм bgcolor="#ffffff ">
    <tr>
        <th bgcolor= "#cOcOcO" bordercolor=
    "#000000">staffNo</th>
<!-- Повторить для других заголовков столбцов -
->
    </tr>
    <XSl:for-each select= "STAFFLIST/STAFF">
    <tr>
        <td bordercolor="#cOcOcO"><xsl:value-of
    select="STAFFNO"/>
        </td>
        <td border-color ="#cOcOcO"><xsl:value-
    of select="NAME/FNAME"></td>
<!-- Повторить для других элементов -->
    </tr>
    </xsl:for-each>
    </table>
    </body>
    </html>
</xsl:template>
</xsl:stylesheet>
```

Листинг 5. Таблица стилей XSL, предназначенная для вывода на экран документа XML, приведенного в листинге 2

Язык XPath (XML Path).

XPath – это декларативный язык запросов для XML, в котором предусмотрены простые синтаксические конструкции для адресации отдельных частей документа XML. Как описано ниже, этот язык предназначен для использования с языком XSLT (при сопоставлении с шаблоном) и языком XPointer (при обработке адреса). Язык XPath позволяет осуществлять выборку коллекций элементов, указывая путь, сформированный по такому же принципу, как путь к каталогу. При этом обозначение пути может включать или не включать условные выражения. В языке XPath используется компактный строковый синтаксис, а не структурированный синтаксис, основанный на применении элементов XML, поэтому выражения XPath могут применяться и в атрибутах XML, и в идентификаторах URI.

В языке XPath документ XML рассматривается как логическое (упорядоченное) дерево, в котором узлами обозначаются каждый элемент, атрибут, текст, команда обработки, комментарий, пространство имен и корневой элемент. В основе механизма адресации лежит использование контекстного узла (таковым является начальная точка пути) и пути локализации (который описывает путь от одной точки документа XML к другой и предоставляет тем самым способ адресации элементов документа XML). Для обозначения абсолютного или относительного местонахождения элемента в документе может применяться язык XPointer. Путь локализации состоит из ряда так называемых шагов, разделяемых символом косой черты (/), который выполняет такую же функцию, как и символ / в пути к каталогу (отсюда и происходит термин путь локализации). Каждый символ косой черты позволяет перейти на более низкий уровень древовидного представления по сравнению с предыдущим шагом.

Каждый шаг состоит из базиса (basis) и необязательных предикатов, а базис состоит из имени оси и условия проверки узла. Условие проверки узла позволяет определить тип узла в документе. Обычно в нем проверяется имя элемента, но могут

также применяться такие функции, как text() (для проверки того, является ли узел текстовым) или node ( ) (для проверки узла любого типа). В языке XPath определено 13 типов осей, включая ancestor (предок), attribute (атрибут) и child (потомок). Например, как показано на рис. 4, элемент `STAFF` имеет ось `child`, которая состоит из пяти узлов (`STAFFNO`, `NAME`, `POSITION`, `DOB` и `SALARY`). Предикат должен быть заключен в квадратные скобки и находиться после базиса. Если элемент содержит больше одного субэлемента, для выборки конкретного субэлемента может применяться конструкция `[positionO = positionNumber]`, где `positionNumber` – номер позиции, начинающийся с 1. В языке XPath поддерживается полный и сокращенный синтаксис. Некоторые примеры путей локализации показаны в табл. 2

Таблица 17.2. Некоторые примеры путей локализации

Путь локализации	**Назначение**
.	Выбирает контекстный узел
..	Выбирает родительский узел (предок) контекстного узла
/	Выбирает корневой узел или служит в качестве разделителя между шагами в пути
//	Выбирает дочерние узлы (потомков) текущего узла
/child::STAFF	Выбирает асе элементы STAFF, являющиеся дочерними узлами корневого узла
child::STAFF (или просто STAFF)	Выбирает асе элементы STAFF, являющиеся дочерними узлами контекстного узла
attribute т:branchNo (ИЛИ просто ©branchNo)	Выбирает атрибут branchNo контекстного узла
attribute::* (или просто®*)	Выбирает все атрибуты контекстного узла
child::STAFF[3]	Выбирэеттретий элемент STAFF,

	являющийся дочерним узлом контекстного узла
/child::STAFF[®branchNo = "B005"]	Выбирает все элементы STAFF, которые имеют атрибут со значением branchHo, равным B005
/child::STAFF[®branchNo = "B005"][position () =1]	Выбирает первый элемент STAFF, который имеет атрибут со значением branchNo, равным B005

Язык XPointer (XML Pointer).

Язык XPointer предоставляет доступ к значениям атрибутов или к содержимому элементов, находящихся в любом месте документа XML. Любой указатель XPointer по сути представляет собой выражение XPath, входящее в идентификатор URI. Кроме всего прочего, язык XPointer позволяет сформировать ссылки на разделы текста, выбрать конкретные элементы или атрибуты и перейти с одного элемента на другой. Этот язык дает возможность также осуществить выборку информации, содержащейся более чем в одном наборе узлов, тогда как с помощью языка XPath эту задачу выполнить невозможно.

Кроме определения узлов, в языке XPointer определяются также точки и диапазоны, которые в сочетании с узлами позволяют обозначить местонахождение данных. Точка – это позиция в документе XML, а диапазон обозначает всю структуру и информационное наполнение XML между начальной и конечной точками, причем и та и другая могут находиться в середине узла. Например, следующий указатель XPointer обозначает диапазон, который начинается с начала дочернего элемента `STAFF`, имеющего значение атрибута `branchNo`, равное `B005`, и оканчивается в конце дочернего элемента `STAFF`, имеющего значение атрибута `branchNo`, равное `B003`:

```
XPointer(/child::STAFF[attribute:rbranchNo =
"B005"] to
/child::STAFF[attribute::branchNo = "B003"])
```

В рассматриваемом примере такая конструкция позволяет выполнить выборку обоих узлов `STAFF`.

Язык XLink (XML Linking).

Язык XLink позволяет вставлять в документы XML специальные элементы, с помощью которых можно создавать и описывать ссылки между ресурсами. В нем применяется синтаксис XML для создания структур, позволяющих описывать ссылки, аналогичные простым, однонаправленным гиперссылкам HTML, а также более сложные ссылки. Ссылки XLink могут относиться к одному из двух типов: простым и расширенным. Простая ссылка соединяет источник с ресурсом назначения, а расширенная ссылка соединяет произвольное количество ресурсов. Кроме того, этот язык предоставляет возможность хранить ссылки в отдельной базе данных (называемой базой ссылок – linkbase). Тем самым предоставляется возможность достичь в определенной степени независимости от местонахождения ресурсов: даже в случае изменения ссылок исходные документы XML остаются неизменными, и обновления вносятся только в базу ссылок.

Язык XHTML.

Язык XHTML (extensible HTML – расширяемый язык HTML) версии 1.0 представляет собой язык HTML 4.01, переформулированный на языке XML 1.0. Он предназначен для использования в качестве языка HTML следующего поколения и по сути представляет собой более строгую и четко сформулированную версию HTML. Например, в нем предусмотрены следующие требования:

- имена дескрипторов и атрибутов должны быть обозначены прописными буквами;
- все элементы XHTML должны иметь конечный дескриптор;
- значения атрибутов должны быть заключены в кавычки, а сокращенный способ их обозначения не допускается;
- вместо атрибута name должен применяться атрибут ID;
- документы должны соответствовать правилам XML.

Определение XML Schema.

В спецификации XML 1.0 для определения модели информационного наполнения (допустимого порядка следования и вложенности элементов), а также (в ограниченной степени) типов

данных атрибутов документа XML предусмотрено использование механизма DTD, но этот механизм имеет целый ряд недостатков:

- определения DTD оформляются с помощью синтаксиса, отличного от XML;
- в них отсутствует поддержка пространств имен;
- они позволяют определить лишь крайне ограниченный набор типов данных.

Поэтому возникла необходимость предусмотреть более полный и строгий метод определения модели информационного наполнения документа XML. Спецификация XML Schema, разработанная W3C, лишена указанных недостатков и обладает гораздо большей выразительной мощью по сравнению с определениями DTD. Дополнительные возможности представления данных позволяют обеспечить гораздо более надежный обмен данными XML между приложениями Web по сравнению с использованием инструментальных средств проверки допустимости, разработанных с учетом требований только конкретного приложения.

Схема XML представляет собой определение конкретной структуры XML (ее организации и применяемых в ней типов данных). Язык XML Schema описывает способ определения в схеме элемента каждого типа и позволяет указать типы данных, связанных с каждым элементом. Схема сама является документом XML, в котором используются элементы и атрибуты, выражающие семантику схемы. А поскольку схема – документ XML, то ее можно редактировать и обрабатывать с помощью таких же инструментальных средств, какие применяются для обработки описанного ею документа XML. В этом разделе рассмотрен пример создания схемы XML для документа XML, приведенного в листинге 2.

Простые и сложные типы

Вероятно, один из самых простых способов создания схемы XML состоит в отслеживании структуры документа и определении каждого элемента по мере его обнаружения. Элементы, содержащие другие элементы, относятся к типу сложных элементов complexType. Например, обнаружив корневой элемент `STAFFLIST`, можно определить, что он относится к типу complexType. Список дочерних элементов элемента `STAFFLIST`

описывается с помощью элемента sequence. Он относится к типу элемента-составителя (compositor), который определяет упорядоченную последовательность субэлементов:

```
<xsd:element name = "STAFFLIST">
<xsd:complexType>
<xsd:sequence>
<!-- Здесь должны находиться определения
дочерних элементов -->
</xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Каждый из элементов в схеме обозначен типовым префиксом xsd:, который связан с пространством имен W3C XML Schema с помощью объявления xmlns:xsd="http://www.w3.org/2000/lO/XMLSchema" (это объявление помещено в элемент schema). Элементы STAFF и NAME также содержат субэлементы и могут быть определены аналогичным образом. Элементы, не имеющие субэлементов или атрибутов, относятся к простому типу simpleType. Например, элементы STAFFNO, DOB и SALARY можно представить следующим образом:

```
<xsd:element name = "STAFFNO" type =
"xsd:string"/>
<xsd:element name = "DOB11 type = "xsd:date"/>
<xsd:element name = "SALARY" type =
"xsd:decimal"/>
```

Эти элементы объявлены с помощью заранее определенных типов W3C XML Schema, т.е. string, date и decimal, а для обозначения их принадлежности к словарю XML Schema введен префикс xsd:. Кроме того, атрибут `branchNo`, который должен всегда занимать последнюю позицию, может быть объявлен следующим образом:

```
<xsd:attribute name="branchNo"
type="xsd:string"/>
```

Кардинальность

Язык XML Schema позволяет представить кардинальность некоторого элемента с помощью атрибутов minOccurs

(минимальное количество вхождений) и maxOccurs (максимальное количество вхождений). Для обозначения необязательного элемента атрибуту minOccurs присваивается значение 0, а для указания на то, что максимальное количество вхождений не ограничено, атрибуту maxOccurs может быть присвоено значение unbounded. В качестве значения любого незаданного атрибута принимается по умолчанию 1. Например, поскольку элемент DOB является необязательным, для обозначения этого может быть предусмотрена следующая конструкция:

```
<xsd:element name="DOB" type="xsd;date"
minOccurs="0"/>
```

А для регистрации вплоть до трех имен ближайших родственников (Next-Of-Kin – NOK) каждого сотрудника компании можно применить конструкцию

```
<xsd:element name="NOK" type="xsd:string"
minOccurs="0" naxOccurs="3"/>
```

Ссылки.

Хотя метод, описанный выше (который предусматривает разработку определения для каждого обнаруженного элемента), является относительно простым, он не позволяет добиться глубокой вложенности встроенных определений, а полученная в результате схема может оказаться неудобной для чтения и сопровождения. Поэтому для создания схемы применяется также подход с использованием ссылок (reference) на объявления элементов и атрибутов, находящихся в области определения того элемента, в котором они используются. Например, можно определить элемент `STAFFNO` следующим образом:

```
<xsd:element name="STAFFNO" type="xsd:string"/>
```

и использовать это определение в схеме каждый раз, когда встречается элемент `STAFFNO`, как показано ниже.

```
<xsd:element ref="STAFFNO"/>
```

Если в документе XML имеется много ссылок на какой-то элемент, в данном случае `STAFFNO`, то использование ссылок позволяет хранить его определение только в одном месте и тем самым повысить удобство сопровождения схемы.

Определение новых типов

В языке XML Schema предусмотрен третий механизм создания элементов атрибутов на основе объявления новых типов данных. Применяемый при этом способ аналогичен определению класса, а затем использованию его для создания объекта. В частности, могут быть определены простые типы для элементов PCDATA или атрибутов, а также сложные типы для элементов. Новым типам присваиваются имена, а их определения размещаются за пределами объявлений элементов и атрибутов. Например, новый простой тип для элемента STAFFNO можно определить следующим образом:

```
<xsd:simpleType name-l lSTAFFNOTYPE">
<xsd:restriction base="xsd:string">
<xsd:maxLength value="5"/>
</xsd:restriction>
</xsd:simpleType>
```

Этот новый тип определен как ограничение (restriction) данных типа string из пространства имен XML Schema (атрибут base), а также указано, что он имеет максимальную длину 5 символов (элемент maxLengtn называется фасетом). В спецификации XML Schema предусмотрено 15 фасетов, в том числе length, minLength, minlnclusive и maxlnclusive. Двумя другими чрезвычайно удобными фасетами являются pattern и enumeration. Элемент pattern определяет регулярное выражение, с которым должно сопоставляться проверяемое значение. Например, на элемент STAFFNO может быть наложено ограничение, чтобы его значение состояло из двух прописных символов, за которыми следуют от одной до трех цифр (например, SG5, SG37, SG999). Это требование можно представить на языке XML Schema с помощью следующего шаблона pattern:

```
<xsd:pattern value="[A-Z] {2} [0-9]{l,3}">
```

Элемент enumeration позволяет ограничить простой тип набором различимых значений. Например, на элемент POSITION, содержащий данные о должности, можно наложить ограничение, согласно которому он может иметь только значения Manager, Supervisor или Assistant. Такое требование можно представить в схеме с помощью следующего перечисления enumeration:

```
<xsd:enumeration value="Manager"/>
```

```
<xsd:enumeration value="Supervisor"/>
<xsd:enumeration value="Assistant"/>
```

Группы.

Язык XML Schema позволяет вводить в схему определения групп элементов и групп атрибутов. Группа – это не тип данных, а конструкция, применяемая как контейнер и содержащая некоторый набор элементов или атрибутов. Например, данные о сотрудниках компании могут быть представлены в виде группы следующим образом:

```
<xsd:group name="STAFFTYPE">
    <xsd:sequence>
        <xsd:element name="STAFFNO"
        type="STAFFNOTYPE"/>
        <xsd:element name="POSITION"
        type="POSITIONTYPE"/>
        <xsd:element name="DOB"
        type="xsd:date"/>
        <xsd:element name="SALARY"
        type="xsd:decimal"/>
    </xsd:sequence>
</xsd:group>
```

После обработки этой конструкции создается именованная группа `STAFFTYPE`, представляющая собой последовательность элементов (для упрощения здесь показаны только некоторые элементы `STAFF`). В определении схемы можно также создать элемент `STAFFLIST` со ссылкой на группу, определенную как последовательность в количестве нуль или больше элементов `STAFFTYPE`, следующим образом:

```
<xsd:element name="STAFFLIST">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:group ref="STAFFTYPE"
                                minOccurs="0"
                                maxOccurs-
            "unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
```

```
</xsd:element>
```

Элементы-составители choice и all.

Как указано выше, sequence – это представитель одного из типов элемента-составителя (compositor). Предусмотрены еще два типа элементов-составителей: choice и all. Элемент-составитель choice определяет возможность выбора между несколькими допустимыми элементами или группами элементов, а элементсоставитель all определяет неупорядоченный набор элементов. Например, такую ситуацию, при которой имя сотрудника компании может быть обозначено только одной строкой или сочетанием двух строк (с именем и фамилией), можно отразить с помощью следующей конструкции:

```
<xsd:group name="STAFFNAMETYPE">
    <xsd:choice>
    <xsd:element name="NAME" type="xsd:string"/>
        <xsd:sequence>
            <xsd:element name="FNAME"
            type="'xsd: string"/>
            <xsd:element name="LNAME"
            type="xsd:string"/>
        </xsd:sequence>
    </xsd:choice>
</xsd:group>
```

Списки и объединения.

Для создания списков элементов, разделенных пробелами, применяется элемент list. Например, список, содержащий табельные номера сотрудников компании, может быть создан следующим образом:

```
<xsd:simpleType name="STAFFNOLIST">
    <xsd:list itemType=STAFFNOTYPE/>
</xsd: sitnpleType>
```

Ниже приведен пример применения этого типа в документе XML.

```
<STAFFNOLIST>"SG5" "SG37" "SG999"</STAFFNOLIST>
```

Теперь на основе списка этого типа может быть определен новый тип, представляющий собой некоторую форму ограничения, например, может быть объявлен ограниченный

список, количество элементов в котором равно 10, следующим образом:

```
<xsd:simpleType name="STAFFNOLIST10">
    <xsd:restriction base="STAFFNOLIST">
        <xsd:Length value="10"/>
    </xsd: restriction
</xsd:simpleType>
```

Простые типы (называемые также элементарными) и списки позволяют определять значения элементов или атрибутов, состоящие из одного или нескольких экземпляров некоторого элементарного типа. В отличие от этого, тип объединения позволяет выбирать значения элемента или атрибута из одного или нескольких экземпляров одного типа, выбранных из объединения нескольких элементарных типов или списков. Применяемый при этом формат аналогичен объявлению choice, описанному выше, поэтому здесь не приведено его подробное описание. Заинтересованный читатель может обратиться к документу W3C XML Schema. Пример схемы XML для документа XML, приведенного в листинге 2, показан в листинге 6.

```
<?xml version= "1.011 encoding= "UTF-8"?>
<xsd:schema xmlns:xsd=
"http://www.w3.org/2000/10/XMLSchema">
<!-- Создать группу для STAFFLIST -->
<xsd:group name="STAFFLISTGROUP">
    <xsd:element. name»"STAFFLIST">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:group ref="STAFFTYPE"
                                minOccurs="Q"
                                maxOccurs="unbou
            nded"/>
            </xsd:sequence>
        </xsd:complexType >
    </xsd:element>
</xsd:group>
<!-- Создать тип для элемента STAFFNO -->
<xsd:simpleType name="STAFFNQTYPE">
    <xsd:restriction base="xsd:string">
```

```
        <xsd:maxLength value="5"/>
        <xsd:pattern value="[A-Z]{2} [0-
        9]{l,3}">
    </xsd:restriction>
</xsd:simpleType>
<!-- Создать тип для атрибута branchNo -->
<xsd:sirapleType name="BRANCHNOTYPE">
    <xsd:restriction base="xsd:string">
        <xsd:maxLength value="4"/>
        <xsd:pattern value="[A-Z] [0-9]{3}">
    </xsd:restriction>
</xsd:simpleType>
<]-- Создать тип для элемента POSITION ->
<xsd:simpleType narae="POSITIONTYPE">
    <xsd:restriction base="xsd:atring">
        <xsd:enumeration value="Manager"/>
        <xsd:enumeration value="Supervisor"/>
        <xsd:enumeration value="Assistant"/>
    </xsd: restriction
</xsd:simpleType>
<!-- Создать группу для элемента STAFF -->
<xsd:group name="STAFFTYPE">
    <xsd:element name="STAFF" >
        <xsd:complexType>
            <xsd:sequence»
                <xsd:element name="STAFFNO"
                type="STAFFNOTYPE"/>
                <xsd:element name="NAME">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="FNAME"
                        type="xsd:string"/>
                        <xsd:element name="LNAME"
                        type="xsd:string"/>
                    </xsd:sequence>
                </xsd:complexType>
                </xsd:element>
                <xsd:element name="POSITION"
                type="POSITIONTYPE"/>
```

```
                <xsd:element name="DOB"
                type="xsd:date"/>
                <xsd:element name="SALARY"
                type="xsd:decimal"/>
                <xsd:attribute name="branchNo"
                type="BRANCHNOTYPE"/>
            <xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:group>
</xsd:schema>
```

Листинг 6. Схема XML для документа XML, приведенного в листинге 2

Ограничения.

В предыдущих разделах уже рассматривались некоторые примеры применения фасетов для проверки допустимости данных, применяемых в документе XML. В спецификации XML Schema предусмотрено также средство определения ограничений уникальности и соответствующих ссылочных ограничений, которые должны соблюдаться в определенном диапазоне значений данных, основанных на использовании средств XPath. В этом разделе рассматриваются ограничения двух типов: ограничения уникальности и ограничения по ключу.

Ограничения уникальности.

Для определения ограничения уникальности должен быть задан элемент unique, определяющий элементы или атрибуты, которые должны быть уникальными. Например, можно определить ограничение уникальности по данным, состоящим из фамилии и даты рождения (Date Of Birth – DOB) сотрудника компании, с помощью следующей конструкции:

```
<xsd:unique name="NAMEDOBUNIQUE">
    <xsd:selector xpath="STAFF"/>
    <xsd:field xpath="NAME/LNAME"/>
    <xsd:field xpath="DOB"/>
</xsd:unique>
```

Местонахождение уникального элемента в схеме позволяет определить контекстный узел, на который распространяется такое ограничение. В данном случае дескриптор, задающий

ограничение, следует за элементом STAFF; тем самым указано, что это ограничение должно быть уникальным только в контексте элемента STAFF, аналогично тому, как определяется ограничение на отношении в реляционной СУБД. Выражения XPath, заданные в следующих трех элементах, являются относительными к контекстному узлу. Первое выражение XPath с элементом selector задает элемент, на который распространяется ограничение уникальности (в данном случае STAFF), а следующие два элемента field указывают узлы, которые должны проверяться на уникальность.

Ограничения по ключу.

Ограничение по ключу аналогично ограничению уникальности, за исключением того, что проверяемое с помощью него значение не должно быть пустым. Оно также позволяет ссылаться на ключ. В следующем примере показано ограничение по ключу, заданное на узле STAFFNO:

```
<xsd:key name="STAFFNOISKEY">
    <xsd:selector xpath="STAFF"/>
    <xsd:field xpath="STAFFNO"/>
</xsd:key>
```

Еще один тип ограничения позволяет ограничивать ссылки значениями указанных ключей. Например, атрибут branchNo в конечном итоге предназначен для указания некоторого отделения компании. Если предположить, что соответствующий элемент создан с ключом BRANCHNOISKEY, то значение этого атрибута можно ограничить значениями ключа следующим образом:

```
<xsd:keyref name="BRANCHNOREF" refer
"BRANCHNOISKEY">
    <xsd:selector xpath="STAFF"/>
    <xsd:field xpath="@branchNo"/>
</xsd:keyref>
```

17.7. Инфраструктура описания ресурсов (RDF)

Очевидно, что спецификация XML Schema предоставляет более полный и строгий метод определения модели информационного наполнения документа XML, чем определения DTD. Но она все же не обеспечивает поддержку необходимого

уровня семантической функциональной совместимости. Например, если два приложения должны обмениваться информацией с помощью XML, то назначение и подразумеваемый смысл применяемых при этом документов должны быть согласованы с той структурой данных, которая ими формируется. Но для этого необходимо сформировать модель предметной области с описанием данных, которые требуются для того и иного приложения. Это позволяет четко определить, какие данные должны передаваться в прямом и обратном направлениях от одного приложения к другому. Такую модель обычно принято описывать в терминах объектов или отношений (например, в предыдущих главах для этой цели использовался язык UML). А поскольку схема XML описывает только синтаксическую структуру документа, одна и та же модель предметной области может быть представлена в виде схемы XML многими разными способами. Поэтому невозможно определить непосредственное соответствие между моделью предметной области и определенной схемой. Эта проблема становится еще более сложной, если в обмене информацией должны участвовать не два, а несколько приложений. В таком случае недостаточно просто установить соответствие между несколькими схемами XML, поскольку задача здесь заключается не в преобразовании одной синтаксической структуры в другую, а в установлении соответствия между объектами и отношениями, принадлежащими к нескольким предметным областям. Таким образом, для решения описанной выше задачи необходимо пройти следующие три этапа:

1. восстановить первоначальные модели предметных областей из схем XML;
2. определить соответствия между объектами моделей предметных областей;
3. определить механизмы преобразования для документов XML, например с помощью языка XSLT.

Эти этапы на практике иногда становятся очень сложными, поэтому может оказаться, что язык XML хорошо подходит для обмена данными только между приложениями, для которых уже известна модель информационного наполнения, но плохо подходит для тех ситуаций, когда к обмену данными присоединяются все новые и новые приложения. Поэтому

требуется другой общепризнанный язык, позволяющий описывать интересующие предметные области. ***Инфраструктура описания ресурсов (Resource Description Framework – RDF)***, разработанная под эгидой W3C, представляет собой информационную среду, которая обеспечивает кодирование, обмен и повторное применение структурированных метаданных. Эта инфраструктура обеспечивает функциональную совместимость метаданных различных приложений за счет применения таких проектных механизмов, которые позволяют создавать общепринятые соглашения по семантике, синтаксису и структуре документов. Инфраструктура RDF не определяет семантику каждой рассматриваемой предметной области, а предоставляет возможность создавать по мере необходимости элементы метаданных для таких предметных областей. В инфраструктуре RDF в качестве общего синтаксиса для обмена и обработки метаданных применяется язык XML. С помощью средств языка XML создаются модели данных RDF, структура которых обеспечивает описание семантики, а также позволяет создавать единообразное описание и обеспечивать обмен стандартизированными метаданными.

Модель данных RDF.

