МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

**Методические указания**
по выполнению лабораторных работ
по дисциплине «**ТЕХНОЛОГИИ ПРОЕКТИРОВАНИЯ И АДМИНИСТРИРОВАНИЯ БАЗ ДАННЫХ**»
для студентов специальности 10.05.01 Компьютерная безопасность
специализация «Информационно-аналитическая и техническая экспертиза компьютерных систем»

Ставрополь
2026

# Оглавление

# 1. Системы основанные на файлах

*Вид занятия – лабораторная работа*
*Время занятия – 4 часа*
*Программное обеспечение – C++*

Одна из самых первых фирм, нацелившая свой научный потенциал на исследование способов хранения и обработки больших массивов данных называлась North American Aviation (NAA)[1]. В середине 60-х годов XX века ее инженеры разработали программное обеспечение (ПО) с очень серьезным названием: Generalized Update Access Method (GUAM). Это было не что иное, как прототип представления данных в виде древообразных структур – **иерархическая модель данных** (*hierarchical data model*). Несколько позднее идея, заложенная в GUAM, получила свое развитие в стенах компании IBM. Результат труда IBM реализовался в рамках ПО Information Management System (IMS). Плоды совместного творчества NAA и IBM не только нашли свое отражение в одном из ключевых теоретических направлений развития БД, но и внесли существенный вклад в более осязаемый проект – благодаря IMS проводились расчеты первых полетов американского космического корабля Apollo на Луну.

Однако, не смотря на существенный успех, NAA и IBM нисколько не монополизировали рынок будущих систем управления базами данных. Примерно в это же время компания General Electric разработала альтернативную систему Integrated Data Store (IDS). С этим проектом связано одно из ключевых имен исследователей СУБД – имя Чарльза Бачмана. Именно его идеи организации данных в дальнейшем воплотились в принципиально новое направление баз данных – **сетевую модель данных** (network data model).

Раз мы заговорили об именах, то нельзя не упомянуть имя ученого, внесшего существенный вклад в становление **реляционной модели данных** (*relational data model*). Это Э.Ф. Кодд, сотрудник одной из научно-исследовательских лабораторий IBM. Благодаря его теоретическим исследованиям в 1976 году в свет вышел проект System-R, послуживший родоначальником всех современных реляционных СУБД.

## Файловые системы, принципы построения

Одним из направлений развития программного обеспечения, внесшего существенный вклад в становление современных БД, стали **системы, основанные на файлах** (file-based system) или, как их чаще называют **файловые системы**. Знание основных идей, заложенных в файловых системах, и присущих им недостатков позволит нам более глубоко разобраться со способами хранения данных в современных БД и, что очень важно, избежать ошибок свойственных файловым системам.

> **ФАЙЛОВАЯ СИСТЕМА** (**ФС**) – программа (набор программ) позволяющая добавлять, редактировать, удалять и просматривать данные. Каждая программа определяет свой собственный формат хранения данных и свой собственный способ управления этими данными.

Первые разработки ФС были нацелены на решение элементарных задач – организацию обработки данных, хранившихся в обычных ручных картотеках. С точки зрения программирования решение подобных задач обычно производится в рамках разработки программ, применяющих типизированные файлы.

## Недостатки файловых систем

Самый первый недостаток файловых систем проявился очень скоро. Вследствие того, что каждый программист выбирал свой собственный формат хранения данных, создаваемые ими программы и типизированные файлы были просто **несовместимы**. Действительно, разрабатывая структуру, пусть даже предназначенную для описания одних и тех же данных, каждый программист шел по своему пути. Один под поле фамилии отводил 20 символов, другой тридцать... Варьировались не только размерности полей, но и их порядок следования. Сколько разработчиков – столько и мнений…

Второй недостаток также не заставил себя долго ждать. Закладываемая на стадии разработки ПО жесткая физическая структура данных приводила к созданию весьма неповоротливых приложений. Так если в нашем примере для хранения имени студента отводилось 10 символов, то появление необходимости увеличить (или уменьшить) это поле даже на один единственный символ влекло целый каскад проблем. Программист должен был не только переписать все свои программы, но и разработать конвертор для ранее набранных данных в новый формат. Эта проблема называется **зависимостью от данных**.