Простейшая модель данных RDF состоит из трех объектов.

Ресурс. Ресурсом является все, что может иметь идентификатор URI, например Web-страница, ряд Web-страниц или даже часть Web-страницы, такая как элемент XML.

Свойство. Это – конкретный атрибут, который служит для описания ресурса. Например, атрибут Author может использоваться для описания лица, подготовившего конкретный документ XML.

Оператор. Это – конструкция, состоящая из сочетания ресурса, свойства и значения. Компоненты оператора RDF принято называть "субъектом", "предикатом" и "объектом". Например, утверждение "The Author of http://www.dreamhorae.co.uk/staff_list.xml is John White" (Автором документа... является Джон Уайт) представляет собой оператор. Последний может быть выражен в инфраструктуре RDF следующим образом:

```
<rdf:RDF
      xmlns:rdf="http://www.w3.org/1999/02/22-
rdf-syntax-nstf"
      xmlns:s="http://www.dreamhome.co.uk/schema
/">
<rdf:Description
    about="http://www.dreamhome.co.uk/staff_list
.xml">
    <s:Author>John White</s:Author>
</rdf:Description>
</rdf:RDF>
```

Эта информация может также быть представлена схематически с помощью направленного размеченного графа, показанного на рис. 17.5, а. А если бы потребовалось представить информацию с описанием автора, то данные о нем можно было бы промоделировать в виде ресурса, как показано на рис. 17.5, б.

В данном случае для описания таких метаданных может применяться следующий фрагмент XML:

```
<rdf:RDF
      xmlns:rdf="http://www.w3.org/1999/02/22-
rdf-syntax-nstt"
      xmlns:s="http://www.dreamhome.co.uk/schema
/">
    <rdf:Description
        about="http://www,dreamhome.co.uk/staff_
        list.xml">
        <s:Author
        rdf:resource="http://www.dreamhome.co.uk
    /Author__001"/>
    </rdf:Description>
    <rdf:Description
    about="http://www.dreamhome.co.uk/Author
    _001">
        <s:Name>John White</s:Name>
        <s:e-mail>white®dreamhome.co.uk</s:e-
        mail>
    </rdf:Description>
</rdf:RDF>
```

Схема RDF

Схема RDF позволяет представить информацию о классах, входящих в схему, в том числе об их свойствах (атрибутах) и о связях между ресурсами (классами). Иначе говоря, механизм определения схемы RDF предоставляет базовую систему типов для использования в моделях RDF, аналогично схеме XML. Схема RDF позволяет определить ресурсы и свойства (например, rdf s: Class и rdf s: subClassOf), которые используются при определении схем для конкретных приложений.

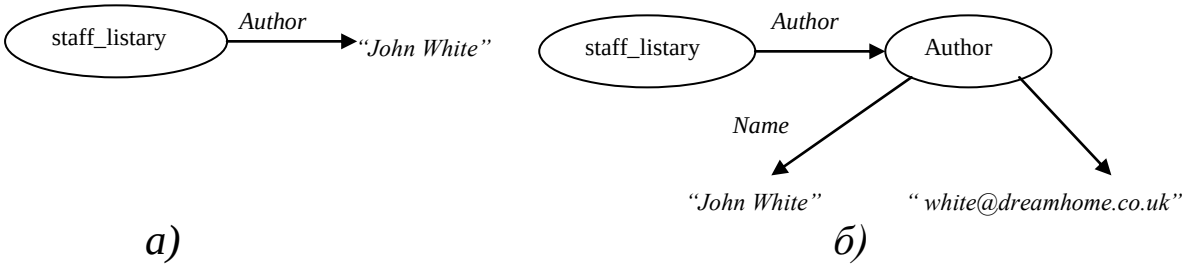


Рис. 17.5. Примеры форматов представления: а) представление информации об авторе в виде свойства; б) представление информации об авторе в виде ресурса.

Схемы RDF предоставляют также возможность определить некоторую часть ограничений, в частности указать необходимую кардинальность и определить допустимые свойства экземпляров классов. Для определения схемы RDF применяется декларативный язык, созданный под влиянием идей из области представления знаний (например, семантических сетей и логики предикатов), а также моделей представления схем баз данных, таких как двоичные реляционные модели, например NIAM, и моделей данных на основе графов. Более полное описание инфраструктуры RDF и схемы RDF выходит за рамки данной книги, и заинтересованный читатель для получения дополнительной информации может обратиться к документам W3C.

Языки запросов XML.

Языки запросов позволяют решать такие хорошо изученные задачи обеспечения функционирования баз данных, как выборку, модификацию и интеграцию данных. Но два стандартных языка запросов для СУБД, описанных в предыдущих главах (а именно SQL и OQL), не могут непосредственно применяться для работы с

данными XML в связи с нерегулярной структурой этих данных. Для обработки документов XML может применяться целый ряд языков слабоструктурированных запросов, включая XML-QL, UnQL и XQL компании Microsoft. В этих языках для перемещения по вложенной структуре XML применяется так называемое обозначение пути. Например, в языке XML-QL для описания части документа, подлежащей выборке, используется вложенная структура, подобная XML, а в результате создается искомая структура XML. Например, для выборки фамилий сотрудников, имеющих заработную плату свыше 30 000 фунтов стерлингов, может использоваться следующий запрос:

```
WHERE <STAFF>
    <SALARY> $S </SALARY>
    <NAMExFNAME> $F </FNAME> <LNAME> $L
    </LNAMEx/NAME>
    </STAFF> IN
    "http://www.dreamhome.co.uk/staff.xml"
    $S > 30000
CONSTRUCT <LNAME> $L </LNAME>
```

Рабочая группа XML Query Working Group

W3C недавно сформировал рабочую группу XML Query Working Group для подготовки модели данных документов XML, определения набора операторов запросов к этой модели и языка запросов, основанного на этих операторах запросов. Запросы выполняются на одном документе или на определенной коллекции документов и позволяют выбирать целые документы или поддеревья документов, соответствующие условиям, в которых учитывается информационное наполнение и структура документа. Запросы позволяют также формировать новые документы исходя из полученных данных. В конечном итоге намечено создание такого языка, который позволял бы обращаться к коллекциям документов XML как к базам данных. Ко времени написания этой книги указанная рабочая группа подготовила следующие четыре документа:

- XML Query Requirements (Требования к запросам XML);
- XML Query Data Model (Модель данных запросов XML);
- XML Query Algebra (Алгебра запросов XML);
- XQuery (Язык запросов для XML).

Документ ***XML Query Requirements*** определяет задачи, типичные сценарии применения и требования к модели данных запросов, алгебре запросов и языку запросов W3C XML. В этих требованиях утверждается следующее:

1. язык запросов должен быть декларативным и определенным независимо от каких-либо протоколов, с которыми он используется;
2. модель данных должна представлять не только символьные данные XML 1.0, но и простые и сложные типы спецификации XML Schema; она должна также включать поддержку для ссылок, заданных в пределах и вне пределов документа;
3. возможность выполнения запросов должна обеспечиваться независимо от того, существует ли для документа схема;
4. язык должен поддерживать кванторы общности и существования, заданные на коллекциях, обеспечивать агрегирование, сортировку и обработку пустых значений, а также позволять переходить по ссылкам внутри самого документа или от одного документа к другому.

К документу XML Query Requirements в качестве приложения приведен набор тестов в виде запросов XML с ожидаемыми результатами.

Модель данных запросов XML.

Документ XML Query Data Model с описанием модели данных запросов XML определяет информацию, которая может содержаться во входных данных любого процессора запросов XML. Эта модель данных основана на информационном наборе XML Information Set, который предоставляет описание информации, доступной в формально правильном документе XML, со следующими новыми средствами:

- поддержка типов XML Schema;
- представление коллекций документов, а также коллекций простых и сложных значений;
- представление ссылок.

Модель данных запросов XML – это представление, в котором используются размеченные узлы и конструкторы древовидных фрагментов. В ней для упрощения представления

значений ссылок XML (таких как IDREF, XPointer и URI) применяется понятие идентификатора узла. Каждый экземпляр модели данных представляет один или несколько полных документов или частей документов и может быть упорядоченным или неупорядоченным.

Основной составляющей модели данных является узел, который может представлять собой документ, элемент, значение, атрибут, пространство имен, команду обработки, комментарий или фрагмент информации. Весь документ XML представлен в виде узла DocNode. Часть документа – это поддерево документа, представленное узлом элемента (EletnNode), узлом значения (ValueNode), узлом команды обработки (PINode) или узлом комментария (CommentNode). В модели данных используются также ссылки на узлы для проверки и привязки идентификаторов узлов в конкретном экземпляре модели данных. Модель поддерживает функции Ref для создания ссылки на узел и Deref для получения узла, на который указывает ссылка. Ниже приведен краткий обзор основных объектов модели данных запросов XML Query Data Model.

Документ.

Документ DocNode имеет конструктор docNode, который принимает в качестве параметров значение ссылки URI и непустой упорядоченный лес2:

```
docNode :(uriReference, [EleraNode | PINode |
CommentNode])->DocNode
```

Для документа предусмотрены следующие три функции доступа:

```
uri     :DocNode->uriReference
children    :DocNode->[ElemNode ( PINode |
CoramentNode]
root    :DocNode->E1emNode
```

Функция uri возвращает значение ссылки URI узла DocNode, функция children возвращает упорядоченный лес дочерних элементов узла DocNode, а функция root возвращает уникальный узел EleraNode в упорядоченном лесу дочерних узлов родительского узла DocNode.

Элементы.

Узел ElemNode имеет два перегруженных конструктора elemNode. Первый конструктор принимает в качестве параметров значение дескриптора (уточненное значение имени QNameValue), неупорядоченный лес пространств имен NSNode, неупорядоченный лес атрибутов AttrNode, неупорядоченный лес дочерних узлов и ссылку на тип узла (экземпляр типа Def_Type):

```
elemNode   :(QNameValue, {NSNode} , {AttrNode}
, [ElemNode | ValueNode
      J PINode | CommentNode | InfoItemNode |
RefNode],
      Def_Type}->ElemNode
```

Второй конструктор принимает в качестве параметров неупорядоченное множество пространств имен NSNode, упорядоченный лес узлов, который включает атрибуты и дочерние узлы элемента, а также ссылку на тип узла. В упорядоченном лесу узлов атрибуты должны предшествовать всем другим узлам:

```
elemNode   :(QNameValue, {NSNode} , [AttrNode |
ElemNode | ValueNode  | PINode | CommentNode |
InfoItemNode j RefNode], Def_Type)-> E1emNode
```

Для элементов предусмотрены следующие семь функций доступа:

```
name    :ElemNode -> QNameValue
children   :ElemNode -> [ElemNode | ValueNode |
PINode |
       CommentNode j  InfoItemNode | RefNode]
attributes :ElemNode -> {AttrNode}
namespaces :ElemNode -> {NSNode}
type    :ElemNode -> DefJType
parent  :ElemNode -> ElemNode | DocNode | NaR
nodes   :ElemNode -> [AttrNode | ElemNode |
ValueNode | PINode | CommentNode | InfoItemNode
j RefNode]
```

Первые пять функций доступа возвращают составные части узла ElemNode; функция parent возвращает ссылку на единственный родительский узел узла ElemNode. Если ElemNode является корневым узлом документа, эта функция возвращает ссылку на соответствующий узел DocNode, а если последний не

существует, возвращает значение NaR (Not a Reference – неправильная ссылка). Функция доступа nodes возвращает упорядоченный лес, содержащий атрибуты элемента, за которыми следуют все его дочерние элементы.

Атрибуты.

Узел AttrNode имеет конструктор attrNode, который принимает в качестве параметров имя и значение атрибута:

```
attrNode :(QNameValue, ValueNode) -> AttrNode
```

Для узла attrNode предусмотрены следующие три функции доступа:

```
name  :AttrNode -> QNameValue
value  :AttrNode -> ValueNode
parent :AttrNode -> EleraNode | NaR
```

Пространства имен.

Узел NSNode имеет конструктор nsNode, принимающий в качестве параметров префикс пространства имен (который может быть пустым) и абсолютный идентификатор URI объявляемого пространства имен (который также может быть пустым):

```
nsNode :(string) null, uriReference | null) ->
NSNode
```

Для узла NSNode предусмотрены следующие три функции доступа:

```
prefix :NSNode -> string | null
uri    :NSNode -> uriReference | null
parent :NSNode -> ElemNode
```

Ссылочный узел.

В этой модели данных предусмотрена возможность применения ссылочных узлов RefNode в качестве механизма инкапсуляции идентификаторов узлов в конкретном экземпляре модели данных. Узел RefNode имеет конструктор refNode, который принимает в качестве параметра узел Node:

```
refNode :Node -> RefNode
```

Для узла RefNode предусмотрены следующие две функции доступа:

```
deref  :RefNode -> Node
```

```
parent :RefNode -> ElemNode | NaR
```

Функция доступа deref возвращает узел, на который указывает ссылочный узел; функция доступа parent возвращает уникальный родительский узел дочернего узла RefNode, а если родительский узел не существует, возвращает значение NaR.

Значения.

Узел ValueNode представляет собой непересекающееся объединение значений 14 простых типов:

```
ValueNode :StringValue | BoolValue |
FloatValue | DoubleValue j DecimalValue |
TimeDurValue j RecurDurValue | BinaryValue |
URIRefValue j IDValue | IDREFValue | QNameValue
| ENTITYValue | NOTATIONValue
```

В модели данных предусмотрен соответствующий конструктор для каждого из 14 типов данных XML Schema, например, следующие конструкторы:

```
stringValue    :(string, Def_string,
[InfoItemNode]) -> StringValue
decimalValue   :(decimal, Def_deciraal) ->
DecimalValue
urirefValue    :(uriReference, Def_uriReference)
-> URIRefValue
qnameValue :(uriReference j null, string,
NCName, Def_QName) -> QNameValue
```

Для узла ValueNode предусмотрены также различные функции доступа, на- пример, следующие:

```
type   :ValueNode -> Def_ST, где ST –
соответствующий простой тип;
string :ValueNode -> StringValue, которая
возвращает строковое представление
```

Кроме того, предусмотрено 14 функций доступа в виде isSTValue, где ST обозначает простой тип. Эти функции возвращают значение true, если узел ValueNode относится к указанному типу, например следующим образом:

```
isStringValue :ValueNode -> boolean
```

17.8. Алгебра запросов XML Query

Рабочей группой W3C Query Working Group была предложена алгебра запросов XML, при разработке которой применен опыт использования таких языков, как SQL и OQL. В этой алгебре применяется простая система типов, которая отражает саму суть структур XML Schema. Это позволяет применять в языке статические определения типов, а также обеспечивает последующую оптимизацию запросов. В данном разделе для иллюстрации операторов предложенной алгебры применяется документ XML, приведенный в листинге 2, а также используется упрощенная версия схемы XML Schema, показанной в листинге 6, но для полноты картины предполагается, что в схеме имеется также необязательный элемент NOK, который может встретиться в документе до трех раз подряд. Эти данные и схема могут быть представлены с помощью данной алгебры, как показано в листинге 7.

```
type STAFFLISTGROUP =
    STAFFLIST [ STAFFTYPE (O, *}]
type STAFFTYPE =
    STAFF [
        STAFFNO [String],
        SALARY [Decimal],
        NOK [String] {0, *}
        ©ЬгапсЬКо [String]
        ]
let STAFFLISTO : STAFFLISTGROUP =
    STAFFLIST [
    STAFF [
        STAFFNO ["SL21"],
        SALARY [30000] ,
        NOK ["Mrs Mary White1 1],
        NOK ["Mr Paul White'1],
        fflbranchNo ["B005"]
    ],
    STAFF [
        STAFFNO ["SG37"],
        SALARY [12000],
        NOK ["Mr John Beech"],
        fflbranchNo ["BOC3"]
```

```
    ]
]
```

Листинг 7. Документ и схема, которые представлены с помощью алгебры запросов XML Query Algebra.

В алгебраической модели, показанной в этом листинге, объявлены два типа (STAFFLISTGROUP и STAFFTYPE) и одна глобальная переменная (STAFFLISTO). Тип STAFFLISTGROUP представляет один элемент STAFFLIST, который содержит лес, состоящий из нуля или большего количества элементов STAFFTYPE. Лес представляет собой последовательность атрибутов и элементов, количество которых может составлять от нуля и больше. Элемент STAFFTYPE представляет один элемент STAFF, который содержит два элемента (STAFFNO и SALARY), за ними следует нуль или больше элементов NOK (Next-Of-Kin – ближайший родственник) и один атрибут (branchNo). Переменная STAFFLISTO связана с литеральным значением XML (элемент STAFFLIST с двумя элементами STAFF). Ниже рассматриваются операции, которые могут применяться в описанной алгебраической конструкции.

Проекция.

Следующая операция проекции возвращает все элементы NOK, содержащиеся в элементах STAFF, которые относятся к глобальной переменной STAFFLISTO:

```
STAFFLISTO/STAFF/NOK: NOK [String] { 0, *}
            ==>     NOK ["Mrs Mary White"],
                  NOK ["Mr Paul White1 1],
                  NOK ["Mr John Beech"]
```

В этом операторе вначале указана операция проекции, затем после двоеточия обозначен тип возвращаемого результата (в данном случае элемент NOK типа String с количеством вхождений от нуля и больше), после чего за стрелкой (==>) следует результат операции.

Для доступа к фактическим значениям элементов или атрибутов используется ключевое слово data ( ) , например:

```
STAFFLISTO/STAFF/NOK/data(): String { 0, *}
                  ⇨ "Mrs Mary White",
```

```
                    "Mr Paul White»,
                    "Mr John Beech"
```

Итерация

Как указано выше, одной из важных операций является циклическая обработка элементов документа для последующей реструктуризации документа. Такая цель в рассматриваемой алгебраической модели может быть достигнута с помощью итерации. Например, следующая операция позволяет получить структуру, содержащую только элементы STAFFNO и NOK (в порядке следования, обратном по отношению к первоначальному документу XML):

```
for S in STAFFLISTO/STAFF do
       STAFF [S/NOK, S/STAFFNO] : STAFF [
            NOK [String] { ! , * } ,
            STAFFNO [String]
            ]{o , *}
    ==>     STAFF [
            NOK ["Mrs Mary White"],
            NOK ["Mr Paul White"],
            STAFFNO ["SL21"]
            ] ,
       STAFF [
            NOK ["Mr John Beech"],
            STAFFNO ["SG37")
]
```

Выражение for позволяет обработать в цикле каждый элемент STAFF, который относится к переменной STAFFLISTO, и выполнить привязку переменной s к каждому такому элементу. К каждому связанному элементу применяется внутреннее выражение и формируется новый элемент STAFF, содержащий элементы NOK, за которыми следует элемент STAFFNO.

Выборка.

Для выборки значений, соответствующих некоторому предикату, может применяться выражение WHERE. Например, следующее выражение позволяет получить все элементы STAFF, которые относятся к переменной STAFFLISTO и характеризуются значением заработной платы больше 20 000 фунтов стерлингов, а

затем сформировать новые элементы STAFF, содержащие элементы STAFFNO и SALARY:

```
for S in STAFFLISTO/STAFF do
where S/SALARY/data() > 20000 do
      STAFF [S/STAFFNO, S/SALARY] : STAFF [
        STAFFNO [String],
        SALARY [Decimal]
        ]{ o , * }
    ==> STAFF [
        STAFFNO ["SL21"],
        SALARY [30000]
]
```

Как правило, конструкция WHERE преобразуется в форму IF...THEN...ELSE (с пустой конструкцией ELSE). Выражение for разрешается вкладывать для выборки данных из других повторяющихся элементов. Например, в следующем выражении показано, как выполнить поиск среди элементов NOK:

```
for S in STAFFLISTO/STAFF do
    for N in S/NOK/data() do
    where N = "Mrs Mary White" do
            STAFF [S/STAFPNO] : STAFF [STAFFNO
        [String] ] { 0, *}
        ==>     STAFF [STAFFNO ["SL21"] ]
```

Выражение for может также использоваться в выражении WHERE. Для повышения удобства чтения могут, применяться локальные переменные, например, с помощью локальной переменной приведенное выше выражение преобразуется в следующий вид:

```
for S in STAFFLISTO/STAFF do
    let MrsWhite = (for N in S/NOK/data() do
        where N = "Mrs Mary White" do)
    where MrsWhite do
        STAFF [S/STAFFNO]
```

Соединение.

Рассматриваемая алгебраическая модель позволяет соединять значения, полученные из одного или нескольких документов. Для иллюстрации такой возможности предположим, что в базе данных

имеется второй документ, содержащий сведения о премиях, выплачиваемых сотрудникам компании:

```
type BONUSLIST =
    BONUSES [
        STAFF [
            STAFFNO [String] ,
            " BONUS [Decimal]
let BONUSLISTO: BONUSLIST -
    BONUSES [
        STAFF [
            STAFFNO ["SG37"] ,
            SALARY [1200]
        ]
        STAFF [
            STAFFNO ["SL21"] ,
            BONUS [3000]
        ]
    ]
```

Эти два источника данных (STAFFLISTO и BONUSLISTO) можно соединить друг с другом с учетом значений STAFFNO следующим образом:

```
for S in STAFFLISTO/STAFF do
    for B in BOMJSLISTO/STAFF do
        where S/STAFFNO = B/STAFFNO do
            STAFF [S/STAFFNO, S/SALARY, B/BONUS]
            :
                STAFF [
                    STAFFNO t String] , SALARY
                [Decimal] , BONUS
                    [Decimal]
                ] ( 0, *}
             ==> STAFF [STAFFNO ["SL21"], SALARY
            [30000], BONUS [3000]
                ]
                STAFF [
                    STAFFNO ["SG37"], SALARY
                [12000], BONUS [1200]
                ]
```

В настоящее время порядок следования данных в полученных результатах определяет самое внешнее выражение for, но это правило может измениться после того, как документ XML Query Algebra будет принят в качестве формальной рекомендации W3C.

Сортировка.

В рассматриваемой алгебраической модели предусмотрено выражение sort, позволяющее изменить порядок следования элементов в лесу некоторым указанным способом. Это выражение имеет следующую общую форму:

```
sort VAR in EXP1 by EXP2
```

Здесь VAR определяет диапазон элементов, которые относятся к лесу, заданному выражением `EXP1`, и позволяет упорядочить эти элементы с использованием значения, которое определено выражением `EXP2`. Например, элементы `STAFFLISTO` можно отсортировать в порядке следования элементов `STAFFNO` следующим образом:

```
sort S in STAFFLISTO/STAFF by S/STAFFNO
```

Агрегирование.

В рассматриваемой алгебраической модели поддерживаются типичные функции агрегирования avg, count, rain, max и sum. Например, для выборки данных о сотрудниках, количество ближайших родственников которых превышает одного, можно применить следующее выражение:

```
for S in STAFFLISTO/STAFF do
    where count(S/NOK) > 1 do
        S: STAFF { 0, *}
```

Полное описание этой алгебраической модели выходит за рамки настоящих лекций. Но в заключение следует отметить, что эта алгебра поддерживает также функции и структурную рекурсию.

Язык запросов для XML (XQuery).

Рабочей группой W3C Query Working Group предложен язык запросов для XML, получивший название ***XQuery*** . XQuery разработан на основе одного из языков запросов XML под названием Quilt , который, в свою очередь, включил в себя многие

средства нескольких других языков, таких как XPath, XML-QL, SQL, OQL, Lorel, XQL и YATL . Как и OQL, язык XQuery является функциональным языком, в котором любой запрос представлен в виде выражения. Язык XQuery поддерживает несколько типов выражений, которые могут быть вложенными (поэтому в нем поддерживается понятие подзапроса). В настоящем разделе рассматриваются два аспекта этого языка: обозначения пути и выражения FLWR. Более полное описание языка XQuery выходит за рамки настоящих лекций.

Обозначения пути.

В обозначениях пути XQuery используется сокращенный синтаксис XPath, дополненный новым оператором снятия ссылки и предикатом нового типа, называемым предикатом диапазона. В языке XQuery результат применения обозначения пути представляет собой упорядоченный список узлов, который включает и узлы-потомки этих узлов. Узлы верхнего уровня в результатах применения обозначения пути упорядочиваются в соответствии с их положением в исходной иерархии; в последовательности сверху вниз, слева направо. В качестве примера такого выражения можно указать функцию document (string), которая возвращает корневой узел указанного документа. Запрос может также содержать обозначение пути, начинающееся с одного или двух символов косой черты (/ или //), которые обозначают неявно заданный корневой узел, определение которого зависит от той среды, где выполняется данный запрос.

Операция снятия ссылки (->) может применяться в шагах обозначения пути вслед за атрибутом типа IDREF для получения элемента (элементов), указанного этим атрибутом. За операцией снятия ссылки следует так называемая операция проверки имени, в которой указан целевой элемент (символ звездочки (*) позволяет ссылаться на целевой элемент любого типа).

Выражение FLWR.

Выражение FLWR (произносится как слово "flower") формируется с помощью конструкций FOR, LET, WHERE и RETURN. Как и в любом запросе SQL, эти конструкции должны присутствовать в выражении в указанном порядке. Выражение FLWR позволяет связать значения с одним или несколькими переменными, а затем использовать эти переменные для

формирования результата (как правило, упорядоченного леса узлов).

Конструкции FOR и/или LET служат для привязки значений к одной или нескольким переменным с использованием выражения (например, обозначений пути). Конструкция FOR применяется в тех случаях, когда необходимо выполнять обработку в цикле и связывать каждую заданную переменную с выражением, которое возвращает список узлов. Результат применения конструкции FOR представляет собой список кортежей, каждый из которых определяет привязку для каждой из переменных таким образом, что каждый из связующих кортежей представляет собой перекрестное произведение списков узлов, возвращаемых всеми выражениями. Каждая переменная конструкции FOR может рассматриваться так, как если бы она проходила в цикле по узлам, возвращенным соответствующим ей выражением.

Конструкция LET также позволяет связать одну или несколько переменных с одним или несколькими выражениями, но без циклического перебора, что приводит к получению по одной привязке для каждой переменной. Например, конструкция FOR $S IN /STAFFLIST/STAFF приводит к получению нескольких привязок, каждая из которых связывает переменную $S с одним элементом STAFF из списка STAFFLIST. С другой стороны, конструкция LET $S : = /STAFFLIST/STAFF связывает переменную $S со списком, содержащим все элементы STAFF списка.

Выражение FLWR может содержать несколько конструкций FOR и LET, а каждая из этих конструкций может включать ссылки на переменные, связанные в предыдущих конструкциях. Результатом применения последовательности конструкций FOR и LET является список кортежей связанных переменных в порядке следования элементов входного документа, с которым они связаны; на первом месте стоит первая связанная переменная, за ней следует вторая связанная переменная и т.д. Но если некоторое выражение, используемое в конструкции FOR, является неупорядоченным (например, в связи с тем, что оно содержит функцию DISTINCT), то кортежи, вырабатываемые последовательностью FOR/LET, также являются неупорядоченными.

Необязательная конструкция WHERE позволяет задать одно или несколько условий для ограничения количества связующих кортежей, вырабатываемых конструкциями FOR и LET. Переменные, связанные в конструкции FOR и представляющие отдельный узел, обычно используются в скалярных предикатах, таких как $S/SALARY > 10000. С другой стороны, переменные, связанные в конструкции LET, могут представлять списки узлов и поэтому чаще всего используются в таком предикате, предназначенном для обработки списка, как AVG($S/SALARY) > 20000. Следует отметить, что конструкция WHERE сохраняет упорядочение связующих кортежей, выработанных конструкциями FOR и LET.

Контрольные вопросы:

1. Как в XML хранятся данные?
2. Какие виды данных можно хранить в данные в XML?
3. Для чего нужно XSLT?
4. Что такое SGML?
5. Как делаются запросы на XPATH?
6. Можно ли производить соединение и объединение?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p.
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p.

МАТЕРИАЛЫ ПРАКТИЧЕСКИХ ЗАНЯТИЙ

Практическое занятие 1. Установка MySQL.

Цель практической работы:

Научиться производить установку СУБД MySQL на операционные системы семества WINDOWS.

Задание:

1. Скачайте необходимые дистрибутивы с официального сайта http://mysql.com (сервер, драйвер ODBC, пакет Workbench).
2. Следуйте следующим инструкциям для установки MySQL http://dev.mysql.com/doc/refman/5.7/en/windows-installation.html
3. После установки произведите первую настройку сервера (не забудьте добавить процесс mysqld в список исключений файервола).

Требования к отчету:

Отчет должен содержать следующие пункты:

1. Титульный лист
2. Название практической работы
3. Цель практической работы
4. Задание
5. Скриншоты пошаговой установки программы (скриншиты выполняются вместе с рабочим столом ОС Windows)
6. Выводы по работе.

Контрольные вопросы:

1. В чем заключаются сложности установки MySQL?
2. Какие параметры необходимо указывать при первом запуске MySQL?
3. В чем разница между запуском в режиме службы и обычным запуском?

Литература:

1. К. Дж. Дейт. – Введение в системы баз данных изд. Вильямс, 1328 стр. 2006 г.
2. Jeffrey D. Ullman – Principles of database systems 3rd Edition – Stanford University Computer Science Press. – 2006 – 1137 p.
3. Маллинс Крейг С. – Администрирование баз данных. Полное справочное руководство по методам и процедурам. Пер. с англ. Издательство: Кудиц-Образ. Год издания: 2003
4. Люк Веллинг, Лора Томсон – MySQL. Учебное пособие – Вильямс 2005 г.

Практическое занятие 2. Настройка и конфигурирование MySQL. Подключение к MySQL. Администрирование MySQL.

Цель практической работы:

Цель работы научиться работать в среде СУБД MySQL и использовать различные инструменты по управлению базами данных.

Задание:

1. Прочитать руководство по работе с инструментами MySQL Workbench по адресу: http://dev.mysql.com/doc/refman/5.7/en/tutorial.html
2. Выполнить подключение к серверу
3. Выполнить несколько простых запросов
4. Создать базу данных и вырать ее с помощью команды USE.
5. Просмотреть таблицы и информацию о базе данных information_schema.
6. Настроить подключение к MySQL по руководству: http://dev.mysql.com/doc/workbench/en/wb-getting-started-tutorial-create-connection.html
7. Установить сервер APACHE и настроить логирование сервера в базу данных на MySQL.

Требования к отчету:

Отчет должен содержать следующие пункты

1. Титульный лист
2. Название практической работы
3. Цель практической работы

4. Задание
5. Скриншоты пошаговой выполения всех пунков задания.
6. Выводы по работе.

Контрольные вопросы:

1. Какие параметры необходимо указать при подключении к MySQL серверу?
2. Для чего нужно команда USE?
3. Как используется команда show?
4. Что можно делать в инструменте Workbench?

Литература:

1. К. Дж. Дейт. – Введение в системы баз данных изд. Вильямс, 1328 стр. 2006 г.
2. Jeffrey D. Ullman – Principles of database systems 3rd Edition – Stanford University Computer Science Press. – 2006 – 1137 p.
3. Маллинс Крейг С. – Администрирование баз данных. Полное справочное руководство по методам и процедурам. Пер. с англ. Издательство: Кудиц-Образ. Год издания: 2003
4. Люк Веллинг, Лора Томсон – MySQL. Учебное пособие – Вильямс 2005 г.

Практическое занятие 3. Запросы к одной таблице (SELECT, INSERT, UPDATE, DELETE).

Цель практической работы:

Научиться разрабатывать базу данных и выполнять простые запросы `SELECT`, `INSERT`, `UPDATE`, `DELETE`.

Задание:

Представленная ниже структура БД позволяет вести учет сотрудников, работающих на предприятии, а также хранить все выданные заработные платы с их расшифровкой по доходным и расходным статьям (например, оклад, районный коэффициент, подоходный налог и т.д.).

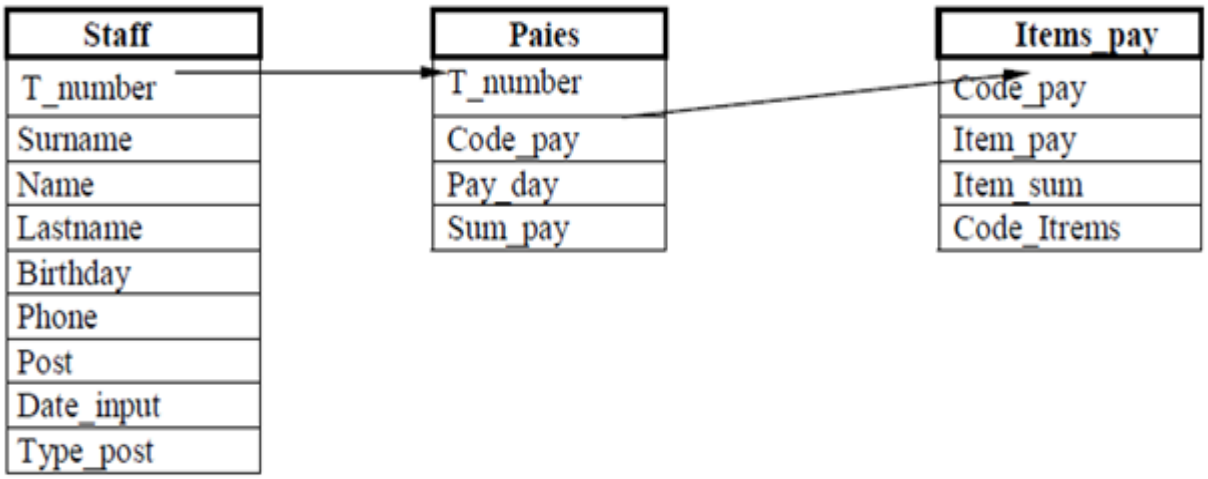


Рис. 3.1. Фрагмент базы данных «Заработная плата».

Связь между таблицами осуществляется с помощью следующих пар полей с типом связи «один-ко-многим» соответственно:

1. Staff.T_number- Paies.T_number.
2. Paies.Code_pay - Items_pay.Code_pay.

Таблица 3.1. Список сотрудников (таблица Staff)

Название поля	**Тип поля**	**Описание поля**
T_number	Integer	Табельный номер сотрудника (уникальный)
Surname	Character	Фамилия сотрудника
Name	Character	Имя сотрудника
Lastname	Character	Отчество сотрудника
Birthday	Date	Дата рождения сотрудника
Phone	Character	Контактный телефон сотрудника
Post	Character	Должность сотрудника
Type_post	Character	Тип сотрудника (ИТР, служащий, рабочий)
Date_input	Date	Дата устройства на работу

Таблица 3.2. Таблица учета выданной зарплаты (таблица Paies).

Название поля	**Тип поля**	**Описание поля**
T_number	Integer	Табельный номер сотрудника, получающего зарплату
Code_pay	Integer	Код выданной зарплаты (уникальный)

Pay_day	Date	Дата выдачи зарплаты
Sum_pay	Numeric	Общая сумма зарплаты на руки

Таблица 3.2. Таблица расшифровки каждой зарплаты по статьям (таблица Items_pay).

Название поля	**Тип поля**	**Описание поля**
Code_pay	Integer	Код выданной зарплаты
Item_pay	Character	Название статьи, по которой начисляют зарплату (как доход, так и расход)
Item_sum	Numeric	Сумма на получение или на вычет из зарплаты
Code_Itrems	Integer	Ключевое поле таблицы

Таблица 3.4. Пример заполнения таблицы Staff.

T_number	Surname	Name	Lastname	Birthday	Phone	Post	Type_post	Date_input
1	Иванов	Иван	Петрович	12.01.1971	124563	Бухгалтер	Служащий	12.04.2000
2	Сидоров	Василий	Михайлович	14.06.1954	451263	Начальник отдела кадров	ИТР	14.11.1999
3	Васильков	Петр	Аркадьевич	14.06.1981	145236	Специалист отдела кадров	Служащий	30.11.2000
67	Артемьев	Иван	Васильевич	05.12.1970	365462	Главный инженер	ИТР	10.02.1998
4	Соянов	Савел	Игнатьевич	15.05.1981	121212	Строитель	Рабочий	25.06.1980
11	Ушаков	Виктор	Семенович	30.05.1970	156462	Бухгалтер	Рабочий	18.11.2003
15	Иванова	Анна	Михайловна	12.03.1940	145214	Строитель	Служащий	12.11.1979

Таблица 3.5. Пример заполнения таблицы Paies

T_number	Code_pay	Pay_day	Sum_pay
1	1	01.01.2003	2544.00
1	2	01.02.2003	4521.00
1	3	01.03.2003	12542.00
2	4	01.01.2003	1452.00
2	5	01.02.2003	2145.00
2	6	01.03.2003	2135.00
3	7	01.01.2003	4511.00
3	8	01.02.2003	1542.00
3	9	01.03.2003	1542.00
4	10	01.03.2003	2456.00

Таблица 3.6. Пример заполнения таблицы Items_pay

Code_pay	Item_pay	Item_sum	Code_Items
1	Премия	124.00	1
1	Налог	-451.00	2
1	Оклад	1457.00	3
1	Поощрение	4512.00	4
1	Оплата учебы	145.00	5
2	Оклад	4656.00	6
2	Налог	-415.00	7
2	Поощрение	326.00	8
3	Оклад	1654.00	9
3	Премия квартальная	1213.00	10
10	За бездетность	-154.00	11
10	Оклад	1456.00	12
10	Премия разовая	1245.00	13
10	Налог подоходный	-452.00	14

1. Создать базу данных с помощью запросов SQL в среде MySQL Workbench.
2. Заполнить таблицы данными, приведенные в таблицах 3.4, 3.5 и 3.6 через команды `INSERT`.
3. Вывести все сведения о сотрудниках из таблицы Staff и отсортировать результат по табельному номеру.
4. Вывести список фамилий, имен, отчеств сотрудников, их должности, отсортировать результат по названиям должностей по возрастанию по фамилиям по убыванию.

5. Выбрать из таблицы Paies табельные номера сотрудников и даты получения зарплат и отсортировать результат по дате по убыванию.
6. Вывести все сведения о сотрудниках из таблицы Staff таким образом, чтобы в результате порядок столбцов был следующим: Name, Lastname, Surname, Post, Date_input, Phone, Birthday, T_number, Type_post.
7. Выбрать все поля из таблицы Paies таким образом, чтобы в результате порядок столбцов был следующим: Sum_pay, Pay_day, T_number, Code_pay.
8. Вывести список сотрудников с должностью, название которой начинается на ‘главный’.
9. Вывести список сотрудников и их должности, которые не являются служащими.
10. Вывести список сотрудников, которые не являются бухгалтерами, и их даты поступления на работу.
11. Вывести список сотрудников, которые были устроены на работу в период с 12.03.2000 по 15.06.2000, и их должности.
12. Вывести список сотрудников и их телефоны, значения которых находятся в диапазоне с 111111 по 222222.
13. Вывести список сотрудников с должностями ‘начальник отдела кадров’, ‘специалист отдела кадров’, ‘операционист отдела кадров’.
14. Вывести список сотрудников с табельными номерами 4, 67, 45, 77.
15. Вывести неповторяющийся список статей в зарплате, которые начинаются на букву ‘н’.
16. Вывести список сотрудников, отчества которых содержат сочетание букв ‘ва’.
17. Выбрать неповторяющийся список должностей, у которых значение оканчивается на ‘ль’.
18. Вывести список сотрудников с датами рождения 01.01.1950 – 01.01.1960 или табельными номерами из диапазона 10 – 150.
19. Перевести всех сотрудников в статус ‘ИТР’, у которых название должности начинается с ‘главный’.

20. Перевести всех сотрудников в статус ‘почетный пенсионер’, а значение должности удалить, если стаж их работы больше 20 лет и возраст больше 60 лет.
21. Изменить значение Post на ‘нет сведений’, если значение поля является пустым.
22. Удалить из таблицы всех сотрудников, у которых возраст больше 80 лет.
23. Удалить из таблицы Статьи зарплат (таблица Items_pay) все записи, у которых в поле названия статьи зарплаты = ‘не известно’.
24. Удалить из таблицы зарплат все записи, в которых значения полей сумма зарплаты и табельный номер сотрудника равны 0.

Требования к отчету:

1. Титульный лист
2. Название практической работы
3. Цель практической работы
4. Задание
5. Скриншоты пошаговой выполения создания и заполнения базы данных.
6. Текст запроса на языке SQL и скриншоты вывода команд SQL.
7. Выводы по работе.

Контрольные вопросы:

1. Какой синтаксис у команды `CREATE TABLE`?
2. Какие типы данных можно использовать при работе с таблицами?
3. Как выполняется выборка данных?
4. Как производится сортировка данных?
5. Как производится отбор данных с помощью предиката `WHERE`?
6. Зачем используются `BETWEEN` и `LIKE`?
7. Как указываются несколько условий отбора данных в команде `SELECT`?
8. Как производится изменение данных?
9. Как производится удаление данных?

Литература:

1. К. Дж. Дейт. – Введение в системы баз данных изд. Вильямс, 1328 стр. 2006 г.
2. Jeffrey D. Ullman – Principles of database systems 3rd Edition – Stanford University Computer Science Press. – 2006 – 1137 p.
3. Мартин Грабер – Введение в SQL – изд. Лори, 382 стр. 1992 г.
4. Маллинс Крейг С. – Администрирование баз данных. Полное справочное руководство по методам и процедурам. Пер. с англ. Издательство: Кудиц-Образ. Год издания: 2003

Практическое занятие 4. Запросы к нескольким таблицам (JOIN, UNION и т.д.).

Цель практической работы:

Научить производить запросы к нескольким таблицам, используя операторы соединения, пересечения и др.

Задание:

1. Создать заново базу данных, описанную в задании из практической работы №3.
2. Вывести список фамилий, имен, отчеств сотрудников (поля Surname, Name, Lastname), а также значения их заработных плат (поле Sum_pay) и даты получения (поле Sum_pay).
3. Вывести табельные номера, даты получения зарплаты и ее расклад по статьям, результат отсортировать по табельному номеру сотрудника.
4. Вывести список фамилий и табельных номеров сотрудников, а также значения их заработных плат и даты получения с раскладкой каждой зарплаты по статьям.
5. Вывести список сотрудников и размеры полученных зарплат за период 01.01.2003 по 01.03.2003.
6. Вывести список сотрудников, получающих одну из следующих надбавок к зарплате: ‘премию’, ‘оплату учебы’, ‘поощрение’.
7. Вывести всех сотрудников, которые получили зарплату 15.03.2003 в размере от 2000 до 3000 руб.

8. Вывести НЕПОВТОРЯЮЩИЙСЯ список табельных номеров и имен сотрудников с табельными номерами 12 – 30 или с зарплатами, превысившими размер 5000 руб.
9. Вывести неповторяющийся список всех сотрудников, у которых размер зарплаты составил от 2000 до 3000 руб.
10. Вывести коды зарплат, в которых была статья вычетов ‘за бездетность’.
11. Вывести неповторяющийся список всех сотрудников, в которых была в зарплате статья вычетов ‘за бездетность’.
12. Вывести список всех сотрудников, их табельные номера, даты и суммы получения зарплаты на руки и зарплаты, если бы у них не брали налог ‘за бездетность’.
13. Вывести список сотрудников и суммарную зарплату каждого.

Требования к отчету:

1. Титульный лист
2. Название практической работы
3. Цель практической работы
4. Задание
5. Скриншоты пошаговой выполения создания и заполнения базы данных.
6. Текст запроса на языке SQL и скриншоты вывода команд SQL.
7. Выводы по работе.

Контрольные вопросы:

1. Как производится соединение?
2. Какие виды соединений вы знаете?
3. Как соединяются более трех таблиц?
4. Зачем нужен оператор `JOIN`?

Литература:

1. К. Дж. Дейт. – Введение в системы баз данных изд. Вильямс, 1328 стр. 2006 г.
2. Jeffrey D. Ullman – Principles of database systems 3rd Edition – Stanford University Computer Science Press. – 2006 – 1137 p.

3. Мартин Грабер – Введение в SQL – изд. Лори, 382 стр. 1992 г.
4. Маллинс Крейг С. – Администрирование баз данных. Полное справочное руководство по методам и процедурам. Пер. с англ. Издательство: Кудиц-Образ. Год издания: 2003

Практическое занятие 5. Выполнение более сложных запросов.

Цель практической работы:

Научиться работать с запросами, содержащими подзапросы и групповые операции.

Задание:

1. Вывести среднюю зарплату, которая когда-либо выдавалась на предприятии.
2. Вывести список сотрудников и суммарную зарплату каждого.
3. Вывести среднюю зарплату каждого сотрудника за прошедший год.
4. Вывести количество сотрудников по каждой должности.
5. Вывести дату устройства на работу самого первого и последнего сотрудника.
6. Вывести список сотрудников и суммарную зарплату каждого, которую поместить в поле с названием Itog.
7. Вывести список всех сотрудников, их табельные номера, даты и суммы получения зарплаты на руки и зарплаты, если бы у них не брали ‘подоходный налог’, результат поместить в столбец Sum_With_Nalog.
8. Объединить данные фамилии, имена, отчества в одном столбце с названием FIO.
9. Объединить данные фамилии, имена, отчества и названия должности в одном столбце с названием FIO_Post.
10. Вывести список сотрудников, получающих одну из следующих надбавок к зарплате: ‘премию’, ‘оплату учебы’, ‘поощрение’, и коды их зарплат.
11. Вывести неповторяющийся список сотрудников, которые получали премию.

12. Вывести список сотрудников, которые ни разу не получали зарплаты.
13. Вывести список сотрудников, у которых размер зарплаты не меньше 3000 руб.
14. Вывести список сотрудников и даты с размерами полученных зарплат, которые превысили средний размер их же зарплат

Требования к отчету:

1. Титульный лист
2. Название практической работы
3. Цель практической работы
4. Задание
5. Текст запроса на языке SQL и скриншоты вывода команд SQL.
6. Выводы по работе.

Контрольные вопросы:

1. Что такое подзапрос?
2. Как выполняются группировка?
3. Какие агрегатные функции вы знаете?
4. Сколько раз выполняются подзапросы?
5. Зачем нужны операторы `IN`, `ANY`, `ALL`?

Литература:

1. К. Дж. Дейт. – Введение в системы баз данных изд. Вильямс, 1328 стр. 2006 г.
2. Jeffrey D. Ullman – Principles of database systems 3rd Edition – Stanford University Computer Science Press. – 2006 – 1137 p.
3. Мартин Грабер – Введение в SQL – изд. Лори, 382 стр. 1992 г.
4. Маллинс Крейг С. – Администрирование баз данных. Полное справочное руководство по методам и процедурам. Пер. с англ. Издательство: Кудиц-Образ. Год издания: 2003

Практическое занятие 6. Инсталляция Microsoft SQL Server.

Цель практической работы:

Научиться производить установкеу сервера Microsoft SQL Server.

Задание:

1. Ознакомиться с руководством по установке сервера Microsoft SQL Server.
2. Скачать необходимый дистрибутив по адресу или использовать для скачивания любой подходящий для этого дистрибутив.
3. Установить SQL Server.

Требования к отчету:

Отчет должен содержать следующие пункты:

1. Титульный лист
2. Название практической работы
3. Цель практической работы
4. Задание
5. Скриншоты пошаговой установки программы (скриншиты выполняются вместе с рабочим столом ОС Windows)
6. Выводы по работе.

Контрольные вопросы:

1. В чем заключаются сложности установки Microsoft SQL Server?
2. Какие службы Microsoft SQL Server стартуют при запуске WINDOWS?
3. В чем разница default и named instance?

Литература:

1. К. Дж. Дейт. – Введение в системы баз данных изд. Вильямс, 1328 стр. 2006 г.
2. Jeffrey D. Ullman – Principles of database systems 3rd Edition – Stanford University Computer Science Press. – 2006 – 1137 p.

3. Мартин Грабер – Введение в SQL – изд. Лори, 382 стр. 1992 г.
4. Маллинс Крейг С. – Администрирование баз данных. Полное справочное руководство по методам и процедурам. Пер. с англ. Издательство: Кудиц-Образ. Год издания: 2003

Практическое занятие 7. Запуск, остановка MySQL Server, работа с файлами.

Цель практической работы:

Научиться настраивать параметры сервера MySQL и управлять основными файлами MySQL.

Задание:

1. Ознакомиться с руководством сервера MySQL по настройке файла my.ini.
2. Настройить колировку в формате UTF-8 для возможночти работать с кирилистическими символами.
3. Увеличить размер кэша, используемого для выполнения запросов.
4. Изменить директорию сохранения файлов баз данных.
5. Прочитать руководство по сбросу пароля пользователя root.
6. Назначить новый пароль пользователя root.

Отчет:

Отчет должен содержать следующие пункты:

1. Титульный лист
2. Название практической работы
3. Цель практической работы
4. Задание
5. Скриншоты изменений в настройках.
6. Скриншоты выполнения сброса пароля пользователя root.
7. Выводы по работе.

Контрольные вопросы:

1. Зачен нужен фай my.ini?

2. Какой режим необходимо запустить для сброса пароля root пользователя?
3. Какая команда сбрасывает пароль пользователя root?

Литература:

1. К. Дж. Дейт. – Введение в системы баз данных изд. Вильямс, 1328 стр. 2006 г.
2. Jeffrey D. Ullman – Principles of database systems 3rd Edition – Stanford University Computer Science Press. – 2006 – 1137 p.
3. Маллинс Крейг С. – Администрирование баз данных. Полное справочное руководство по методам и процедурам. Пер. с англ. Издательство: Кудиц-Образ. Год издания: 2003

Практическое занятие 8. Управление пользователями.

Цель практической работы:

Научиться создавать, удалять пользователя, а также назначать и убирать пользователям права.

Задание:

1. Создать нового пользователя для схемы базы данных «Сотрудники».
2. Дать разрешение пользователю на чтение таблиц «Staff», «Paies», «Items_Pay».
3. Подключиться под новым пользователем к базе данных.
4. Проверить привелегии, данные пользователю.
5. Расширить привелегии новому пользователю, дав разрешение на изменение структуры таблицы «Staff».
6. Добавить 2 столбца в таблицу «Staff».
7. Отобрать все привелегии у нового пользователя.

Отчет:

Отчет должен содержать следующие пункты:

1. Титульный лист
2. Название практической работы
3. Цель практической работы
4. Задание

5. Запрос на создание пользователя.
6. Скриншот подключения нового созданного пользователя.
7. Команду выполнения наделения правами пользователя.
8. Скриншоты выполнения измения таблицы Staff.
9. Выводы по работе.

Контрольные вопросы:

1. Как производится создание пользователя?
2. Какая команды назначает права пользователю?
3. Что произойдет если пользователю не назначить права CONNECT?
4. Как производится изменение паролья пользователя?
5. Как отобрать у пользователя права?
6. Какие видами прав можно наделять пользователя?
7. Что такое роль?

Литература:

1. К. Дж. Дейт. – Введение в системы баз данных изд. Вильямс, 1328 стр. 2006 г.
2. Jeffrey D. Ullman – Principles of database systems 3rd Edition – Stanford University Computer Science Press. – 2006 – 1137 p.
3. Маллинс Крейг С. – Администрирование баз данных. Полное справочное руководство по методам и процедурам. Пер. с англ. Издательство: Кудиц-Образ. Год издания: 2003
4. Люк Веллинг, Лора Томсон – MySQL. Учебное пособие – Вильямс 2005 г.

Практическое занятие 9. Создание резервных копии и восстановление.

Цель практической работы:

Получить практические навыки использования SQL скриптов для восстановления базы данных MySQL

Задание:

Используя скачанный архив по адресу
http://195.158.2.210/employees_db-code-1.0.6.rar

Необходимо восстановить базу данных employee для последующих практических работ.

Для этого необходимо в режиме администратора запустить консольный обработчик команд CMD.

Затем перейти в папку с разархивированными данными с помощью команды **CD “путь к папке”**

Например **CD “C:\Downloads\”**

Затем найти где установлен MySQL server. По умолчанию он устанавливается в папку C:\Program Files\MySQL\My sql server 5.6\bin\mysql.exe

Запустить команду выполнения скрипта **“Путь к mysql” –u root –p –t < employees.sql**

Например “C:\Program Files\MySQL\My sql server 5.6\bin\mysql.exe” –u **root –p –t < employees.sql**

После ввода команды будет запрошен пароль администратора (данный пароль вы указывал при установке), а при правильном вводе пароля будет запушен скрипт.

По завершении скрипта будет выведено сообщение об удачном выполнении

Отчет:

Отчет должен содержать следующие пункты:

1. Титульный лист
2. Название практической работы
3. Цель практической работы
4. Задание
5. Скриншоты результата выполнения скрипта в командной строке и снимок таблиц базы данных из среды SQL.

P.S. Снимки экрана должны отображать весь рабочий экран, включая панель заголовка окна и панель задач

Контрольные вопросы:

1. Как производится восстановление базы данных?
2. Чем экспорт отличается от восстановления базы данных?
3. В каких случаях необходимо восстанавливать данные, а в каких случаях импортировать?
4. Как производится резервное копирование данных?

Литература:

1. К. Дж. Дейт. – Введение в системы баз данных изд. Вильямс, 1328 стр. 2006 г.
2. Jeffrey D. Ullman – Principles of database systems 3rd Edition – Stanford University Computer Science Press. – 2006 – 1137 p.
3. Маллинс Крейг С. – Администрирование баз данных. Полное справочное руководство по методам и процедурам. Пер. с англ. Издательство: Кудиц-Образ. Год издания: 2003
4. Люк Веллинг, Лора Томсон – MySQL. Учебное пособие – Вильямс 2005 г.

МАТЕРИАЛЫ ЛАБОРАТОРНЫХ ЗАНЯТИЙ

Лабораторная работа 1. Анализ предметной области. Разработка модели «СУЩНОСТЬ-СВЯЗЬ».

Цель лабораторной работы:

Научиться проводить обследование заданной предметной области, разрабатывать логическую модель «СУЩНОСТЬ-СВЯЗЬ».

Краткие теоретические сведения:

Наиболее часто на практике семантическое моделирование используется на первой стадии проектирования базы данных. При этом в терминах семантической модели производится концептуальная схема базы данных, которая затем вручную преобразуется к реляционной (или какой-либо другой) схеме. Этот процесс выполняется под управлением методик, в которых достаточно четко оговорены все этапы такого преобразования.

Семантическое моделирование представляет собой моделирование структуры данных, опираясь на смысл этих данных. В качестве инструмента семантического моделирования используются различные варианты диаграмм сущность-связь (ER – Entity-Relationship).

Первый вариант модели сущность-связь был предложен в 1976 г. Питером Пин-Шэн Ченом. В дальнейшем многими авторами были разработаны свои варианты подобных моделей (нотация Мартина, нотация IDEF1X, нотация Баркера и др.). Кроме того, различные программные средства, реализующие одну и ту же нотацию, могут отличаться своими возможностями. По сути, все варианты диаграмм сущность-связь исходят из одной идеи – рисунок всегда нагляднее текстового описания. Все такие диаграммы используют графическое изображение сущностей предметной области, их свойств (атрибутов), и взаимосвязей между сущностями.

Сущность – это класс однотипных объектов, информация о которых должна быть учтена в модели. Каждая сущность должна

иметь наименование, выраженное существительным в единственном числе.

Экземпляр сущности – это конкретный представитель данной сущности. Экземпляры сущностей должны быть различимы, т.е. сущности должны иметь некоторые свойства, уникальные для каждого экземпляра этой сущности.

Атрибут сущности – это именованная характеристика, являющаяся некоторым свойством сущности. Атрибут сущности служит для уточнения, идентификации, классификации, числовой характеристики или выражения состояния сущности.

Ключ сущности – это неизбыточный набор атрибутов, значения которых в совокупности являются уникальными для каждого экземпляра сущности. Неизбыточность заключается в том, что удаление любого атрибута из ключа нарушается его уникальность.

Связь – это некоторая ассоциация между двумя сущностями. Одна сущность может быть связана с другой сущностью или сама с собою. Эта ассоциация всегда является бинарной и может существовать между двумя разными сущностями или между сущностью и ей же самой

Каждая связь может иметь один из следующих типов связи:

1. «один-к-одному» означает, что один экземпляр первой сущности (левой) связан с одним экземпляром второй сущности (правой). Связь один-к-одному чаще всего свидетельствует о том, что на самом деле мы имеем всего одну сущность, неправильно разделенную на две.
2. «один-ко-многим» или «многие-к-одному» означает, что один экземпляр первой сущности (левой) связан с несколькими экземплярами второй сущности (правой). Это наиболее часто используемый тип связи. Левая сущность (со стороны «один») называется родительской, правая (со стороны «много») – дочерней.
3. «много-ко-многим» означает, что каждый экземпляр первой сущности может быть связан с несколькими экземплярами второй сущности, и каждый экземпляр второй сущности может быть связан с несколькими экземплярами первой сущности. Тип связи много-ко-многим является временным типом связи, допустимым на ранних этапах разработки модели. В дальнейшем этот тип связи должен быть заменен

двумя связями типа один-ко-многим путем создания промежуточной сущности.

Каждая связь может иметь одну из двух модальностей связи:

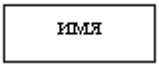
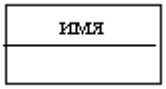
1. Модальность «может» означает, что экземпляр одной сущности может быть связан с одним или несколькими экземплярами другой сущности, а может быть и не связан ни с одним экземпляром (необязательная связь).
2. Модальность «должен» означает, что экземпляр одной сущности обязан быть связан не менее чем с одним экземпляром другой сущности (обязательная связь).

Мы остановились только на самых основных и наиболее очевидных понятиях ER-модели данных. К числу более сложных элементов модели относятся следующие:

- Подтипы и супертипы сущностей. Как в языках программирования с развитыми типовыми системами (например, в языках объектно-ориентированного программирования), вводится возможность наследования типа сущности, исходя из одного или нескольких супертипов. Интересные нюансы связаны с необходимостью графического изображения этого механизма.
- Уточняемые степени связи. Иногда бывает полезно определить возможное количество экземпляров сущности, участвующих в данной связи (например, служащему разрешается участвовать не более, чем в трех проектах одновременно). Для выражения этого семантического ограничения разрешается указывать на конце связи ее максимальную или обязательную степень.
- Каскадные удаления экземпляров сущностей. Некоторые связи бывают настолько сильными (конечно, в случае связи "один-ко-многим"), что при удалении опорного экземпляра сущности (соответствующего концу связи "один") нужно удалить и все экземпляры сущности, соответствующие концу связи "многие". Соответствующее требование "каскадного удаления" можно сформулировать при определении сущности.
- Домены. Как и в случае реляционной модели данных бывает полезна возможность определения потенциально

допустимого множества значений атрибута сущности (домена).

Таблица 1.1 Условные обозначения в нотации Питера Чена.

Элемент диаграммы	Обозначает
	независимая сущность
	зависимая сущность
	родительская сущность в иерархической связи
	Связь
	идентифицирующая связь
	атрибут
	первичный ключ
	внешний ключ (понятие внешнего ключа вводится в реляционной модели данных)
	многозначный атрибут
	получаемый (наследуемый) атрибут в иерархических связях

Связь соединяется с ассоциируемыми сущностями линиями. Возле каждой сущности на линии, соединяющей ее со связью, цифрами указывается класс принадлежности (рис 1.1).

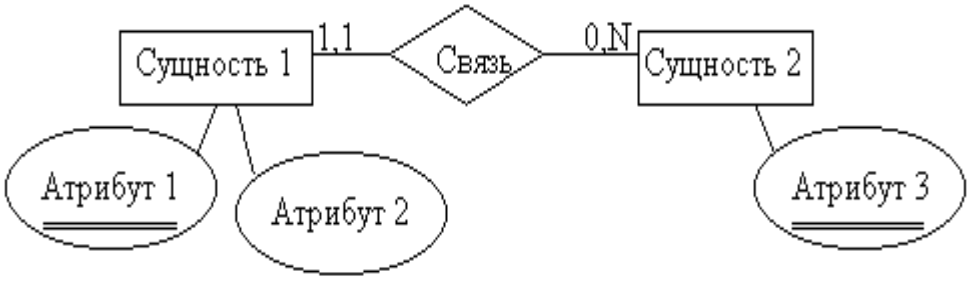
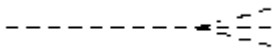
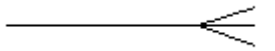


Рис 1.1. Пример диаграммы «сущность-связь» по нотации Чена.

Нотация Баркера.

Сущности обозначаются прямоугольниками, внутри которых приводится список атрибутов. Ключевые атрибуты отмечаются символом # (решетка). Связи обозначаются линиями с именами, место соединения связи и сущности определяет кардинальность связи (таблица 1.2).

Таблица 1.2 Условные обозначения связей в нотации Баркера.

Обозначение	Кардинальность
	0,1
	1,1
	0,N
	1,N

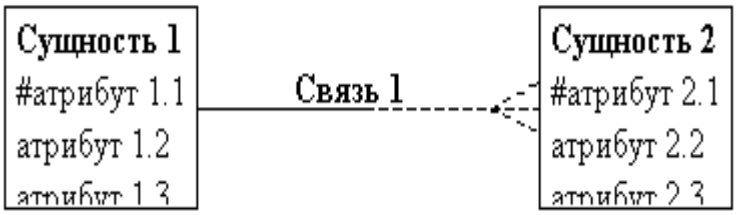


Рис 1.2. Пример диаграммы «сущность-связь» по нотации Баркера.

Для обозначения отношения категоризации вводится элемент "дуга" (рис. 1.3).

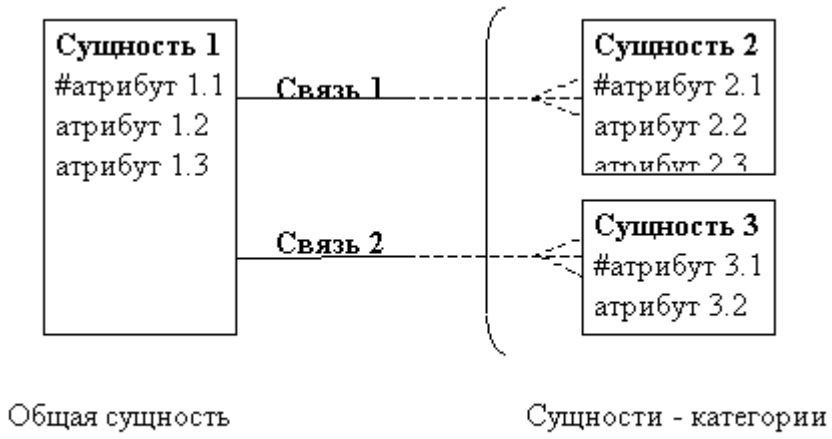
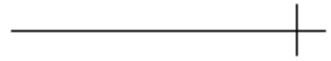
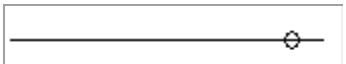


Рис. 1.3. Категоризация в диаграмме «сущность-связь».

Модальность связей в обеих нотациях отображается следующим образом, на конце связи возле сущности ставится условный знак (таблица 1.3).

Таблица 1.3 Модальность связей.

Обозначение	Модальность
	должен
	может

Задание:

1. Выбрать предметную область из списка по вариантам задания, представленным в работе.
2. Написать описание предметной области.
3. Определить сущности из описания предметной области.
4. Определить основные атрибуты сущностей выбранной предметной области.
5. Определить связи между сущностями.
6. Построить схему «сущность-связь предметной области».
7. Подготовить отчет по выполненной работе.

Требования к отчету:

Отчет должен быть выполнен в соответствии с правилами оформления отчетов указанном в приложении 1.

Отчет по выполненной работе должен содержать:

1. Тему лабораторной работы.
2. Цель лабораторной работы.
3. Задание.
4. Описание выбранной предметной области.
5. Список сущностей.
6. Список атрибутов сущностей (выполнить в виде таблицы)
7. Указание связей (выполнить в виде таблицы)
8. Диаграмму «сущность-связь»

Варианты заданий:

1. Библиотека.
2. Отдел продаж.
3. Производство.
4. Кооперативы.
5. Автомастерская.
6. Успеваемость студентов.
7. Поликлиника.
8. Сельскохозяйственные поставки.
9. Городской транспорт.
10. Интернет-провайдер.
11. Домоуправление.
12. Аэропорт.
13. Авиакасса.
14. Магазин компьютерной техники.
15. Деканат.
16. Зоопарк.
17. Судоходство.
18. Автотранспортное предприятие.
19. Программные продукты.
20. Добыча полезных ископаемых.
21. Театр.
22. Справочная служба.
23. Столовая.
24. Туристическая фирма.
25. Строительство.
26. Аукцион.
27. Учет подписки на печатные издания.
28. Гостиница.
29. Учет клиентов в банке.

30. Салон красоты.
31. Налоговая инспекция.
32. Рекламное агентство.
33. Государственная автоинспекция.
34. Налоговая инспекция.
35. Таможенный контроль
36. Отдел кадров.
37. Собственный вариант.

Контрольные вопросы:

1. Что такое сущность?
2. Как определить в каком случае мы имеем дело с сущностью, а в каком со связью, аттрибутом?
3. Какие виды связей встречаются в моделях сущность-связь?
4. Как строится модель сущность-связь?
5. Какая нотация по-вашему более удобна для построения модели сущность-связь?

Литература:

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p

Лабораторная работа 2. Создание реляционной модели базы.

Цель лабораторной работы:

Научиться разрабатывать реляционную модель базы данных, отвечающую всем требованиям модели «сущность-связь». Приобрести навыки по нормализации реляционной модели.

Краткие теоретические сведения:

Недостатки иерархической и сетевой моделей привели к появлению новой, реляционной модели данных, созданной Коддом в 1970 году и вызвавшей всеобщий интерес. Он показал, что любое представление данных сводится к совокупности двумерных таблиц особого вида, известного в математике как

отношение – relation. Реляционная модель была попыткой упростить структуру базы данных. В ней отсутствовали явные указатели на предков и потомков, а все данные были представлены в виде простых таблиц, разбитых на строки и столбцы.

Теоретической основой этой модели стала теория отношений, основу которой заложили два логика – американец Чарльз Содерс Пирс (1839–1914) и немец Эрнст Шредер (1841–1902). В руководствах по теории отношений было показано, что множество отношений замкнуто относительно некоторых специальных операций, то есть образует вместе с этими операциями абстрактную алгебру.

Доменом называется множество атомарных значений одного и того же типа. Наиболее правильной интуитивной трактовкой понятия домена является понимание домена как допустимого потенциального множества значений данного типа.

Вхождение домена в отношение принято называть **атрибутом.**

Кортеж, соответствующий данной схеме отношения, – это множество пар {имя атрибута, значение}, которое содержит одно вхождение каждого имени атрибута, принадлежащего схеме отношения. "Значение" является допустимым значением домена данного атрибута (или типа данных, если понятие домена не поддерживается). Тем самым, степень или "арность" кортежа, т.е. число элементов в нем, совпадает с "арностью" соответствующей схемы отношения. Попросту говоря, кортеж – это набор именованных значений заданного типа.

Отношение – это множество кортежей, соответствующих одной схеме отношения. Иногда схему отношения называют заголовком отношения, а отношение как набор кортежей – телом отношения. На самом деле, понятие схемы отношения ближе всего к понятию структурного типа данных в языках программирования. Было бы вполне логично разрешать отдельно определять схему отношения, а затем одно или несколько отношений с данной схемой.

Схема отношения – это именованное множество пар {имя атрибута, имя домена (или типа, если понятие домена не поддерживается)}

Ключ (key) – это группа из одного или более атрибутов, которая уникальным образом идентифицирует строку.

Классическая реляционная модель данных требует, чтобы данные хранились в так называемых плоских таблицах. Более точно, пользователи и приложения, обращающиеся к данным, должны работать с данными так, как если бы они размещались в таких таблицах. В упрощенном виде плоская таблица – это таблица, каждая ячейка которой может быть однозначно идентифицирована указанием строки и столбца таблицы. Кроме того, в одном столбце все ячейки должны содержать данные одного простого типа.

Для массового пользователя реляционных СУБД можно с успехом использовать неформальные эквиваленты этих понятий:

- база данных – набор таблиц;
- схема базы данных – набор заголовков таблиц;
- отношение – таблица (иногда Файл);
- заголовок отношения – заголовок таблицы;
- тело отношения – тело таблицы;
- кортеж – строка (иногда Запись);
- атрибут – столбец, поле;
- домены и типы данных – типы данные в ячейках таблицы;
- степень (-арность) отношения – количество столбцов таблицы;
- мощность отношения – количество строк таблицы.

Используя эквиваленты понятий реляционной модели, можно изобразить отношение «сотрудник» (рис. 2.1).

	Атрибут 1 **Имя**	Атрибут 2 **Возраст**	Атрибут 3 **Пол**	Атрибут 4 **ТабельныйНомер**
Кортеж 1	Андерсон	21	Ж	010110
Кортеж 2	Деккер	22	М	010100
.	Гловер	22	М	101000
.	Джексон	21	Ж	201100
.	Мур	19	М	111100
.	Наката	20	Ж	111101
Кортеж 7	Смит	19	М	111111

Рис. 2.1. Пример отношения «Сотрудник».

Чтобы таблица была отношением, она должна удовлетворять определенным ограничениям:

1. Во-первых, значения в ячейках таблицы должны быть одиночными – ни повторяющиеся группы, ни массивы не допускаются
2. Строки имеют фиксированное число полей (столбцов) и значений (множественные поля и повторяющиеся группы недопустимы). Иначе говоря, в каждой позиции таблицы на пересечении строки и столбца всегда имеется в точности одно значение или ничего.
3. Строки таблицы обязательно отличаются друг от друга хотя бы единственным значением, что позволяет однозначно идентифицировать любую строку такой таблицы. Порядок строк не имеет значения.
4. Все записи в столбце должны быть одного типа.
5. Каждый столбец имеет уникальное имя; порядок столбцов в таблице несуществен. Наконец, в отношении не может быть двух одинаковых строк, и
6. Полное информационное содержание базы данных представляется в виде явных значений данных, и такой метод представления является единственным.

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. На рис 2.2 изображен пример базы данных, содержащей сведения о подразделениях предприятия и работающих в них сотрудниках, применительно к реляционной модели.

Рис.2.2. База данных о подразделениях и сотрудниках предприятия.

Получение реляционной схемы из ER-диаграммы.

1. Каждая простая сущность превращается в таблицу (отношение). Имя сущности становится именем таблицы.
2. Каждый атрибут становится возможным столбцом с тем же именем. Столбцы, соответствующие необязательным атрибутам, могут содержать неопределенные значения; столбцы, соответствующие обязательным атрибутам – не могут. Если атрибут является множественным, то для него строится отдельное отношение.
3. Компоненты уникального идентификатора сущности превращаются в первичный ключ. Если имеется несколько возможных уникальных идентификаторов, выбирается наиболее используемый. Если в состав уникального идентификатора входят связи, то к числу столбцов первичного ключа добавляется копия уникального идентификатора сущности, находящейся на дальнем конце связи (этот процесс может продолжаться рекурсивно). Для именования этих столбцов используются имена концов связей и/или имена сущностей.
4. Связи «многие к одному» и «один к одному» становятся внешними ключами. Т.е. создается копия уникального идентификатора с конца связи «многие», и соответствующие столбцы составляют внешний ключ.

5. Индексы создаются для первичного ключа (уникальный индекс), а также внешних ключей и тех атрибутов, которые будут часто использоваться в запросах.
6. Если в концептуальной схеме присутствуют подтипы, то возможны два варианта:
   - Все подтипы хранятся в одной таблице, которая создается для самого внешнего супертипа, а для подтипов создаются представления. В таблицу добавляется, по крайней мере, один столбец, содержащий код типа, и он становится частью первичного ключа.
   - Во втором случае для каждого подтипа создается отдельная таблица (для более нижних – представления) и для каждого подтипа первого уровня супертип воссоздается с помощью представления UNION (из всех таблиц подтипов выбираются общие столбцы – столбцы супертипа).
7. Если остающиеся внешние ключи все принадлежат одному домену, т.е. имеют общий формат, то создаются два столбца: идентификатор связи и идентификатор сущности. Столбец идентификатора связи используется для различных связей. Столбец идентификатора сущности используется для хранения значений уникального идентификатора сущности на дальнем конце соответствующей связи. Если результирующие внешние ключи не относятся к одному домену, то для каждой связи, покрываемой дугой исключения, создаются явные столбцы внешних ключей.

Задание:

По построенной в 1-й лабораторной работе модели сущность-связь по правилам перевода произвести построение схемы реляционной модели данных. В схеме указать первичные вторичные ключи, а также, где необходимо, ограничения NOT NULL, UNIQUE

Требования к отчету:

Отчет должен быть выполнен в соответствии с правилами оформления отчетов указанном в приложении 1.

Отчет по выполненной работе должен содержать:

1. Тему лабораторной работы.
2. Цель лабораторной работы.
3. Задание.
4. Схему сущность-связь предыдущей лабораторной работы.
5. Схему реляционной модели данных.

Контрольные вопросы:

1. Что такое реляционная модель?
2. Каковы основные виды ключей в реляционной модели?
3. Какие существуют правила построения реляционной модели из модели сущность-связь?
4. Какие основные требования к «ОТНОШЕНИЯМ»?

Литература

1. Thomas Connolly, Carolyn Begg – Database systems. A practical Approach to Design, Implementation and Management. 4th Edition – Addison Wesley 2005 – 1373p
2. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p

Лабораторная работа 3. Создание базы данных, используя операторы языка определения данных DDL SQL.

Цель лабораторной работы:

Научиться, используя операторы DDL (data definition language) языка запросов SQL (structured query language), создавать и модифицировать базу данных.

Краткие теоретические сведения:

Система управления базами данных (СУБД) – это комплекс языковых и программных средств, предназначенный для создания, ведения и совместного использования БД многими пользователями.

Функции СУБД

1. Определение данных. СУБД должна допускать определения данных (внешние схемы, концептуальную и внутреннюю схемы, соответствующие отображения). Для этого СУБД

включает в себя языковый процессор для различных языков определений данных.

2. Обработка данных. СУБД должна обрабатывать запросы пользователя на выборку, а также модификацию данных. Для этого СУБД включает в себя компоненты процессора языка обработки данных.
3. Безопасность и целостность данных. СУБД должна контролировать запросы и пресекать попытки нарушения правил безопасности и целостности.
4. Восстановление данных и дублирование. СУБД должна обеспечить восстановление данных после сбоев.
5. Словарь данных. СУБД должна обеспечить функцию словаря данных. Сам словарь можно считать системной базой данных, которая содержит данные о данных пользовательской БД, т.е. содержит определения других объектов системы. Словарь интегрирован в определяемую им БД и, поэтому, содержит описание самого себя.
6. Производительность. СУБД должна выполнять свои функции с максимальной производительностью.

Факторы, влияющие на выбор СУБД, можно разделить на ряд групп. Прежде всего, можно выделить факторы, характеризующие саму СУБД и программные средства ее окружения. Другая группа факторов связана с инфраструктурой, сложившейся вокруг каждого из программных продуктов. Третья группа связана с особенностями предполагаемого использования СУБД.

Возможны разные подходы к выбору СУБД. Часто используется упрощенный подход, заключающийся в том, что определяется класс требуемой СУБД, а затем выбирается наиболее популярная на данный момент система данного класса.

Характеристики СУБД рассматриваются с разной степенью детализации в зависимости от стоящих перед проектировщиком задач.

Необходимо определить набор свойств, которым обязательно должна соответствовать выбираемая СУБД (например, обеспечивать должный уровень безопасности, функционировать в определенной операционной среде, поддерживать заданные информационные технологии и др.). Когда говорят о количественных показателях, чаще всего речь идет о времени

выполнения тех или иных операций обработки данных, хотя количественные характеристики этим не ограничиваются.

Среди количественных характеристик можно указать:

- Цена покупки
- Скорость разработки
- Цена сопровождения
- Количество пользователей
- Минимальный, средний и максимальный объем читаемых одним пользователем записей. Их вероятностное распределение.
- Минимальный, средний и максимальный объем изменяемых/добавляемых одним пользователем записей. Их вероятностное распределение.
- Возможности резервного копирования
- Устойчивость к аппаратным сбоям
- Распространенность БД

SQL (Structured Query Language – «язык структурированных запросов») – универсальный компьютерный язык, применяемый для создания, модификации и управления данными в реляционных базах данных. SQL основывается на исчислении кортежей.

DDL (Data Definition Language – «язык описания данных») – это семейство компьютерных языков, используемых в компьютерных программах для описания структуры баз данных.

В состав языка определения данных DDL входят операторы:

1. `CREATE` – создает объектов базы данных
2. `ALTER` – изменяет объект
3. `DROP` –удаляет объект

Стандарт SQL-92 определяет команды для следующих объектов:

1. `ASSERTION` – утверждения для проверки
2. `CHARACTER SET` – набор символов
3. `COLLATION` – правила сортировки для набора символов
4. `DOMAIN` – домен (пользовательского типа данных столбца).
5. `SCHEMA` – схема (именованной группы объектов)
6. `TABLE` – таблица базы данных

7. TRANSLATION – правила преобразования (трансляции) из одного набора символов в другой (используется в операторе TRANSLATE)
8. VIEW – представления данных

Типы данных

ANSI SQL включает в себя следующие типы данных:

Таблица 3.1 Символьные строки.

Тип данных	Описание
CHARACTER(n) или CHAR(n)	строка фиксированной длины в n символов, разделенная пробелами
CHARACTER VARYING(n) или VARCHAR(n)	строка переменной длины с максимальным количеством символов n
NATIONAL CHARACTER(n) или NCHAR(n)	строка фиксированной длины с поддержкой международных кодировок
NATIONAL CHARACTER VARYING(n) или NVARCHAR(n)	строка переменной длины NCHAR

Таблица 3.2. Битовые данные.

Тип данных	Описание
BIT(n)	массив из n битов
BIT VARYING(n)	массив длинной до n битов

Таблица 3.3. Числа.

Тип данных	Описание
INTEGER и SMALLINT	целые числа
FLOAT, REAL и DOUBLE PRECISION	вещественные числа
NUMERIC(precision, scale) или DECIMAL(precision, scale)	вещественное число с указанием в скобках количество знаков до запятой и после запятой

Таблица 3.3. Дата и время.

Тип данных	Описание
DATE	дата (2010-05-30)

TIME	время (14:55:37)
TIMESTAMP	это DATE и TIME соединенные вместе в одной переменной (2010-05-30 14:55:37)

Создание базы данных:

CREATE {DATABASE | SCHEMA} <имя_базы_данных>;

Удаление базы данных:

```
DROP {DATABASE | SCHEMA} <имя_базы_данных>;
```

Создание таблицы:

```
CREATE TABLE [ IF NOT EXISTS ] <имя_таблицы>
(
<имя столбца_1> <тип данных> [ DEFAULT
expression ] [ {NULL | NOT NULL} ],
<имя столбца_2> <тип данных> [ DEFAULT
expression ] [ {NULL | NOT NULL} ]
...
<имя столбца_N> <тип данных> [ DEFAULT
expression ] [ {NULL | NOT NULL} ],
[ CONSTRAINT <имя ограничения> ]
PRIMARY KEY ( <имя столбца_1>, <имя столбца_2>,
... ) |
FOREIGN KEY (<имя столбца_1>, <имя столбца_2>,
... ) REFERENCES <имя_таблицы_2> [ (<имя
столбца_1>, <имя столбца_2>, ... ) ] [ ON
UPDATE {NO ACTION | SET NULL | SET DEFAULT |
CASCADE} ] [ ON DELETE {NO ACTION | SET NULL |
SET DEFAULT | CASCADE} ] |
UNIQUE (<имя столбца_1>, <имя столбца_2>, ... )
|
CHECK ( <условие> ) [ {INITIALLY DEFERRED |
INITIALLY IMMEDIATE} ] [ {NOT DEFERRABLE |
DEFERRABLE} ]
);
```

где

- DEFAULT expression – значение по умолчанию;
- NULL | NOT NULL – разрешается ли пустое поле;
- CONSTRAINT – ограничение
- PRIMARY KEY – первичный ключ

- FOREIGN KEY – вторичный ключ
- ON DELETE – при удалении в родительской таблице
- ON UPDATE – при обновлении в родительской таблице
- NO ACTION – нет действий
- SET NULL – устанавливается значение NULL
- SET DEFAULT – устанавливается значение по умолчанию
- CASCADE – каскадно
- UNIQUE – уникальный
- CHECK – проверка

Например:

```
CREATE TABLE Customer (
 number VARCHAR(40) NOT NULL,
 name VARCHAR(100) NOT NULL,
 ssn VARCHAR(50) NOT NULL,
 age INTEGER NOT NULL,
 CONSTRAINT cust_pk PRIMARY KEY (number),
 UNIQUE ( ssn ),
 CONSTRAINT age_check CHECK (age >= 0 AND age <
200)
);
```

Переименование таблицы:

```
ALTER TABLE <имя_таблицы> RENAME TO
<новое_имя_таблицы>;
```

Переименование столбца:

```
ALTER TABLE <имя_таблицы> RENAME [ COLUMN ]
<имя_столбца> TO <новое_имя_столбца>;
```

Добавление столбца:

```
ALTER TABLE <имя_таблицы> ADD [COLUMN]
<имя_столбца> <тип_данных> [ DEFAULT expression
] [ {NULL | NOT NULL} ];
```

Добавление первичного ключа ограничения к таблице:

```
ALTER TABLE <имя_таблицы> ADD [ CONSTRAINT <имя
ограничения> ]
PRIMARY KEY ( <имя столбца_1>, <имя столбца_2>,
... );
```

Добавление вторичного ключа ограничения к таблице:

```
ALTER TABLE <имя_таблицы> ADD [ CONSTRAINT <имя
ограничения> ]
FOREIGN KEY (<имя столбца_1>, <имя столбца_2>,
...  ) REFERENCES <имя_таблицы_2> [ (<имя
столбца_1>, <имя столбца_2>, ... ) ] [ ON
UPDATE {NO ACTION | SET NULL | SET DEFAULT |
CASCADE} ] [ ON DELETE {NO ACTION | SET NULL |
SET DEFAULT | CASCADE} ] |
```

Добавление уникального поля к таблице:

```
ALTER TABLE <имя_таблицы> ADD [ CONSTRAINT <имя
ограничения> ]
UNIQUE (<имя столбца_1>, <имя столбца_2>, ...
);
```

Добавление проверки столбца к таблице:

```
ALTER TABLE <имя_таблицы> ADD [ CONSTRAINT <имя
ограничения> ]
CHECK ( <условие> ) [ {INITIALLY DEFERRED |
INITIALLY IMMEDIATE} ] [ {NOT DEFERRABLE |
DEFERRABLE} ];
```

Изменение типа данных столбца:

```
ALTER TABLE <имя_таблицы> MODIFY <имя_столбца>
<новый_тип_данных>;
```

Изменение столбца ограничений столбца:

```
ALTER TABLE <имя_таблицы> ALTER [COLUMN]
<имя_столбца> SET default_expr;
ALTER TABLE <имя_таблицы> ALTER [COLUMN]
<имя_столбца> DROP DEFAULT;
```

Изменение удаление столбца:

```
ALTER TABLE <имя_таблицы> DROP [COLUMN]
<имя_столбца>;
```

Удаление ограничения таблицы:

```
ALTER TABLE <имя_таблицы> DROP CONSTRAINT
<имя_ограничения>;
```

Удаление первичного ключа:

```
ALTER TABLE <имя_таблицы> DROP PRIMARY KEY;
```

Удаление таблицы:

```
DROP TABLE [ IF EXISTS ] <имя_таблицы>;
```

Задание:

По построенной во 2-й лабораторной работе схеме реляционной модели на языке определения данных DDL написать запросы, создающие БД предметной области. Запросы упорядочить в соответствии со схемой (сначала таблицы с первичными ключами, а затем со вторичными).

Требования к отчету:

Отчет должен быть выполнен в соответствии с правилами оформления отчетов указанном в приложении 1.

Отчет по выполненной работе должен содержать:

1. Тему лабораторной работы.
2. Цель лабораторной работы.
3. Задание.
4. Схему реляционной модели данных.
5. Основные запросы на языке SQL, создающие таблицы базы данных.

Контрольные вопросы:

1. Какие основные команды относятся к DDL?
2. Каков синтаксис создания таблиц к SQL?
3. Как реализуется изменение имени таблиц, полей?
4. Как можно добавить или удалить ограничения в таблицах?
5. Какие основные объекты можно создавать в базе данных?
6. Что такое тип данных?
7. Какие типы данных вы знаете?

Литература:

1. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p
2. Мартин Грабер. – Введение в SQL. изд. Лори, 382 стр. 1992 г.
3. Jarrod Hollingworth, Bob Swart, Mark Cashman, Paul Gustavson – Borland C++ Builder 6 Developer's Guide 2nd Edition – 2002 – SAMS Publishing – 1128p

Лабораторная работа 4. Создание системы запросов к одной таблице. Сортировка данных, использование предиката WHERE.

Цель лабораторной работы:

Научиться, используя операторы DML (data manipulation language) языка запросов SQL (structured query language), создавать систему запросов к базе данных.

Краткие теоретические сведения:

SQL, или язык структурированных запросов, – на сегодняшний день наиболее важный из языков манипулирования реляционными данными. Он рекомендован Американским национальным институтом стандартов (ANSI) в качестве стандартного языка манипулирования реляционными базами данных и используется как язык доступа к данным многими коммерческими СУБД, включая DB2, SQL/DS, Oracle, INGRES, SYBASE, SQL Server, dBase for Windows, Paradox, Microsoft Access и многие другие. Благодаря своей популярности SQL стал стандартным языком для обмена информацией между компьютерами. Поскольку существует версия SQL, которая может работать почти на любом компьютере и операционной системе, компьютерные системы способны обмениваться данными, передавая друг другу запросы и ответы на языке SQL.

Синтаксические конструкции SQL делятся на четыре основные категории:

- **Идентификаторы.** Представляют собой пользовательские или системные имена объектов баз данных, таких, как база данных, таблица, ограничение в таблице, столбцы таблицы, представления и т. п.
- **Константы.** Представляют собой созданные пользователем или системой строки или значения, не являющиеся идентификаторами или ключевыми словами. Константы могут представлять собой строки, например «hello», числа, например «1234», даты, например «1 января 2002», или булевы значения, например TRUE.
- **Операторы.** Символы, показывающие, какое действие выполняется над одним или несколькими выражениями,

чаще всего в инструкциях DELETE, INSERT, SELECT или UPDATE. Операторы также часто применяются для создания объектов базы данных.

- **Зарезервированные и ключевые слова.** Имеют специальный смысл для обработчика кода SQL. Например, SELECT, GRANT, DELETE или CREATE. Зарезервированные слова (Reserved words), обычно команды и инструкции SQL, нельзя использовать в качестве идентификаторов на данной платформе. Ключевые слова (keywords) - это слова, которые могут стать зарезервированными в будущем.

Язык манипулирования данными.

В основу языка манипулирования данными входят 4 основных оператора:

- `SELECT` – используется для выборки записей из таблиц;
- `INSERT` –используется для добавления записей в таблицу;
- `UPDATE` – используется для обновления записей таблицы;
- `DELETE` – используется для удаления записей из таблицы.

В самой простой форме, команда SELECT просто инструктирует базу данных, чтобы извлечь информацию из таблицы.

Пример базы данных

Для рассмотрения примеров будем использовать базу данных, состоящую из трех таблиц:

Таблица 4.1. Salespeople (Продавцы)

SNUM	SNAME	CITY	COMM
1001	Peel	London	0.12
1002	Serres	San Jose	0.13
1004	Motika	London	0.11
1007	Rifkin	Barcelona	0.15
1003	Axelrod	New York	0.10

Таблица 4.2. Customers (Покупатели)

CNUM	CNAME	CITY	RATING	SNUM
2001	Hoffman	London	100	1001
2002	Giovanni	Rome	200	1003
2003	Liu	San Jose	200	1002

2004	Grass	Berlin	300	1002
2006	Clemens	London	100	1001
2008	Cisneros	San Jose	300	1007
2007	Pereira	Rome	100	1004

Оператор INSERT

Все строки в SQL вводятся с использованием команды модификации INSERT. В самой простой форме, INSERT использует следующий синтаксис:

```
INSERT INTO <имя_таблицы> [(<имя столбца_1>
[, <имя столбца_1> ...])] {VALUES (<значение_1>
[, <значение_2> ...]) | <выражение SELECT>};
```

Так, например, чтобы ввести строку в таблицу Продавцов, вы можете использовать следующее условие:

```
INSERT INTO Salespeople VALUES (1001,
'Peel', 'London', 0.12);
```

Команды INSERT не производят никакого вывода, но ваша программа должна дать вам некоторое подтверждение того что данные были введены.

Вы можете также указывать столбцы, куда вы хотите вставить значение имени. Это позволяет вам вставлять имена в любом порядке. Предположим что вы берете значения для таблицы Заказчиков из отчета выводимого на принтер, который помещает их в таком порядке: city, cname, и cnum, и для упрощения, вы хотите ввести значения в том же порядке:

```
INSERT INTO Customers (city, cname, cnum)
VALUES ('London', 'Honman', 2001);
```

Обратите внимание, что столбцы rating и snum – отсутствуют. Это значит, что эти строки автоматически установлены в значение – по умолчанию. По умолчанию может быть введено или значение NULL или другое значение определяемое как – по умолчанию. Если ограничение запрещает использование значения NULL в данном столбце, и этот столбец не установлен как по умолчанию, этот столбец должен быть обеспечен значением для любой команды INSERT, которая относится к таблице.

Оператор UPDATE

Теперь, вы должны узнать, как изменять некоторые или все значения в существующей строке. Это выполняется командой UPDATE.

```
UPDATE TABLE <имя_таблицы>
   SET <имя столбца_1> = <значение_1> [, <имя столбца_2> = <значение_2> ] ...
   [WHERE <условие>];
```

Эта команда содержит предложение UPDATE, в которой указано имя используемой таблицы и предложение SET которое указывает на изменение, которое нужно сделать для определенного столбца. Например, чтобы изменить оценки всех заказчиков на 200, вы можете ввести

```
UPDATE Customers SET rating = 200;
```

Конечно, вы не всегда захотите указывать все строки таблицы для изменения единственного значения, так что UPDATE может брать предикаты. Вот как например можно выполнить изменение одинаковое для всех заказчиков продавца Peel (имеющего snum=1001):

```
UPDATE Customers SET rating = 200 WHERE snum = 1001;
```

Однако, вы не должны, ограничивать себя модифицированием единственного столбца с помощью команды UPDATE. Предложение SET может назначать любое число столбцов, отделяемых запятыми. Все указанные назначения могут быть сделаны для любой табличной строки, но только для одной в каждый момент времени. Предположим, что продавец Motika ушел на пенсию, и мы хотим переназначить его номер новому продавцу:

```
UPDATE Salespeople SET sname = 'Gibson', city = 'Boston', comm = 0.10 WHERE snum = 1004;
```

Вы можете использовать скалярные выражения в предложении SET команды UPDATE, однако, включив его в выражение пол которое будет изменено. В этом их отличие от предложения VALUES команды INSERT, в котором выражения не могут использоваться; это свойство скалярных выражений – весьма полезна особенность. Предположим, что вы решили удвоить комиссионные всем вашим продавцам. Вы можете использовать следующее выражение:

```
UPDATE Salespeople SET comm = comm * 2;
```

Оператор DELETE

Вы можете удалять строки из таблицы командой модификации – DELETE. Она может удалять только введенные строки, а не индивидуальные значения полей.

```
DELETE FROM table [WHERE <условие>];
```

Обычно, вам нужно удалить только некоторые определенные строки из таблицы. Чтобы определить какие строки будут удалены, вы используете предикат WHERE, так же как вы это делали для запросов. Например, чтобы удалить продавца под номером 1003 из таблицы, вы можете ввести

```
DELETE FROM Salespeople WHERE snum = 1003;
```

Мы использовали поле snum вместо пол sname потому, что это лучшая тактика при использовании первичных ключей, когда вы хотите чтобы действию подвергалась одна и только одна строка. Конечно, вы можете также использовать DELETE с предикатом, который бы выбирал группу строк, как показано в этом примере:

```
DELETE FROM Salespeople WHERE city = 'London';
```

Оператор SELECT

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

```
SELECT [[ALL] | DISTINCT]{ * | <имя_столбца1> [,<имя_столбца2>] ...}
FROM {базовая_таблица | представление} [псевдоним]
      [,{базовая_таблица | представление} [псевдоним]] ...
[WHERE <условие>]
[GROUP BY <имя_столбца1> [,<имя_столбца2>] ... [HAVING ] [,<условие>];
[ORDER BY <имя_столбца1> [,<имя_столбца2>] ...]
```

Если вы хотите видеть каждый столбец таблицы, имеется необязательное сокращение, которое вы можете использовать.

Звездочка (*) может применяться для вывода полного списка столбцов следующим образом:

```
SELECT * FROM Salespeople;
```

Как и при обновлении и удалении строк, можно выводить только строки, соответствующие определенному условию, с помощью предиката WHERE. Например, нам нужно вывести информацию о поставщиках из Лондона:

```
SELECT * FROM Salespeople WHERE city = 'London';
```

Существует возможность выбрать лишь некоторые столбцы из таблицы и в любом порядке. Для этого необходимо после ключевого слова SELECT через запятую указать столбцы и порядок их извлечения из таблицы:

```
SELECT comm, sname, snum FROM Salespeople WHERE city = 'London';
```

Для того чтобы выбрать уникальные неповторяющиеся значения по одному или нескольким столбцам, можно использовать предикат DISTINCT:

```
SELECT DISTINCT city from Customers;
```

Теперь вы знаете несколько способов заставить таблицу давать вам ту информацию, какую вы хотите, а не просто выбрасывать наружу все ее содержание. Вы можете переупорядочивать столбцы таблицы или устранять любой из них. Вы можете решать, хотите вы видеть дублированные значения или нет.

Наиболее важно то, что вы можете устанавливать условие, которое определяет или не определяет указанную строку таблицы из тысяч таких же строк, будет ли она выбрана для вывода.

Условия могут становиться очень сложными, предоставляя вам высокую точность в решении, какие строки вам выбирать с помощью запроса. Именно эта способность решать точно, что вы хотите видеть, делает запросы SQL такими мощными.

Для создания сложных условий используются операторы:

Таблица 4.3. Операторы, использующиеся совместно с предикатом WHERE.

Оператор	Описание
=	равный к
>	больше чем

<	меньше чем
>=	больше чем или равно
<=	меньше чем или равно
<>	не равно
BETWEEN	выбирает значения, лежащие в указанном диапазоне значений.
LIKE	выбирает строковые значения, удовлетворяющие заданному шаблону
IS NULL	выбирает значения, имеющие NULL значения
NOT	логический оператор отрицания
AND	логический оператор И
OR	логический оператор ИЛИ

Например:

```
SELECT * FROM Customers WHERE rating > 200;
```

```
SELECT * FROM Customers WHERE city = " San Jose’ AND rating > 200;
```

```
SELECT * FROM Customers WHERE NOT city = " San Jose’ OR rating > 200;
```

```
SELECT * FROM Salespeople WHERE city = ‘Barcelona’ OR city = ‘London’;
```

```
SELECT * FROM Customers WHERE cname BETWEEN ‘A’ AND ‘G’;
```

```
SELECT * FROM Customers WHERE cname LIKE ‘G%’;
```

```
SELECT * FROM Customers WHERE city IS NULL;
```

Задание:

1. По выбранной предметной области для созданной с помощью SQL базы данных определить основные данные для всех таблиц (не менее 5 записей).

2. С помощью оператора INSERT произвести ввод данных, соблюдая целостность базы данных, т.е. соблюдая все ограничения.
3. Используя оператор SELECT создать и выполнить по 2 запроса, производящих:
   - выборку всех данных из таблиц;
   - выборку части данных из таблиц с измененным порядком следования столбцов с помощью предиката WHERE;
   - выборку данных, используя предикаты BETWEEN, LIKE, DISTINCT, >, <, =, <=, >=, <>, AND, NOT, OR, IS NULL.
   - выборку данных из одной таблицы, отсортированных по любому полю с помощью предиката ORDER BY.
4. Создать и выполнить по 2 запроса с оператором UPDATE, производящих обновление:
   - всех строк таблицы;
   - некоторых строк, удовлетворяющих определенному условию.
5. Создать и выполнить по 2 запроса с оператором DELETE, производящих удаление:
   - всех строк таблицы;
   - некоторых строк, удовлетворяющих определенному условию.

Требования к отчету:

Отчет должен быть выполнен в соответствии с правилами оформления отчетов указанном в приложении 1.

Отчет по выполненной работе должен содержать:

1. Тему лабораторной работы.
2. Цель лабораторной работы.
3. Задание.
4. Схему реляционной базы данных из лабораторной работы №2.
5. Таблицы с данными.
6. Запросы на языке SQL, заполняющие таблицы базы данных приведенными выше данными с помощью оператора INSERT.

7. Список запросов к базе данных. Каждый запрос должен содержать описание запроса (что этот запрос выполняет) и запрос на языке SQL. Все запросы разделить по категориям, которые указаны в задании.

Контрольные вопросы:

1. Какие основные команды входят в DML?
2. Как осуществляется ввод данных?
3. Как производится изменение нескольких строк одним запросом?
4. Какая из команд SQL позволяет удалять данные?
5. Как провильно строится запросы по выборке данных из одной таблицы?
6. Как строятся условия в предикате WHERE?

Литература

1. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p
2. Мартин Грабер. – Введение в SQL. изд. Лори, 382 стр. 1992 г.
3. Jarrod Hollingworth, Bob Swart, Mark Cashman, Paul Gustavson – Borland C++ Builder 6 Developer's Guide 2nd Edition – 2002 – SAMS Publishing – 1128p

Лабораторная работа 5. Создание запросов с использованием GROUP BY и HAVING.

Цель лабораторной работы:

Научиться, используя операторы DML (data manipulation language) языка запросов SQL, создавать сложные запросы, использующие группировку, агрегатные функции, подзапросы, операции соединения, объединения, разности, пересечения.

Краткие теоретические сведения:

Агрегатные функции.

Другим классом операций над отношениями в целом являются операции агрегации значений в столбце. Агрегация – это формирование единственного значения из списка значений, содержащихся в столбце. Примерами агрегации являются сумма

значений и среднее значение в столбце. SQL позволяет не только агрегировать столбцы, но и группировать кортежи отношения по какому-либо критерию, например по конкретному значению из другого столбца, а затем проводить агрегацию в полученных группах.

В SQL есть операторы, которые применяются к столбцу отношения и порождают на нем некоторый итог, или агрегацию.

1. `SUM` – сумма значений в столбце.
2. `AVG` – среднее из всех значений в столбце.
3. `MIN` – минимальное из всех значений в столбце.
4. `MAX` – максимальное из всех значений в столбце.
5. `COUNT` – число значений (включая дубликаты, если только они явным образом не удаляются с помощью DISTINCT).

Эти операторы применяются к выражению со скалярным значением, обычно к имени столбца, в пункте `SELECT`. Только числовые поля могут использоваться с `SUM` и `AVG`. С `COUNT`, `MAX`, и `MIN`, могут использоваться и числовые или символьные поля. Когда они используются с символьными полями, `MAX` и `MIN` будут транслировать их в эквивалент ASCII, который должен сообщать, что `MIN` будет означать первое, а `MAX` последнее значение в алфавитном порядке.

Для рассмотрения примеров будем использовать базу данных, состоящую из трех таблиц:

Таблица 5.1. Salespeople (Продавцы).

SNUM	SNAME	CITY	COMM
1001	Peel	London	0.12
1002	Serres	San Jose	0.13
1004	Motika	London	0.11
1007	Rifkin	Barcelona	0.15
1003	Axelrod	New York	0.10

Таблица 5.2. Customers (Покупатели).

CNUM	CNAME	CITY	RATING	SNUM
2001	Hoffman	London	100	1001
2002	Giovanni	Rome	200	1003
2003	Liu	SanJose	200	1002
2004	Grass	Berlin	300	1002

2006	Clemens	London	100	1001
2008	Cisneros	SanJose	300	1007
2007	Pereira	Rome	100	1004

Таблица 5.3. Orders (заказы).

ONUM	AMT	ODATE	CNUM	SNUM
3001	18.69	10/03/1990	2008	1007
3003	767.19	10/03/1990	2001	1001
3002	1900.10	10/03/1990	2007	1004
3005	5160.45	10/03/1990	2003	1002
3006	1098.16	10/03/1990	2008	1007
3009	1713.23	10/04/1990	2002	1003
3007	75.75	10/04/1990	2004	1002
3008	4723.00	10/05/1990	2006	1001
3010	1309.95	10/06/1990	2004	1002
3011	9891.88	10/06/1990	2006	1001

Применение оператора агрегации со знаком *, т.е. к кортежу в целом, имеет смысл только дня `COUNT`. Бесполезно применять любой другой оператор агрегации более чем к одному столбцу.

Чтобы найти `SUM` всех наших покупок в таблицы заказы, мы можем ввести следующий запрос:

```
SELECT SUM ((amt)) FROM Orders;
```

Функция COUNT несколько отличается от всех. Она считает число значений в данном столбце, или число строк в таблице. Когда она считает значения столбца, она используется с DISTINCT чтобы производить счет чисел различных значений в данном поле. Мы могли бы использовать ее, например, чтобы сосчитать номера продавцов в настоящее время описанных в таблице заказов.

```
SELECT COUNT ( DISTINCT snum ) FROM Orders;
```

Обратите внимание в вышеупомянутом примере, что DISTINCT, сопровождаемый именем поля с которым он применяется, помещен в круглые скобки, но не сразу после `SELECT`, как раньше. Этого использования `DISTINCT` с `COUNT` применяемого к индивидуальным столбцам, требует стандарт ANSI, но большое количество программ не предъявляют к ним такого требования.

Группирование.

Иногда необходимо рассматривать кортежи отношения в виде групп, соответствующих значению одного из столбцов.

Например, предположим, что вы хотите найти наибольшую сумму приобретений, полученную каждым продавцом. Вы можете сделать раздельный запрос для каждого из них, выбрав MAX (amt) из таблицы заказов для каждого значения поля snum. GROUP BY, однако, позволит Вам поместить их все в одну команду:

```
SELECT snum, MAX (amt) FROM Orders GROUP BY snum;
```

Для получения такой таблицы применяется предложение GROUP BY, стоящее в запросе непосредственно за предложением WHERE. За ключевыми словами GROUP BY указывается список группируемых атрибутов. В простейшей ситуации в предложении FROM указано только одно отношение, и его кортежи группируются согласно их значениям в группируемых атрибутах. Любой оператор агрегации, используемый в пункте SELECT, применяется только внутри групп.

Хотя запросы с GROUP BY в общем случае могут содержать в предложении SELECT и группируемые атрибуты, и агрегаты, с технической точки зрения это совсем не обязательно. Например, можно написать:

```
SELECT City FROM Salespeople GROUP BY City;
```

Этот запрос группирует кортежи отношения Salespeople согласно названиям их городов и выдает название города для каждой группы независимо от того, сколько кортежей содержит данное название города. Значит, этот запрос дает такой же результат, как и запрос:

```
SELECT DISTINCT City FROM Salespeople;
```

Предложение GROUP BY можно применять и в запросе, адресованном нескольким отношениям. Такой запрос интерпретируется согласно следующим шагам.

1. Вычисляется отношение R, которое получается из предложений FROM и WHERE, т.е. R – это декартово произведение отношений, указанных в предложении FROM, к которому применен выбор, указанный в предложении WHERE.

2. Кортежи отношения R группируются согласно атрибутам из предложения GROUP BY.
3. Результат выдается в виде атрибутов и агрегатов пункта SELECT так, словно запрос касался отношения R.

Допустим, нужно вывести на печать таблицу, показывающую общую сумму, потраченную каждым из потребителей на заказы. Нужна информация из двух отношений: Customers (cnum, cname, city, raiting, snum) и Orders (amt, cnum). Поэтому мы начинаем с тэта-соединения, сравнивая входящие в них номера сертификатов. В результате получается отношение, в котором каждый кортеж из Customers спарен с кортежами из orders, содержащими все заказы данного покупателя. И наконец, подсчитывается сумма, потраченная в каждой группе.

```
SELECT cname, city, raiting, SUM(Amt) FROM Customers, Orders WHERE Customers.cnum = Orders.cnum GROUP BY cname, raiting;
```

Предложение HAVING.

Предположим, что мы не хотим включать в таблицу примера всех покупателей. Тогда перед группированием можно ограничить кортежи так, чтобы нежелательные группы оказались пустыми. Например, если нужна общая сумма заказов, сделанных только покупателями, имеющими raiting 200:

```
SELECT cname, city, raiting, SUM(Amt) FROM Customers, Orders WHERE Customers.cnum = Orders.cnum AND Raiting = 200 GROUP BY cname, raiting;
```

Однако иногда нужно выбрать группы, основанные на некотором общем свойстве самой данной группы. Тогда за предложением GROUP BY указывается предложение HAVING, состоящее из ключевого слова HAVING, за ним следует условие, касающееся группы.

Например, нужно определить общую сумму заказов только для покупателей, потративших более 1500$. В конец запроса добавляется предложение:

```
HAVING SUM(Amt) > 1500
```

В результате получается запрос, удаляющий из отношения такие группы кортежей, в которых каждый кортеж имеет в компоненте SUM(Amt) значение 1500 и менее.

```
SELECT cname, city, raiting, SUM(amt) FROM Customers, Orders WHERE Customers.cnum = Orders.cnum AND Raiting = 200 GROUP BY cname, raiting HAVING SUM(amt) > 1500;
```

Соединения.

Одна из наиболее важных особенностей запросов SQL – это их способность определять связи между многочислеными таблицами и выводить информацию из них в терминах этих связей, всю внутри одной команды. Этот вид операции называется – соединением, которое является одним из видов операций в реляционных базах данных. Главное в реляционном подходе это связи, которые можно создавать между позициями данных в таблицах. Используя соединения, мы непосредственно связываем информацию с любым номером таблицы, и таким образом способны создавать связи между сравнимыми фрагментами данных. При соединении таблицы, представленные списком в предложении FROM запроса, отделяются запятыми. Предикат запроса может ссылаться к любому столбцу любой связанной таблицы и, следовательно, может использоваться для связи между ними. Обычно, предикат сравнивает значения в столбцах различных таблиц, чтобы определить, удовлетворяет ли WHERE установленному условию.

Предположим, что вы хотите поставить в соответствии вашему продавцу ваших заказчиков в городе, в котором они живут, поэтому вы увидите все комбинации продавцов и заказчиков для этого города. Вы будете должны брать каждого продавца и искать в таблице Заказчиков всех заказчиков того же самого города. Вы могли бы сделать это, введя следующую команду:

```
SELECT Customers.cname, Salespeople.sname, Salespeople.city FROM Salespeople, Customers WHERE Salespeople.city = Customers.city;
```

Так как это поле city имеется и в таблице Продавцов и таблице Заказчиков, имена таблиц должны использоваться как префиксы. Хотя это необходимо только когда два или более полей имеют одно и то же имя, в любом случае это хорошая идея включать имя таблицы в соединение для лучшего понимания и непротиворечивости.

Можно в соответствии со стандартом ANSI использовать ключевое слово JOIN для выполнения соединений. Выше приведенный пример будет выглядеть следующим образом:

```
SELECT Customers.cname, Salespeople.sname, Salespeople.city FROM Salespeople INNER JOIN Customers ON Salespeople.city = Customers.city;
```

Данный вид записи, как вы видите, более удобен тем, что условие соединения отделяется от условия выборки строк из таблицы, т.к. предикат WHERE будет следовать в самом конце запроса.

Связь таблиц покупателей и продавцов через поле snum называется состоянием справочной целостности. Используя соединение можно извлекать данные в терминах этой связи. Например, чтобы показать имена всех заказчиков соответствующих продавцам которые их обслуживают, мы будем использовать такой запрос:

```
SELECT cname, sname FROM Customers, Salespeople WHERE Salespeople.snum = Customers.snum;
```

Это – пример соединения, в котором столбцы используются для определения предиката запроса, и в этом случае, snum столбцы из обеих таблиц, удалены из вывода. И это прекрасно. Вывод показывает, какие заказчики каким продавцом обслуживаются; значения поля snum которые устанавливают связь – отсутствуют.

Вы можете также создавать запросы, объединяющие более двух таблиц. Предположим, что мы хотим найти все заказы заказчиков, не находящихся в тех городах, где находятся их продавцы. Для этого необходимо связать все три наши типовые таблицы:

```
SELECT onum, cname, Orders.cnum, Orders.snum FROM Salespeople, Customers, Orders WHERE Customers.city <> Salespeople.city AND Orders.cnum = Customers.cnum AND Orders.snum = Salespeople.snum;
```

Для соединения таблицы с собой, вы можете сделать каждую строку таблицы, одновременно, и комбинацией ее с собой и комбинацией с каждой другой строкой таблицы.

Вы можете изобразить соединение таблицы с собой, как соединение двух копий одной и той же таблицы. Таблица на самом деле не копируется, но SQL выполняет команду так, как если бы это было сделано. Другими словами, это соединение – такое же, как и любое другое соединение между двумя таблицами, за исключением того, что в данном случае обе таблицы идентичны.

Синтаксис команды для соединения таблицы с собой, тот же что и для соединения многочисленых таблиц, в одном экземпляре. Когда вы соединяете таблицу с собой, все повторяемые имена столбца, заполняются префиксами имени таблицы. Чтобы ссылаться к этим столбцам внутри запроса, вы должны иметь два различных имени для этой таблицы. Вы можете сделать это с помощью определения временных имен называемых переменными диапазона, переменными корреляции или просто – псевдонимами. Вы определяете их в предложении FROM запроса. Это очень просто: вы набираете имя таблицы, оставляете пробел, и затем набираете псевдоним для нее. Имеется пример, который находит все пары заказчиков, имеющих один и тот же самый рейтинг:

```
    SELECT first.cname, second.cname,
first.rating
    FROM Customers first, Customers second
    WHERE first.rating = second.rating;
```

Существуют различные виды соединений: внешнее, внутреннее, левое внешнее, правое внутреннее, полное – описание, которых можно найти в любом справочном пособии по SQL и в списке литературы, приведенной в данном пособии.

Задание:

Для созданной в предыдущих лабораторных работах базы данных разработать систему запросов (минимум по 3 запроса):

1. Содержащих агрегатные функции, с использованием ключевых слов `GROUP BY` и `HAVING`.
2. Производящих выборку данных из двух и более соединенных таблиц.
3. Содержащих подзапросы с использованием ключевых слов `IN`, `ANY`, `ALL`.

Каждый запрос должен снабжаться описанием, чтобы проверить правильность построения запроса.

Требования к отчету:

Отчет должен быть выполнен в соответствии с правилами оформления отчетов указанном в приложении 1.

Отчет по выполненной работе должен содержать:

1. Тему лабораторной работы.
2. Цель лабораторной работы.
3. Задание.
4. Схему реляционной базы данных из лабораторной работы №2.
5. Список запросов к базе данных. Каждый запрос должен содержать описание запроса (что этот запрос выполняет) и запрос на языке SQL. Все запросы разделить по категориям, которые указаны в задании.

Контрольные вопросы:

1. Для чего нужно использовать предикаты `HAVING` и `GROUP BY`?
2. Какие выбы агрегатных функций вы знаете?
3. Можно ли вкладывать агрегатные функции друг в друга?
4. Как можно получить с помощью группировки уникальные значения?
5. Для чего используются операторы `IN`, `ANY` и `ALL`?
6. Какие выды соединений существуют?
7. Каков синтаксис соединения на языке SQL?
8. В чем разница между соединением и декартовым произведением?

Литература:

1. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p
2. Мартин Грабер. – Введение в SQL. изд. Лори, 382 стр. 1992 г.
3. Jarrod Hollingworth, Bob Swart, Mark Cashman, Paul Gustavson – Borland C++ Builder 6 Developer's Guide 2nd Edition – 2002 – SAMS Publishing – 1128p

Лабораторная работа 6. Использование UNION, INTERSECT и MINUS.

Цель лабораторной работы:

Научиться, используя операторы DML (data manipulation language) языка запросов SQL, создавать сложные запросы, использующие подзапросы, объединения, разности, пересечения.

Краткие теоретические сведения:

Подзапросы.

С помощью SQL вы можете вкладывать запросы внутрь друга друга. Обычно, внутренний запрос генерирует значение, которое проверяется в предикате внешнего запроса, определяющего верно оно или нет. Например, предположим, что мы знаем имя продавца: Motika, но не знаем значение его поля snum, и хотим извлечь все заказы из таблицы заказов. Имеется один способ, чтобы сделать это:

```
SELECT * FROM Orders WHERE snum = ( SELECT snum FROM Salespeople WHERE sname = 'Motika');
```

Необходимо, чтобы наш подзапрос в предыдущем примере возвращал одно и только одно значение. Имея выбранным поле snum «WHERE city = 'London'» вместо «WHERE sname = 'Motika'», можно получить несколько различных значений. Это может сделать уравнение в предикате основного запроса невозможным для оценки верности или неверности, и команда выдаст ошибку. При использовании подзапросов в предикатах, основанных на реляционных операторах, вы должны убедиться, что использовали подзапрос который будет выдавать одну и только одну строку вывода. Если вы используете подзапрос, который не выводит никаких значений вообще, команда не потерпит неудачи; но основной запрос не выведет никаких значений. Подзапросы, которые не производят никакого вывода (или нулевой вывод), вынуждают рассматривать предикат ни как верный, ни как неверный, а как неизвестный. Однако, неизвестный предикат имеет тот же самый эффект что и неверный: никакие строки не выбираются основным запросом.

Один тип функций, который автоматически может производить одиночное значение для любого числа строк, конечно же, – агрегатная функция.

Любой запрос, использующий одиночную функцию агрегата без предложения GROUP BY, будет выбирать одиночное значение для использования в основном предикате. Например, вы хотите увидеть все заказы имеющие сумму приобретений выше средней на 4-е Октября:

```
SELECT * FROM Orders WHERE amt > ( SELECT AVG (amt) FROM Orders WHERE odate = 10/04/1990 );
```

Вы можете использовать подзапросы, которые производят любое число строк если вы используете специальный оператор IN. Как вы помните, IN определяет набор значений, одно из которых должно совпадать с другим термином уравнения предиката в порядке, чтобы предикат был верным. Когда вы используете IN с подзапросом, SQL просто формирует этот набор из вывода подзапроса. Мы можем, следовательно, использовать IN чтобы выполнить такой же подзапрос, который не будет работать с реляционным оператором, и найти все атрибуты таблицы заказов для продавца в Лондоне:

```
SELECT * FROM Orders WHERE snum IN ( SELECT snum FROM Salespeople WHERE city = 'LONDON' );
```

Операторы BETWEEN, LIKE, и IS NULL не могут использоваться с подзапросами, но можно использовать операторы ALL и ANY

Например выбрать заказчиков, которые имеют больший рейтинг чем любой заказчик в Риме:

```
SELECT * FROM Customers WHERE rating > ANY ( SELECT rating FROM Customers WHERE city = 'Rome' );
```

С помощью ALL, предикат является верным, если каждое значение выбранное подзапросом удовлетворяет условию в предикате внешнего запроса. Если мы хотим пересмотреть наш предыдущий пример чтобы вывести только тех заказчиков чьи оценки, фактически, выше чем у каждого заказчика в Лондоне:

```
SELECT * FROM Customers WHERE rating > ALL
(SELECT rating FROM Customers WHERE city =
'Rome' );
```

Смысл всех подзапросов тот, что все они выбирают одиночный столбец. Это обязательно, поскольку выбранный вывод сравнивается с одиночным значением. Подтверждением этому то, что SELECT * не может использоваться в подзапросе. Имеется исключение из этого, когда подзапросы используются с оператором EXISTS.

Когда вы используете подзапросы в SQL, вы можете обратиться к внутреннему запросу таблицы в предложении внешнего запроса FROM, сформировав – соотнесенный подзапрос. Когда вы делаете это, подзапрос выполняется неоднократно, по одному разу для каждой строки таблицы основного запроса. Соотнесенный подзапрос - один из большого количества тонких понятий в SQL из-за сложности в его оценке. Если вы сумеете овладеть им, вы найдете что он очень мощный, потому что может выполнять сложные функции с помощью очень лаконичных указаний.

Например, имеется один способ найти всех заказчиков в заказах на 3-е Октября:

```
SELECT * FROM Customers outer WHERE
10/03/1990 IN ( SELECT odate
FROM Orders inner WHERE outer.cnum =
inner.cnum );
```

EXISTS – это оператор, который производит верное или неверное значение, другими словами, выражение Буля. Это означает, что он может работать автономно в предикате или в комбинации с другими выражениями Буля использующими Булевы операторы AND, OR, и NOT. Он берет подзапрос как аргумент и оценивает его как верный, если тот производит любой вывод или как неверный, если тот не делает этого. Этим он отличается от других операторов предиката, в которых он не может быть неизвестным. Например, мы можем решить, извлекать ли нам некоторые данные из таблицы заказчиков если, и только если, один или более заказчиков в этой таблице находятся в San Jose:

```
SELECT cnum, cname, city FROM Customers WHERE EXISTS ( SELECT * FROM Customers WHERE city = ‘San Jose’ );
```

Или мы можем вывести продавцов, которые имеют многочисленых заказчиков:

```
SELECT DISTINCT snum FROM Customers outer WHERE EXISTS ( SELECT * FROM Customers inner WHERE inner.snum = outer.snum AND inner.cnum < > outer.cnum );
```

Однако для нас может быть полезнее вывести больше информации об этих продавцах, а не только их номера. Мы можем сделать это, объединив таблицу Заказчиков с таблицей Продавцов:

```
SELECT DISTINCT first.snum, sname, first.city FROM Salespeople first, Customers second WHERE EXISTS ( SELECT * FROM Customers third WHERE second.snum = third.snum AND second.cnum < > third.cnum ) AND first.snum = second.snum;
```

Объединение.

Вы можете поместить многочисленые запросы вместе и объединить их вывод используя предложение UNION. Предложение UNION объединяет вывод двух или более SQL запросов в единый набор строк и столбцов. Например чтобы получить всех продавцов и заказчиков размещенных в Лондоне и вывести их как единое целое вы могли бы ввести:

```
SELECT snum, sname FROM Salespeople WHERE city = 'London'
UNION
SELECT cnum, cname FROM Customers WHERE city = 'London';
```

Как вы можете видеть, столбцы, выбранные двумя командами, выведены так, как если она была одна. Заголовки столбца исключены, потому что ни один из столбцов выведенных объединением, не был извлечен непосредственно из только одной таблицы.

Операторы пересечения (INTERSECT) и разности (MINUS) выполняются таким же образом, как и объединение.

Задание:

Для созданной в предыдущих лабораторных работах базы данных разработать систему запросов (минимум по 3 запроса):

1. Содержащих подзапросы.
2. Производящих объединение, пересечение или разность двух и более запросов.

Каждый запрос должен снабжаться описанием, чтобы проверить правильность построения запроса.

Требования к отчету:

Отчет должен быть выполнен в соответствии с правилами оформления отчетов указанном в приложении 1.

Отчет по выполненной работе должен содержать:

1. Тему лабораторной работы.
2. Цель лабораторной работы.
3. Задание.
4. Схему реляционной базы данных из лабораторной работы №2.
5. Список запросов к базе данных. Каждый запрос должен содержать описание запроса (что этот запрос выполняет) и запрос на языке SQL. Все запросы разделить по категориям, которые указаны в задании.

Контрольные вопросы:

1. Для чего нужны объединения?
2. Как формируются подзапросы?
3. Колько раз выполняется подзапрос в запросе?
4. В чем разница в механизмах работы подзапросов и соединений?
5. Что такое соотнеченные подзапросы?
6. В каких командах языка SQL возможно использование подзапросов?

Литература:

1. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p
2. Мартин Грабер. – Введение в SQL. изд. Лори, 382 стр. 1992 г.

3. Jarrod Hollingworth, Bob Swart, Mark Cashman, Paul Gustavson – Borland C++ Builder 6 Developer's Guide 2nd Edition – 2002 – SAMS Publishing – 1128p

Лабораторная работа 7. Создание простых приложений на языке C++, взаимодействующих с базой данных.

Цель лабораторной работы:

Научиться создавать программы, позволяющие вести работу с базами данных пользователям – чтение поиск необходимой информации из базы данных.

Краткие теоретические сведения:

Приложение баз данных, как следует уже из его названия, предназначено для взаимодействия с некоторым источником данных – базой данных (БД). Взаимодействие подразумевает получение данных, их представление в определенном формате для просмотра пользователем, редактирование в соответствии с реализованными в программе бизнес-алгоритмами и возврат обработанных данных обратно в базу данных.

В качестве источника данных могут выступать как собственно базы данных, так и обычные файлы – текстовые, электронные таблицы и т. д. Но здесь мы будем рассматривать приложения, работающие с базами данных. Как известно, базы данных обслуживаются специальными программами – системами управления базами данных (СУБД), которые делятся на локальные, преимущественно однопользовательские, предназначенные для настольных приложений, и серверные – сетевые (часто удаленные), многопользовательские, функционирующие на выделенных компьютерах – серверах. Главный критерий такой классификации – объем базы данных и средняя нагрузка на СУБД.

Тем не менее, несмотря на разнообразие реализаций, общая архитектура приложения баз данных остается неизменной.

Само приложение включает механизм получения и отправки данных, механизм внутреннего представления данных в том или ином виде, пользовательский интерфейс для отображения и редактирования данных, бизнес-логику для обработки данных.

Механизм получения и отправки данных обеспечивает соединение с источником данных (часто опосредованно). Он должен «знать», куда ему обращаться и какой протокол обмена использовать для обеспечения двунаправленного потока данных.

Механизм внутреннего представления данных является ядром приложения баз данных. Он обеспечивает хранение полученных данных в приложении и предоставляет их по запросу других частей приложения.

Пользовательский интерфейс обеспечивает просмотр и редактирование данных, а также управление данными и приложением в целом.

Бизнес-логика приложения представляет собой набор реализованных в программе алгоритмов обработки данных.

Между приложением и собственно базой данных находится специальное программное обеспечение (ПО), связывающее программу и источник данных и управляющее процессом обмена данными. Это ПО может быть реализовано самыми разнообразными способами, в зависимости от объема базы данных, решаемых системой задач, числа пользователей, способами соединения приложения и базы данных. Промежуточное ПО может быть реализовано как окружение приложения, без которого оно вообще не будет работать, как набор драйверов и динамических библиотек, к которым обращается приложение, может быть интегрировано в само приложение. Наконец, это может быть отдельный удаленный сервер, обслуживающий тысячи приложений.

Технологии доступа к данным.

1. ***Data Access Objects (DAO)*** — технология доступа к данным компании Microsoft.

DAO ведёт своё начало от компонента Visual Basic 2.0 под названием «VT Objects», предоставлявшего сильно ограниченный доступ к данным ODBC-источников.

DAO 1.0 появилась в ноябре 1992 года как API для работы с СУБД Jet. Технология Jet поддерживала доступ к файлам формата MDB (Microsoft Access), ODBC-источникам данных и к источникам данных ISAM.

В программном обеспечении Data Access Object (DAO) – это объект, который предоставляет абстрактный интерфейс к какому-либо типу базы данных или механизму хранения. Определённые

возможности предоставляются независимо от того, какой механизм хранения используется и без необходимости специальным образом соответствовать этому механизму хранения.

2. *Open Database Connectivity (ODBC)* — это программный интерфейс (API) доступа к базам данных, разработанный фирмой Microsoft, в сотрудничестве с Simba Technologies/

В начале 1990 г. существовало несколько поставщиков баз данных, каждый из которых имел собственный интерфейс. Если приложению было необходимо общаться с несколькими источниками данных, для взаимодействия с каждой из баз данных было необходимо написать свой код. Для решения возникшей проблемы Microsoft и ряд других компаний создали стандартный интерфейс, получивший название CLI (Common Language Interface), для получения и отправки источникам данных различных типов. Этот интерфейс был назван Open Database Connectivity, или открытый механизм взаимодействия с базами данных. Затем каждая фирма разрабатывала драйвер перевода своего диалекта SQL в CLI и наоборот.

Технология ODBC предусматривает создание дополнительного уровня между приложением и используемой СУБД. В архитектуре ODBC используется один ODBC Driver Manager и несколько ODBC-драйверов, отвечающих за реализацию особенностей доступа к каждой отдельной СУБД.

Преимущества:

- простота разработки приложения;
- технология ODBC позволяет создавать распределенные гетерогенные приложения без учета конкретных СУБД, т.е. приложение становится независимым от СУБД.

Недостатки:

- снижение скорости доступа к данным, что связано с необходимостью трансляции запросов;
- увеличение время обработки запросов, что связано с введением дополнительного программного слоя;
- необходимы предварительная инсталляция и настройка ODBC-драйвера (указание драйвера СУБД, сетевого пути к серверу, базы данных и т.д.) на каждом рабочем месте.

Параметры этой настройки являются статическими, т.е. приложение изменить их самостоятельно не может;

- предоставляет доступ только к реляционным SQL-ориентированным БД.

Данные в БД могут быть представлены в любом виде и формате (электронные таблицы, документы в rtf- формате, почтовые системы и т.д.).

С помощью ODBC прикладные программисты могли разрабатывать приложения для использования одного интерфейса доступа к данным, не беспокоясь о тонкостях взаимодействия с несколькими источниками.

3. *Remote Data Objects (RDO)* – технология доступа к базам данных компании Microsoft. Представляет собой набор COM-объектов инкапсулирующих ODBC API, а также клиентскую курсорную библиотеку.

Технология RDO появилась в 1995 году одновременно с выходом продукта Visual Basic 4.0.

RDO позиционировалась как технология более простая чем прямое использование вызовов ODBC и в то же время более эффективная чем технология DAO. RDO была ориентирована на обработку данных на стороне сервера БД (такого как MS SQL Server, Oracle итд) в отличие от DAO ориентированной в основном на обработку данных на стороне клиента.

Уже с 1996 года Microsoft стала продвигать новую технологию – ADO, никак не связанную с ODBC, как перспективный способ доступа к данным, что сильно подорвало позиции RDO. В настоящее время технология практически не используется.

4. *ActiveX Data Objects (ADO)* – «объекты данных ActiveX» – интерфейс программирования приложений для доступа к данным, разработанный компанией Microsoft (MS Access, MS SQL Server) и основанный на технологии компонентов ActiveX. ADO позволяет представлять данные из разнообразных источников (реляционных баз данных, текстовых файлов и т. д.) в объектно-ориентированном виде.

Объектная модель ADO состоит из следующих объектов высокого уровня и семейств объектов:

- Connection (представляет подключение к удалённому источнику данных);
- Recordset (представляет набор строк, полученный от источника данных);
- Command (используется для выполнения команд и SQL-запросов с параметрами);
- Record (может представлять одну запись объекта Recordset или же иерархическую структуру, состоящую из текстовых данных);
- Stream (используется для чтения и записи потоковых данных, например, документов XML или двоичных объектов);
- Errors (представляет ошибки);
- Fields (представляет столбцы таблицы базы данных);
- Parameters (представляет набор параметров SQL-инструкции);
- Properties (представляет набор свойств объекта);

Компоненты ADO используются в языках высокого уровня, таких как VBScript в ASP, JScript в WSH, Visual Basic, Delphi.

Последней версией ADO является версия 2.8. В рамках платформы Microsoft .NET интерфейс ADO заменён ADO.NET.

5. ***ADOdb*** – программная библиотека, обеспечивающая прикладной интерфейс доступа к базам данных для языков программирования PHP и Python, основанная на некоторых концепциях Microsoft ActiveX Data Objects. Библиотека обеспечивает разработчика приложений абстрактным инструментарием, позволяющим создавать приложения без необходимости программирования поддержки каждого из конкретных возможных типов источников данных. В частности, у разработчиков появляется возможность изменить СУБД без необходимости вносить исправления в программный код.

6. ***Object Linking and Embedding, Database (OLE DB)*** – набор интерфейсов, основанных на COM, которые позволяют приложениям обращаться к данным, хранимым в разных источниках информации или хранилищах данных с помощью унифицированного доступа.

OLE DB (связывание и внедрение объектов, базы данных, а иногда в литературе встречается как OLEDB или OLE-DB)

является API разработанной Microsoft для доступа к различным типам данных, которые хранятся в единой форме. Программа представляет собой набор интерфейсов реализованных с помощью Component Object Model (COM); в данном случае это связано с OLE. Она была разработана в качестве дальнейшего развития и должна прийти на замену и в качестве преемника ODBC, расширяя набор функций для поддержки более широкого круга нереляционных источников данных, таких как объектно-ориентированые базы данных или электронные таблицы, и для которых не обязательно использовать SQL.

OLE DB отделяет хранилище данных из приложения, которое должно иметь доступ к нему через набор абстракций, которые включают DataSource, сессию, командную строку. Это было сделано потому, что различным приложениям необходим доступ к различным видам и источникам данных и не всегда нужно знать, как получить доступ к методологии функционирования конкретной технологии. OLE DB концептуально разделена на потребителей и поставщиков. Потребителями являются приложения, которым необходим доступ к данным, а поставщик реализует в своем интерфейсе программный компонент и, следовательно, обеспечивает информацией потребителя. OLE DB провайдеры могут быть созданы для обеспечения простого доступа к таким хранилищам данных как текстовые файлы и электронные таблицы, вплоть до таких сложных баз данных, как Oracle, SQL Server и Sybase ASE. Он может также обеспечить доступ к иерархическими хранилищами данных таких, как системы электронной почты.

7. *Java DataBase Connectivity (JDBC)* – соединение с базами данных на Java – платформенно-независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД, реализованный в виде пакета java.sql, входящего в состав Java SE.

JDBC основан на концепции так называемых драйверов, позволяющих получать соединение с базой данных по специально описанному URL. Драйверы могут загружаться динамически (во время работы программы). Загрузившись, драйвер сам регистрирует себя и вызывается автоматически, когда программа требует URL, содержащий протокол, за который драйвер отвечает.

Интерфейсы JDBC API содержит два основных типа интерфейсов: первый – для разработчиков приложений и второй (более низкого уровня) – для разработчиков драйверов.

Соединение с базой данных описывается классом, реализующим интерфейс java.sql.Connection.

Имея соединение с базой данных, можно создавать объекты типа Statement, служащие для исполнения запросов к базе данных на языке SQL.

Существуют следующие виды типов Statement, различающихся по назначению:

- java.sql.Statement – Statement общего назначения;
- java.sql.PreparedStatement – Statement, служащий для выполнения запросов, содержащих подставляемые параметры (обозначаются символом '?' в теле запроса);
- java.sql.CallableStatement – Statement, предназначенный для вызова хранимых процедур.

Интерфейс java.sql.ResultSet позволяет легко обрабатывать результаты запроса.

Преимуществами JDBC считают:

- Лёгкость разработки: разработчик может не знать специфики базы данных, с которой работает;
- Код не меняется, если компания переходит на другую базу данных;
- Не нужно устанавливать громоздкую клиентскую программу;
- К любой базе можно подсоединиться через легко описываемый URL.

8. ***PHP Data Objects (PDO)*** – расширение для PHP, предоставляющее разработчику простой и универсальный интерфейс для доступа к различным базам данных.

PDO предлагает единые методы для работы с различными базами данных, хотя текст запросов может немного отличаться. Так как многие СУБД реализуют свой диалект SQL, который в той или иной мере поддерживает стандарты ANSI и ISO, то при использовании простых запросов можно добиться совместимости между различными языками. На практике это означает, что можно

достаточно легко перейти на другую СУБД, при этом не меняя или частично изменяя код программы.

PDO не использует абстрактных слоёв для подключения к БД, наподобие ODBC, а использует для разных БД их «родные» драйверы, что позволяет добиться высокой производительности. В настоящее время для PDO существуют драйверы практически ко всем общеизвестным СУБД и интерфейсам. Впрочем, есть и драйвер для подключения к ODBC.

9. *Borland Database Engine (BDE)* – «движок баз данных Borland» – 32-битный движок баз данных под Microsoft Windows для доступа к базам данных из Borland Delphi, C++ Builder, IntraBuilder, Paradox for Windows и Visual dBASE for Windows.

История Turbo Pascal фирмы Borland включал в себя «базу данных» Toolbox, которая была первым дополнением для компиляторов Borland, предназначенным для работы с БД. Затем появился движок БД Paradox for Windows – PXENGWIN, который мог быть включён в программу для подключения к таблицам формата Paradox.

Первым механизмом подключения, основанным на использовании DLL, был ODAPI (от англ. Open Database API — «открытый интерфейс прикладного программирования баз данных»). Он представлял собой попытку Borland унифицировать взаимосвязи в своём программном пакете, включавшим в себя совершенно новый Paradox for Windows 4 и Quattro. С версиями 4.5 / 5.0 Paradox for Windows этот движок баз данных оформился как IDAPI (от англ. Integrated Database Application Program Interface – «интегрированный интерфейс прикладного программирования баз данных»).

В 2000 году Borland представила новую, основанную на SQL-драйверах, архитектуру, названную dbExpress, которая сделала устаревшей использовавшуюся в BDE технологию SQL Links.

В BDE имеющийся набор драйверов баз данных даёт единообразный доступ к стандартным источникам данных: Paradox, dBASE, FoxPro, Access, а также текстовым БД. Вы можете добавлять драйверы Microsoft ODBC при необходимости подключения к ODBC-сокету. Кроме того, Borland предоставляет SQL Links для доступа к широкому диапазону мощных СУБД, включая Informix, DB2, InterBase, Oracle и Sybase.

BDE имеет объектно-ориентированное устройство. Во время выполнения приложение взаимодействует с BDE, создавая различные BDE-объекты. Эти объекты затем используются для управления элементами БД, такими как таблицы и запросы. BDE API даёт прямой и оптимизированный доступ к движку, а также к встроенным в BDE драйверам для dBASE, Paradox, FoxPro, Access и текстовых БД.

Файлы ядра движка БД существуют как набор DLL, код которых полностью реентерабелен и потокобезопасен. В поставку BDE входит набор дополнительных утилит и примеров приложений.

Система BDE конфигурируется с помощью BDE Administrator (BDEADMIN.EXE)

В BDE используется «Local SQL», подмножество стандарта ANSI-92 языка SQL, расширенное для поддержки используемых в Paradox и DBF (называемых в BDE «стандартными» таблицами) соглашений о наименовании таблиц и полей. Local SQL позволяет вам использовать SQL для запросов к локальным «стандартным» таблицам, которые не находятся на серверах БД, в т. ч. удалённых. Local SQL также является необходимым средством для создания запросов с выборками из многих таблиц, часть которых локальна, а часть находится на удалённых SQL-серверах.

Разработка приложений с базами данных с помощью Borland C++ Builder.

Для работы с локальными и удаленными базами данных приложения C++Builder используют механизм доступа к СУБД, называемый Borland Database Engine (BDE). В случае локальных баз данных, таких как таблицы dBase или Paradox, BDE использует свои собственные встроенные драйверы СУБД. Для доступа к удаленным базам данных, например, Oracle или Sybase, BDE связывается с их серверами, используя драйверы SQL Links и/или драйверы ODBC. Эти драйверы часто обращаются к библиотекам драйверов самой СУБД. Схематически взаимодействие этих компонентов показано на рис. 7.1.

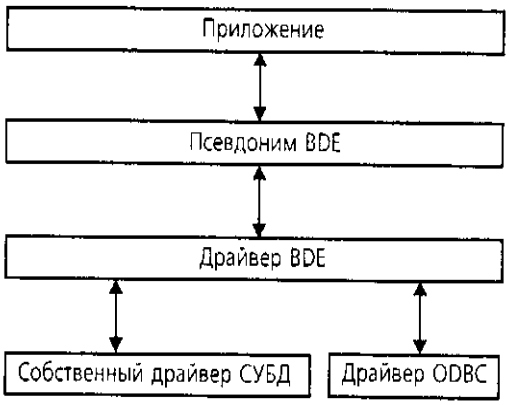


Рис. 7.1. Простая, но гибка архитектура доступа к базам данных C++Builder.

Внутри вашего приложения интерфейсные компоненты обращаются к компонентам TDataSource. Обычно в форме содержится небольшое количество компонентов TDataSource, хотя интерфейсных компонентов может быть достаточно много. Компоненты TDataSource, в свою очередь, обращаются к одному или нескольким компонентам TDataSet. В некоторых случаях форма может включать только один компонент TDataSet и один TDataSource. Рисунок 7.2 иллюстрирует взаимосвязь названных компонентов.

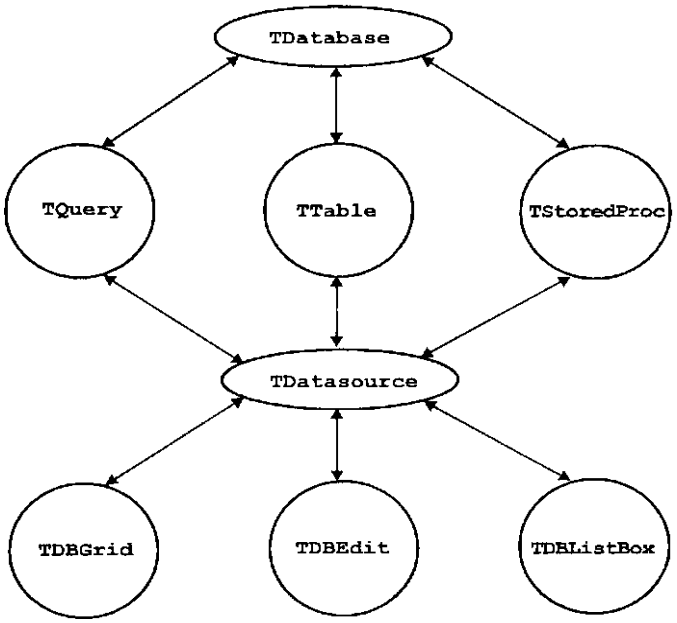


Рис. 7.2. Доступ к базе данных с точки зрения приложения C++Builder.

Гибкость, присущая этой многоуровневой архитектуре, позволяет достаточно легко разрабатывать приложения для работы с базами данных, отличающиеся не только надежностью, но и возможностью расширения и изменения. Благодаря разделению внутренних драйверов BDE и компонентов приложения, возможно (по крайней мере, теоретически) изменять внутреннюю структуру базы данных даже без перекомпиляции приложения. Модульность архитектуры позволяет заменять отдельные участки, не разрабатывая заново все приложение.

Задание:

Для созданной базы данных разработать приложение с пользовательским интерфейсом, которое обеспечивает полноценную работу с приложением. Приложение должно выполнять следующие функции:

1. Зарегистрировать драйвер ADO к MySQL в системе Windows.
2. Обеспечивать подключение к БД через компоненты ADO.
3. Обеспечивать Обеспечить вывод данных через компонент DBGrid и DBNavigator.
4. Все основные таблицы базы данных, схема которой разработана в предыдущих лабораторных работах, отобразаить на формах через DBGrid и DBNavigator.

Требования к отчету:

Отчет должен быть выполнен в соответствии с правилами оформления отчетов указанном в приложении 1.

Отчет по выполненной работе должен содержать:

1. Тему лабораторной работы.
2. Цель лабораторной работы.
3. Задание.
4. Схему реляционной базы данных из лабораторной работы №2.
5. Скриншоты процесса добавления подключения к БД через драйвер ADO.
6. Скриншоты программы отображающие данные таблиц.

Контрольные вопросы:

1. Что такое ADO?
2. Как работает ADO?
3. Объясните механизм работы приложений разработанных в среде C++ Builder с базой данных через ADO?
4. Для чего нужны компоненты TADOTable, TADOConnection, TDataSource?
5. Как осуществить работу по изменению данных с помощью ADOTable и DBNavigator?
6. Как инициировать соединение с базой данных?

Литература:

1. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p
2. Мартин Грабер. – Введение в SQL. изд. Лори, 382 стр. 1992 г.
3. Jarrod Hollingworth, Bob Swart, Mark Cashman, Paul Gustavson – Borland C++ Builder 6 Developer's Guide 2nd Edition – 2002 – SAMS Publishing – 1128p

Лабораторная работа 8. Добавление, модификация и удаление строк из приложений C++, используя ADO и ODBC.

Цель лабораторной работы:

Научиться создавать программы, позволяющие вести работу с базами данных пользователям: добавление, удаление и редактирование.

Краткие теоретические сведения:

Компоненты для работы с базами данных.

Рассмотрим теперь компоненты, используемые для создания приложений БД. Кроме компонентов, C++ Builder также предоставляет разработчику специальные объекты, например, объекты типа Field. Как и другие управляющие элементы C++ Builder, связанные с БД компоненты делятся на визуальные и невизуальные.

Невизуальные компоненты предназначены для организации доступа к данным, содержащимся в таблицах. Они представляют

собой промежуточное звено между данными таблиц БД и визуальными компонентами.

Визуальные компоненты используются для создания интерфейсной части приложения. С их помощью пользователь может выполнять такие операции с таблицами БД, как просмотр или редактирование данных. Визуальные компоненты также называют элементами, чувствительными к данным.

Компоненты, используемые для работы с БД, находятся на страницах Data Access, Data Controls, dbExpress, BDE, ADO и InterBase.

Состав компонентов может настраиваться в диалоговом окне Palette Properties (Свойства палитры), вызываемого командой Properties (Свойства) контекстного меню Палитры компонентов. Мы приводим состав Палитры компонентов, который получается после установки C++ Builder 6.

На изрядно поредевшей странице Data Access (рис. 8.1) находятся невизуальные компоненты, предназначенные для организации доступа к данным:

- DataSource – источник данных;
- clientDataSet – клиентский набор данных;
- DataSetProvider – провайдер набора данных.

Рис. 8.1. Страница DataAccess.

На странице Data Controls (рис. 8.2) расположены визуальные компоненты, предназначенные для управления данными:

- DBGrid – сетка (таблица);
- DBNavigator – навигационный интерфейс; □ DBText – надпись;
- DBEdit – однострочный редактор (поле редактирования);
- DBMemo – многострочный редактор (панель редактирования);
- DBimage – графический образ (изображение);
- DBListBox – простой список;
- DBComboBox – комбинированный список;

- DBCheckBox – независимый переключатель;
- DBRadioGroup – группа зависимых переключателей;
- DBLookupListBox – простой список, формируемый по полю другого набора данных;
- DBLookupComboBox – комбинированный список, формируемый по полю другого набора данных;
- DBRichEdit – полнофункциональный тестовый редактор (поле редактирования);
- DBCtriGrid – модифицированная сетка;
- DBChart – диаграмма.

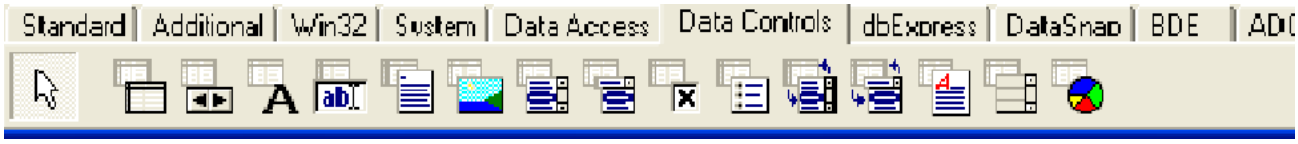


Рис. 8.2. Страница DataControls.

Страница BDE (рис. 8.3) содержит компоненты, предназначенные для управления данными с использованием BDE:

- Table – набор данных, основанный на таблице БД;
- Query – набор данных, основанный на SQL–запросе;
- storedProc – вызов хранимой процедуры сервера;
- DataBase – соединение с БД;
- session – текущий сеанс работы с БД;
- BatchMove – выполнение операций над группой записей;
- updateSQL – модификация набора данных, основанного на SQL–запросе;
- NestedTable – вложенная таблица;
- BDEClientDataSet – клиентский набор данных.

Рис. 8.3. Страница BDE.

На странице ADO (рис. 8.4) расположены компоненты, предназначенные для управления данными с использованием технологии ADO (Active Data Objects):

- ADOConnection – соединение;

- ADOcommand – команда;
- ADODataSet – набор данных;
- ADOTable – набор данных Table;
- ADOQuery – набор данных Query;
- ADOStoredProc – вызов хранимой процедуры сервера;
- RDSconnection – соединение RDS.

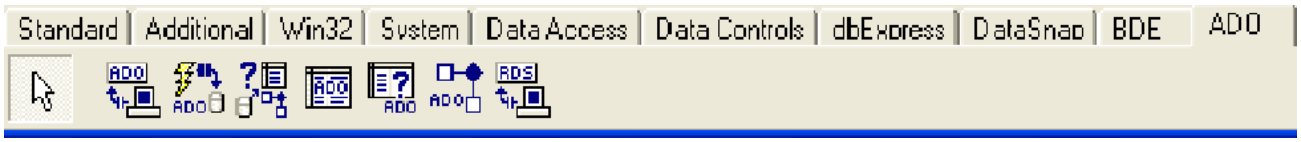


Рис. 8.4. Страница ADO.

Соединение RDS служит для управления передачей объекта Recordset от одного процесса (компьютера) к другому при создании серверных приложений.

На странице InterBase (рис. 8.5) находятся компоненты, предназначенные для работы с сервером InterBase:

- IBTable – набор данных Table;
- IBQuery – набор данных Query;
- IBStoredProc – вызов хранимой процедуры;
- IBDatabase – соединение с БД;
- IBTransaction – транзакция;
- IBUpdateSQL – модификация набора данных, основанного на SQL–запросе;
- IBDataSet – источник данных;
- IBSQL – выполнение SQL–запроса;
- IBDatabaseInfo – информация о БД;
- IBSQLMonitor – монитор выполнения SQL–запросов;
- IBEvents – событие сервера;
- IBExtract – извлечение данных;
- IBClientDataSet – клиентский источник данных.

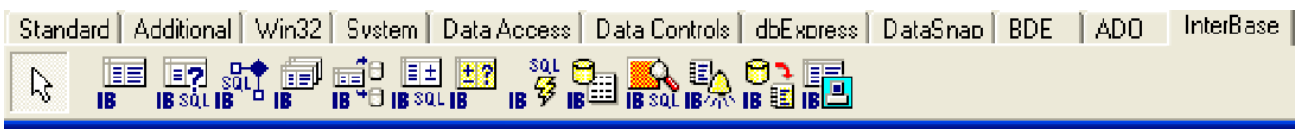


Рис. 8.5. Страница InterBase.

Названия многих компонентов, предназначенных для работы с данными, содержат префиксы, например, DB, IB ИЛИ QR. Префикс DB означает, что визуальный компонент связан с

данными и используется для построения интерфейсной части приложения. Такие компоненты размещаются на форме и предназначены для управления данными со стороны пользователя. Префикс IB означает, что компонент предназначен для работы с сервером InterBase.

Имя таблицы лучше выбирать из раскрывающегося списка в поле значения свойства TableName. Если путь к БД (свойство DataBaseName) задан правильно, то в этом списке отображаются главные файлы всех доступных таблиц.

Значение True свойству Active нужно устанавливать после задания таблицы БД, т. е. после установки нужных значений свойств DataBaseName и TableName.

Компонент DataSource является промежуточным звеном между компонентом Table, соединенным с реальной таблицей БД, и визуальными компонентами DBGrid и DBNavigator, с помощью которых пользователь взаимодействует с этой таблицей. На компонент Table, с которым связан компонент DataSource, указывает свойство DataSet последнего.

Компонент DBGrid отображает содержимое таблицы БД в виде сетки, в которой столбцы соответствуют полям, а строки – записям таблицы. По умолчанию пользователь может просматривать и редактировать данные. Компонент DBNavigator позволяет пользователю осуществлять перемещение по таблице, редактировать, вставлять и удалять записи. Компоненты DBGrid и DBNavigator связываются со своим источником данных – компонентом DataSource – через свойства DataSource.

При разработке приложения значения всех свойств компонентов можно задать с помощью Инспектора объектов. При этом требуемые значения либо непосредственно вводятся в поле, либо выбираются из раскрывающихся списков. В последнем случае приложение создается с помощью мыши и не требует набора каких–либо символов с клавиатуры.

Для автоматизации процесса создания формы, использующей компоненты для операций с БД, можно вызвать Database Form Wizard (Мастер форм баз данных), показанный на рис. 8.6. Этот Мастер расположен на странице Business Хранилища объектов.

Мастер позволяет создавать формы для работы с отдельной таблицей и со связанными таблицами, при этом можно использовать наборы данных Table или Query.

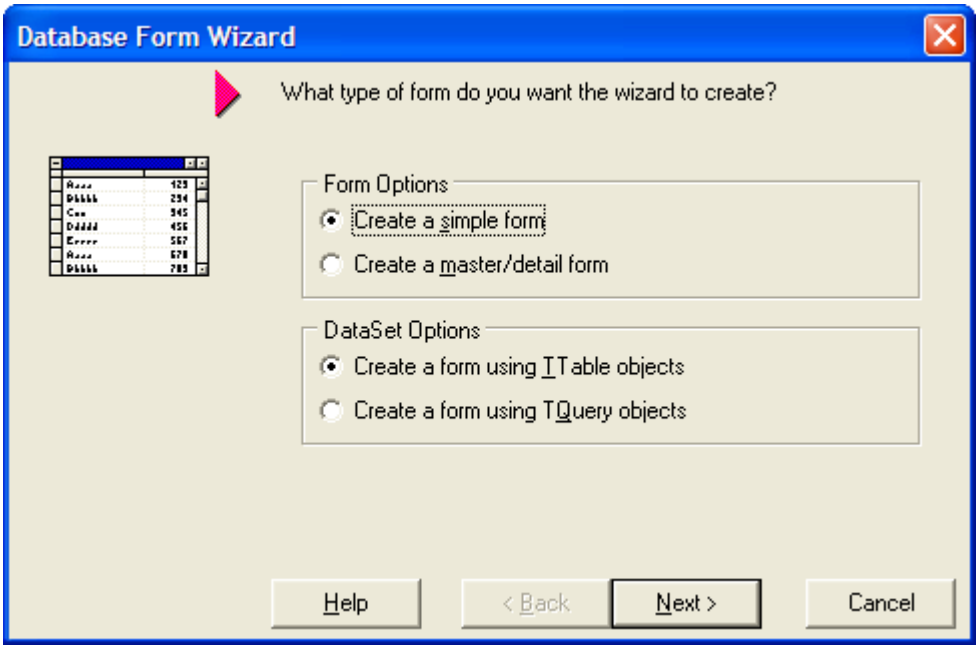


Рис. 8.6. Окно Database Form Wizard.

Использование модуля данных.

При конструировании формы невизуальные компоненты, используемые для доступа к данным, такие как DataSource или Table, размещаются на форме, но при выполнении приложения эти компоненты не видны. Поэтому их можно размещать в любом удобном месте формы, выступающей для них контейнером – модулем. Кроме того, для размещения невизуальных компонентов, через которые осуществляется доступ к данным, предназначен специальный объект – модуль данных.

Существует три типа модуля данных:

1. простой модуль данных;
2. удаленный модуль данных;
3. Web–модуль.

Ниже рассматривается простой модуль данных, который представлен объектом DataModule. Использование удаленного модуля данных и Web–модуля рассматривается в разделах трехуровневые приложения и публикация БД в Интернете в книге Б. С. Послед «Borland C++ Builder 6. Разработка приложений баз данных».

Если применяется простой модуль данных, то взаимосвязь компонентов приложения и таблицы БД имеет вид, показанный на рис. 8.7.

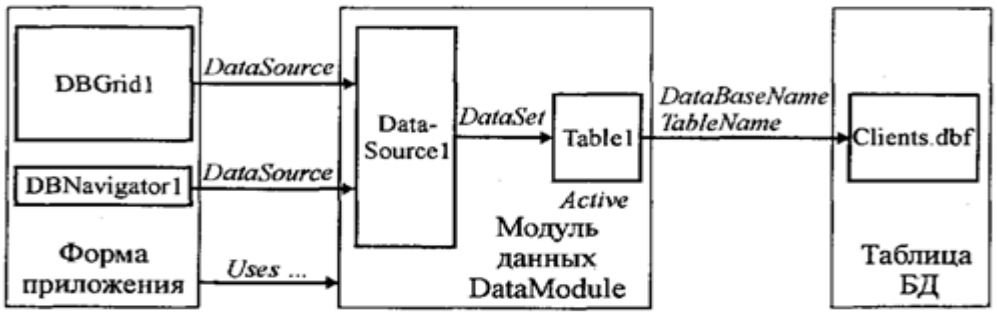


Рис. 8.7. Модуль данных.

Модуль данных, как и форма, является контейнером для своих невизуальных компонентов, и для него создается модуль кода с расширением C++. Добавление модуля данных к проекту выполняется командой File/New/DataModule главного меню C++ Builder. В окне модуля компоненты размещаются таким же образом, как и на форме. При выборе объекта в Инспекторе объектов отображаются его свойства, значения которых можно просматривать и изменять.

При обращении к содержащимся в модуле данных компонентам для них указывается составное имя, в которое, кроме имени компонента, входит также имя модуля данных. Составное имя имеет формат

<Имя модуля данных> -> <Имя компонента>

Ниже приводится пример кода, в котором осуществляется обращение к компонентам модуля данных.

```
void __fastcall TForm1::FormCreate(TObject
*Sender)
{
DataModule2->Table1->DatabaseName = "dbdemos";
DataModule2->Table1->TableName = "clients.dbf";
DataModule2->DataSource1->DataSet =
DataModule2->Table1;
DBGrid1->DataSource = DataModule2->DataSource1;
DBNavigator1->DataSource = DataModule2-
>DataSource1;
DataModule2->Table1->Active = true;
}
```

Для компонентов выполняется установка значений свойств, связывающих между собой эти компоненты и таблицу БД. Значения свойств устанавливаются динамически в процессе

выполнения приложения, для чего использован обработчик события создания главной формы приложения. В составных именах компонентов доступа к данным, которыми являются источник данных DataSourcel и набор данных Tablel, указывается имя модуля данных DataModule2.

Чтобы обеспечить возможность доступа к компонентам модуля данных в модуле формы, в список uses раздела implementation необходимо включить ссылку на модуль данных:

```
#include "unit2.cpp";
```

Ссылку на другой модуль можно написать самостоятельно, но C++ Builder дает возможность вставить ее автоматически. При выборе команды File/Use Unit.

Помимо компонентов доступа к данным, которыми являются session, Database, Table, Query, storedProc, BatchMove и др., в модуле данных можно размещать также невизуальные компоненты, не имеющие прямого отношения к БД, например, ImageList, OpenDialog ИЛИ Timer.

При работе с модулем данных в Палитре компонентов доступны только невизуалные компоненты.

Модуль данных позволяет:

- отделить управление БД от обработки данных;
- создать модуль, совместно используемый несколькими приложениями.

Основным назначением модуля данных является централизованное хранение компонентов доступа к данным, а также кода для этих компонентов, в частности, обработчиков событий. В модуле данных удобно размещать код, выполняющий управление БД, например, реализацию бизнес–правил.

Использование простого модуля данных несколькими приложениями позволяет ускорить разработку приложений, т. к. готовый модуль данных впоследствии можно включать в новые приложения. Кроме того, управление БД через общий модуль дает возможность определить для всех пользователей одинаковые режимы и правила работы с базой, а также делает более простым изменение этих режимов и правил.

Однако для небольших приложений использование простого модуля данных не всегда оправдано, т. к. может затруднить, а не облегчить разработку приложения.

Задание:

Разработать поиск и другие функций со сложными запросами (соединение, объединение, агрегатные функции, подзапросами):

1. Создать формы по редактированию и добавлению данных с использованием компонентов ADOTable и DBControls.
2. Используя ADOTabe произвести фильтрацию и сортировку данных.
3. Создать формы поиска с помощью ADOQuery.
4. Используя соединение и компонент ADOQuery реализовать соединение таблиц.
5. Реализовать поиск системы, используя компоненты ADOQuery.

Требования к отчету:

Отчет должен быть выполнен в соответствии с правилами оформления отчетов указанном в приложении 1.

Отчет по выполненной работе должен содержать:

1. Тему лабораторной работы.
2. Цель лабораторной работы.
3. Задание.
4. Схему реляционной базы данных из лабораторной работы №2.
5. Скриншоты работы программы ободражающие добавление, редактирование и удаление данных.
6. Скриншоты программы отображающие поиск и соединение таблиц.

Контрольные вопросы:

1. Как осуществить фильрацию данных через ADOTable?
2. Как осуществить сортировку через ADOTable?
3. Как использовать собственный SQL для манипуляции данными в приложении на C++?
4. В чем разница между методами Open() и ExecSQL() в ADOQuery?
5. Как сделать подзапросы и соединения в приложении, разработанном в среде C++ Bulder?
6. Можно ли создать привязку элементов, отображающих данные в виде списка, с другими элементами, т.е. при выборе

строки значения всех столбцов показать в соответствующих элементах Tedit или TCombobox?

Литература:

1. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p
2. Мартин Грабер. – Введение в SQL. изд. Лори, 382 стр. 1992 г.
3. Jarrod Hollingworth, Bob Swart, Mark Cashman, Paul Gustavson – Borland C++ Builder 6 Developer's Guide 2nd Edition – 2002 – SAMS Publishing – 1128p

Лабораторная работа 9. Использование элементов управления данными в Windows Form приложениях.

Цель лабораторной работы:

Найчиться работать с компонентыми управления данными в Windows Form, а аткже создавать отчеты с использованием компонентов QuickReports

Краткие теоретические сведения:

При работе с Базой Данных почти всегда требуется организация вывода некоторых данных на печать. Как раз для этого и были придуманы отчеты. Отчетом является некоторый печатный документ, который содержит текущее содержание Базы Данных. Хотя в отчете можно отобразить и не только информацию из Базы Данных, но и совершенно другие данные.

В Delphi для создания отчетов используется генератор отчетов QuickReport. Компоненты, предназначенные для создания отчетов находятся на странице QReport Палитры Компонентов. Многие компоненты на этой странице являются визуальными и почти не отличаются от аналогичных компонентов на других страницах Палитры Компонентов.

В комплект поставки C++ Builder входят три шаблона отчетов, содержащиесяна странице Forms репозитария объектов:

Таблица 9.1.Типы отчетов QuickReports.

Шаблон	Описание
QuickReport Mailing	Шаблон для создания почтовых этикеток

Labels	
QuickReport List	Шаблон для создания простого табличного отчета
QuickReport Master/detail	Шаблон для создания отчета Master/Detail.

В сгенерированный автоматически текст модуля, связанного с формой, созданнойна основе этих шаблонов, включаются в виде комментариев инструкции по модификацииполученной формы.

Главным элементом отчета является компонент QuickRep, который представляет собой фундамент будущего отчета, на котором располагаются остальные элементы отчета. Обычно компонент QuickRep располагают на отдельной форме. Этот компонент при помещении на форму представляет собой страницу формата А4 в натуральную величину. Первоначально он имеет имя QuickRep1 и при желании ему можно задать любое имя.

Для того чтобы в отчете отображалось текущее содержание Базы Данных, компонент отчета нужно связать с набором данных TTable или TQuery. Именно для этого служит свойство DataSet.

```
QuickRep1.DataSet:=Table1;
QuickRep2.DataSet:=Query1;
```

Сам отчет, на этапе разработки, состоит из отдельных полос. Каждая полоса представляет собой отдельную часть отчета. В каждой полосе располагаются определенные элементы, которые собственно и формируют внешний вид отчета.

Для создания полос используется свойство Band типа TQuickRepBands. Наличием той или иной полосы можно управлять путем установки логического значения соответствующего подсвойства. Вот список всех полос:

- HasPageHeader - Верхний колонтитул;
- HasTitle - Заголовок отчета;
- HasColumnHeader - Заголовок столбца отчета;
- HasDetail - Область отображения самих данных;
- HasSummary - Итогоги отчета;
- HasPageFooter - Нижний колонтитул.

Полосы так же можно добавлять в отчет и при помощи компонента полосы QRBand. При этом тип полосы устанавливается при помощи свойства BandType.

Параметры страницы отчета определяются свойством Page типа TQRPage. С помощью этого свойства можно установить размер и ориентацию страницы, высоту и ширину страницы, отступы от границ страницы. Эти же свойства можно изменить и при помощи диалогового окна Report Setting. Данное окна можно вызвать двойным щелчком мыши по странице отчета, или командой контекстного меню.

Для печати отчета предназначен метод Print. При вызове данного метода происходит печать отчета без каких-либо запросов и диалоговых окон. Так же существует возможность предварительного просмотра отчета при помощи метода Preview. В результате вызова этого метода на экране появляется специальное окно, в котором отображается отчет со всем его содержимым. При этом, если отчет связан с набором данных, то отчет всегда будет отображать текущее состояние набора данных.

```
// Печать отчета
QuickRep1.Print;
// Предварительный просмотр отчета
QuickRep2.Preview;
```

Фильтры

Для экспорта отчетов в файлы распространенных форматов используются специальные компоненты QuickReport, называемые фильтрами. Фильтры можно также использовать для экспорта данных из базы данных в другой формат.

Классы фильтров входящие в стандартную поставку Delphi:

- TQRTextFilter – экспорт отчета в текстовый файл ASCII. В качестве разделителей полей используются пробелы;
- TQRCSVFilter – экспорт отчета в файл CSV-формата, в котором поля разделяются запятыми, а значения полей заключаются в кавычки. Формат поддерживается электронными таблицами. Свойство Separator позволяет задавать символ разделителя;
- TQRHTMLFilter – экспорт отчета в файл HTML-формата.

Классы фильтров входящие в профессиональную поставку:

- TQRExcelFilter – экспорт отчета в формат электронной таблицы MS Excel;
- TQRRTFFilter – экспорт отчета в формате RTF;

- TQRWMFFilter – экспорт отчета в графический файл формата WMF (Windows Metafile).

Для использования возможностей экспорта достаточно просто поместить компоненты фильтров на компонент TQuickRep. После этого из окна предварительного просмотра отчет можно будет сохранить в виде файла в форматах, поддерживаемых фильтрами, размещенными на TQuickRep.

Задание:

Разработать поиск и другие функций со сложными запросами (соединение, объединение, агрегатные функции, подзапросами):

1. Создать формы по формированию отчетов с использованием элементов QuickReports.
2. Используя ADOTable произвести фильтрацию и сортировку данных и отбразить в виде отчета.
3. Создать отчет, использующий поиск, с помощью ADOQuery.

Требования к отчету:

Отчет должен быть выполнен в соответствии с правилами оформления отчетов указанном в приложении 1.

Отчет по выполненной работе должен содержать:

1. Тему лабораторной работы.
2. Цель лабораторной работы.
3. Задание.
4. Схему реляционной базы данных из лабораторной работы №2.
5. Основные запросы на SQL для отчетов
6. Скриншоты работы программы ободражающие формирование и работу с отчетами.

Контрльные вопросы:

1. Как создаются отчеты?
2. Что такое и в чем предназначение QBand?
3. В чем преимущество использования QuickReport?
4. Какие элементы необходимы для построения отчета в среде C++ Builder?

Литература:

1. C. J. Date – An Introduction to Database Systems – Addison-Wesley Professional – 2003 – 1024 p
2. Мартин Грабер. – Введение в SQL. изд. Лори, 382 стр. 1992 г.
3. Jarrod Hollingworth, Bob Swart, Mark Cashman, Paul Gustavson – Borland C++ Builder 6 Developer's Guide 2nd Edition – 2002 – SAMS Publishing – 1128p

ГЛОССАРИЙ
Русский

1. **Администратор базы данных (АБД)** – это лицо или группа специалистов, знакомых с теорией построения информационных систем с базами данных и со спецификой предметной области данной информационной системы. Администратор базы данных осуществляет централизованное управление базой данных посредством конкретной СУБД.

2. **Аномалии операций с отношениями базы данных** – нежелательные эффекты, которые могут иметь место при осуществлении операций, связанных с изменением данных, а именно операций INSERT (вставка кортежа), DELETE (удаление кортежа) и UPDATE (обновление значений атрибута какого-либо кортежа).

3. **База данных** – совокупность связанных данных, организованных по определенным правилам, предусматривающим общие принципы описания, хранения и манипулирования, независимая от прикладных программ. База данных является информационной моделью предметной области. Обращение к базам данных осуществляется с помощью системы управления базами данных (СУБД).

4. **База данных (БД)** – это совокупность данных, организованная по определенным правилам, предусматривающим общие принципы описания, хранения и манипулирования данными.

5. **Домен реляционного отношения** – это множество скалярных (атомарных, неделимых) элементов, из которого могут браться значения конкретного атрибута.

6. **Запрос** – процесс обращения пользователя к БД с целью ввода, получения или изменения информации в БД.

7. **Логическая структура БД** – определение БД на физически независимом уровне, ближе всего соответствует концептуальной модели БД.

8. **Локальная автономность** – означает, что информация локальной БД и связанные с ней определения данных принадлежат локальному владельцу и им управляются.

9. **Пользователь БД** – программа или человек, обращающийся к БД на ЯМД.

10. **Словарь (справочник) базы данных** – специализированная подсистема СУБД, предназначенная для централизованного хранения единообразной информации обо всех хранимых в БД данных, используемой СУБД для доступа к данным

11. **Сущность** – конкретный опознаваемый предмет, элемент, единица (реальные или абстрактные), имеющие четко определенное функциональное назначение, четко определенные границы в данной предметной области, обусловленные контекстом задач, в рамках которых представляет интерес информация о данной предметной области.

12. **Схема базы данных** – это описание базы данных в контексте конкретной модели данных

13. **Топология БД, структура распределенной БД** – схема распределения физической БД по сети.

14. **Транзакция** – это логическая единица работы СУБД, последовательность операторов манипулирования данными, выполняющаяся как единое целое и переводящая базу данных из одного согласованного состояния в другое. Лозунг транзакции “все или ничего.

15. **Удаленный запрос** – запрос, который выполняется с использованием модемной связи.

16. **Экземпляры связей** – связи, имеющие место между конкретными экземплярами объектов.

17. **Язык манипулирования данными (или язык запросов к базе данных) (Data Manipulation Language - DML)** – набор команд, реализующих операции манипулирования данными.

18. **Язык описания данных (Data Definition Language - DDL)** – это язык высокого уровня, предназначенный для задания схемы базы данных. С его помощью описываются типы данных, подлежащих хранению в базе или выборке из нее, структура данных и их связи между собой.

Английский

1. **The database administrator (DBA)** is a person or group of specialists who are familiar with the theory of building information

systems with databases and with the specifics of the subject area of this information system. The database administrator performs centralized management of the database through a specific DBMS.

2. **Anomalies in operations with database relationships** are undesirable effects that may occur when performing operations related to data changes, namely INSERT operations, DELETE and UPDATE (update of attribute values of a tuple).

3. **Database -** a set of related data, organized according to certain rules, providing general principles of description, storage and manipulation, independent of the application programs. The database is an information model of the domain. The database is accessed through a database management system (DBMS).

4. **The domain of a relational relation** is a set of scalar (atomic, indivisible) elements from which the values of a particular attribute can be taken**.**

5. **Request -** the process of accessing the user to the database in order to enter, receive or modify information in the database.

6. **Logical structure of the database -** the definition of the database at a physically independent level, closest corresponds to the conceptual model of the database.

7. **Local autonomy mea**ns that the information of the local database and associated data definitions belong to the local owner and are managed**.**

8. **The user of the database** is a program or person accessing the database on the DML.

9. **The database dictionary** is a specialized DBMS subsystem intended for centralized storage of uniform information about all data stored in the database used by the DBMS to access data

10. **The entity** is a concrete identifiable object, an element, a unit (real or abstract), having a well-defined functional purpose, clearly defined boundaries in the given subject area, conditioned by the context of the tasks within which information on the given subject area is of interest.

11. **A database schema** is a description of a database in the context of a particular data model

12. **Database topology, distributed database structure -** the scheme of physical database distribution over the network.

13. **A transaction** is a logical unit of the DBMS operation, a sequence of data manipulation operators that runs as a unit and

transfers the database from one agreed state to another. The slogan of the transaction is "all or nothing."

14. **A remote request** is a request that is made using a dial-up connection.

15. **Link instances** are the relationships that occur between specific instances of objects.

16. **Data manipulation language (or data query language) (Data Manipulation Language (DML**) is a set of commands that implement data manipulation operations.

17. **The Data Definition Language (DDL)** is a high-level language intended for specifying a database schema. It describes the types of data to be stored in a database or a sample from it, the data structure and their relationships with each other.

Узбекский

1. **Ma'lumotlar bazasi administratori (DBA) -** bir kishi yoki ma'lumotlar bazalari va axborot tizimining o'ziga xos domen bilan axborot tizimlari nazariyasi bilan tanish mutaxassislar bir guruh. Ma'lumotlar bazasi ma'muri ma'lumotlar bazasini markazlashtirilgan boshqarish uchun maxsus DBMS orqali amalga oshiradi.

2. **Ma'lumotlar bazasi munosabatlar bilan anomaliyalar operatsiyalar -** ma'lumotlar, masalan, INSERT operatsiyalar (Insert shamlardan) o'zgarishi bilan bog'liq operatsiyalarni amalga oshirishda yuzaga kelishi mumkin kiruvchi ta'sir, o'chirish (shamlardan nomlash), va UPDATE (yangilash bir shamlardan xususiyati qiymatlari).

3. **Ma'lumotlar bazasi -** ilovaning mustaqil bayon qilish, saqlash va qayta ishlash, umumiy tamoyillari ta'minlash muayyan qoidalar asosida tashkil tegishli ma'lumotlarni to'plash. Ma'lumotlar bazasi domenning axborot modeli. Ma'lumotlar bazasiga ma'lumotlar bazasini boshqarish tizimi orqali kirish mumkin.

4. **Relyatsion algebraning maydoni,** ma'lum bir xususiyatning qiymatlari qabul qilinishi mumkin bo'lgan skalar (atom, bo'linmas) elementlarning to'plamidir.

5. **Talab -** ma'lumotlar bazasida ma'lumotlarni kiritish, olish yoki o'zgartirish uchun foydalanuvchini ma'lumotlar bazasiga kirish jarayoni.

6. **Ma'lumotlar bazasining mantiqiy tuzilishi -** jismonan mustaqil darajada ma'lumotlar bazasini aniqlash, eng yaqin ma'lumotlar bazasining kontseptual modeliga mos keladi.

7. **Mahalliy muxtoriyat –** mahalliy ma'lumotlar bazasi va unga aloqador ma'lumotlar ta'riflari mahalliy egasiga tegishli ekanligini anglatadi va boshqariladi.

8. **Ma'lumotlar bazasi foydalanuvchisi –** dastur yoki NMD bazasiga kirgan shaxs.

9. **Ma'lumotlar bazasi lug'ati -** ma'lumotlar bazasiga kirish uchun DBMS tomonidan foydalaniladigan ma'lumotlar bazasida saqlanadigan barcha ma'lumotlar haqida yagona ma'lumotni markaziy saqlash uchun mo'ljallangan maxsus DBMS kichik tizimidir

10. **Mazmuni –** aniqlangan aniq funktsional maqsadga ega bo'lgan, ushbu mavzu bo'yicha aniq belgilangan chegaralarni aniqlaydigan aniq ob'ektni, elementni, birlikni (haqiqiy yoki mavhum), ushbu mavzu bo'yicha axborotni qiziqtiradigan vazifalar konteksti bilan shartlangan.

11. **Ma'lumotlar bazasi sxemalari –** ma'lumotlar bazasining ma'lum bir ma'lumot modeli kontekstidagi tavsifi.

12. **Ma'lumotlar bazasi topologiyasi, tarqalgan ma'lumotlar bazasi tuzilishi –** tarmoq orqali jismoniy ma'lumotlar bazasini tarqatish sxemasi.

13. **Tranzaksiya –** DBMS operatsiyaning mantiqiy birligi bo'lib, bir birlik sifatida ishlaydigan ma'lumotlar bazasini boshqaruvchi operatorlarning ma'lumotlar bazasini boshqasiga o'zgartiradi. Jurnalning shiori "butun yoki hech narsa".

14. **Masofaviy so'rov –** bu modemning aloqasidan foydalanib, so'rov**.**

15. **Bog'lanish misollari** ob'ektlarning muayyan nusxalari orasidagi o'zaro bog'liqlikdir.

16. **Ma'lumotni manipulyatsiya tili (yoki ma'lumotlar so'rovining tili) (Data Manipulation Language (DML) -** ma'lumotni manipulyatsiya operatsiyalarini bajaradigan buyruqlar majmui**.**

17. **Ma'lumotni aniqlash tili (DDL) -** ma'lumotlar bazasi sxemasini aniqlash uchun mo'ljallangan yuqori darajadagi tildir. Ma'lumot bazasida yoki undan namunada saqlanadigan ma'lumotlar turlarini, ma'lumotlar strukturasini va ularning o'zaro munosabatlarini tavsiflaydi.

Список использованных литературы

1. Ўзбекистон Республикасини янада ривожлантириш бўйича ҳаракатлар стратегияси тўғрисида. Ўзбекистон Республикаси Президентининг ПФ – 4947 – сон фармони. Тошкент, 2017 йил 7 феврал.
2. Роб П. Системы баз данных: проектирование, реализация и управление (5-е издание) издательство "БХВ - Санкт-Петербург" ·1200 стр, 2003 г. ·.
3. Григорьев Ю.А., Плутенко А.Д.. Жизненный цикл проектов распределенных баз данных. Благовещеснк АмГУ, 1999.
4. Дунаев С.С. Доступ к базам данных и техника работы в сети. Практические приемы современного программирования. М.: Диалог – МИФИ, 1999.
5. Дж.Ульман, Дж Уидом. Введение системы баз данных. Пер.с англ. М.: «Лори», 2000.
6. Диго С.М. Базы данных Проектирование и использование. издательство "Финансы и статистика" · 592 стр, 2005 г.
7. Конноли Т., Брегк К. Базы данных, проектирование, реализация и сопровождения, теория и практика, Университет Пейсли, Шотландия, изд. М.- СПБ.- Киев, 2003.
8. Четвериков, В. Н. Базы и банки данных [Текст] : учебник для вузов по спец. "Автоматизир. системы управления" / Г. И. Ревунков, Э. Н. Самохвалов. - М. : Высш. шк., 1987. - 248 с. : ил. - Библиогр.: с.246 (14 назв.). Предм. указ.: с. 247.
9. А. Д. Хомоненко, В. М. Цыганков, М. Г. Мальцев Базы данных [Текст] : учебник для вузов / - 4-е изд., доп. и перераб. - СПб : Корона принт, 2004. - 736 с. - 1 экз.
10. Б. Я. Советов, В. В. Цехановский, В. Д. Чертовский. Базы данных. Теория и практика [Текст] : учебник для студ. вузов / - М. : Высш. шк., 2005. - 463 с. : ил. - Список лит. с. 459-460. - 2 экз.
11. Мирзиёев Ш.М. Буюк келажагимизни мард ва олижаноб ҳалқимиз билан бирга қурамиз. 2017 йил.
12. Мирзиёев Ш.М. Қонун устуворлиги ва инсон манфаатларини таъминлаш - юрт тараққиёти ва ҳалқ фаровонлигининг гарови. 2017 йил.
13. Мирзиёев Ш.М. Эркин ва фаровон, демократик Ўзбекистон давлатини барпо этамиз. 2017 йил.

14. Астахова И.Ф., Толстобров А.П. СҚЛ в примерах и задачах. Учебное пособие. Новое знание, 176 стр, 2002 г.
15. Полякова. Л.Н. Основы СҚЛ. Курс лекций. Учебное пособие. издательство "ИНТУИТ.РУ" · 368 стр, 2004 г.
16. Бен Форта Освой самостоятельно СҚЛ. 10 минут на урок (3-е издание) издательство "Вильямс" 288 стр, 2005 г. ·
17. Клыков Ю. И. Банки данных для принятия решений [Текст] : монография / Ю. И. Клыков, Л. Н. Горьков. - М. : Сов. радио, 1980. - 208 с. - 1 экз.
18. Шомье Ж. Банки данных [Текст] : использ. электрон. вычисл. техники / Пер. с фр. Ю.Л. Смирнова ; Под ред. Б.А. Щукина. - М. : Энергоиздат, 1981. - 70 с. : ил. - (Б-ка по автоматике ; вып.619). - Библиогр.: с. 69. - 2 экз.
19. Аппак М. А. Базы данных в АСУ-связь [Текст] : монография / М. А. Аппак. - М. : Радио и связь, 1987. - 80 с. - 2 экз.
20. Лори, Питер Базы данных для микроЭВМ [Текст] : монография / Пер. с англ. Ю.К. Трубина. - М. : Машиностроение, 1988. - 135 с. : ил. - 1 экз
21. Хансен, Гэри Базы данных [Текст]: разработка и управление / Пер. с англ. под ред. С. Каратыгина. - М. : БИНОМ, 1999. - 699 с. - 1 экз.
22. Марков, А. С. Базы данных. Введение в теорию и методологию [Текст] : учебник для студ. / А. С. Марков, К. Ю. Лисовский . - М. : Финансы и статистика, 2004. - 512 с. : ил. - Рекомендуемая лит. с. 431-435. -Предм. указ. с. 499-511. - 1 экз.
23. Маллинс Крейг С. Администрирование баз данных. Полное справочное руководство по методам и процедурам. Пер. с англ. Издательство: Кудиц-Образ. Год издания: 2003.
24. Кренке Д. Теория и практика построения баз данных. 8-издание, СПб.: Питер, 2003.-880с.
25. Конноли Томас, Каролин Бегг Базы данных .Проектирование, реализация и сопровождение. Теория и практика. 3-издание – М. : Изд.дом Вильямс - 2003. – 1440 с.
26. Дунаев В.В. Базы данных. Язык СҚЛ – СПб.: --БХВ-Петербург,2006.-288с.
27. Григорьев Ю.А. Банки данных. Учеб.для вузов.- М.: Изд-во МГТУ им. Н.Э. Баумана, 2002. -320с.

28. Назирова Е.Ш., Усмонов Ж.Т. Маълумотлар базаси ва банки фанидан лаборатория ишлари боъйича услубий кўрсатма.// Лаборатория ишларини бажариш учун услубий кўрсатма. Тошкент, 2014, 40 бет
29. my.gov.uz
30. egov.uz
31. lex.uz
32. www. Ziyonet.uz
33. www.library.tuit.uz
34. www.intuit.ru