Вследствие первой пары недостатков файловые системы “обогатились” очередной отрицательной чертой. Несовместимость файлов и зависимость программ от данных привели к тому, что информационные файлы **дублировались** по всему предприятию. Один программист написал программу учета студентов для деканата, другой для отдела кадров, третий для кафедры. Даже если они договорились об едином типе данных, все равно, везде хранится одно и тоже, что весьма неэффективно. Другой аспект этого недостатка

[1] Сегодня эта компания называется Rockwell International

еще важнее – данные могут стать противоречивыми. Деканат отчислил нерадивого студента, а бухгалтерия продолжает выдавать ему стипендию, ведь в их версии данных этот студент продолжает учиться. Снежный ком имеет особенность к накоплению своей массы. В конечном итоге дублирование данных приводит к их **изолированности**. Во всех отделах и службах предприятия, обрабатывая изначально одинаковые данные, пользователи наполняют их разным (и что самое страшное не всегда корректным) содержанием.

Напоследок расскажем о пятом недостатке файловых систем. Он связан с лавинообразным **разрастанием количества приложений**, обслуживающих файловую систему. В первую очередь это объясняется тем, что любая элементарная задача (например, появление нового запроса к данным) приводит к необходимости кардинальной переработке старого приложения либо к созданию нового.

Таким образом, файловые системы обладают целой плеядой недостатков:

1. Несовместимость данных.
2. Зависимость от данных.
3. Дублирование данных.
4. Изолированность данных.
5. Быстрое разрастание количества приложений.

Это далеко не полный перечень минусов файловых систем, однако их вполне достаточно для того, чтобы сделать вывод о необходимости дальнейшего совершенствования способов хранения данных.

## Задание

В среде C++ Builder (или в другой среде проектирования по согласованию с преподавателем) разработайте консольное приложение системы, основанной на файлах. Приложение должно позволять: добавлять, редактировать, удалять и просматривать данные следующего вида:

В.1. Сотрудники фирмы (фамилия, имя, отчество, должность, зарплата).

В.2. Список товаров (наименование, стоимость за единицу, количество, поставщик).

В.3. Телефонный справочник (фамилия, инициалы, номер телефона).

В.4. Домашняя библиотека (автор, название произведения, жанр).

В.5. Телепередачи (название телевизионного канала, название передачи, дата и время эфира).

В.6. Фонотека (исполнитель, название концерта, год выхода).

В.7. Справочник адресов (фамилия, инициалы, город, улица, дом, квартира).

В.8. Склад телевизоров (фирма-изготовитель, модель, диагональ экрана, разрешение экрана).

В.9. Кадры (должность, фамилия и инициалы, номер кабинета, рабочий телефон).

В.10. Тарифы мобильного оператора (название тарифа, число доступных минут, число доступных СМС, стоимость в месяц).

**Советы:**

1. На начальном этапе выполнения лабораторной работы хорошей идеей станет разработка структуры **`struct`** для хранения отдельной записи (см. лаб.работу № 1 “Методы программирования”).
2. Для организации доступа к бинарному файлу допускается использовать низкоуровневые функции для работы с файлами или наиболее подходящие классы (например `TFileStream` в C++ Builder).

**Внимание!**

Разрабатываемая система, основанная на файлах, должна хранить данные в **бинарном** (двоичном файле)!

# 2. Реляционные таблицы

*Вид занятия – лабораторная работа*
*Время занятия – 2 часа*
*Программное обеспечение – среда Microsoft Access*

Подавляющее большинство недостатков характерных для файловых систем появились вследствие всего из-за двух причин. **Первая** из них в том, что в файловых системах **способы описания данных** (их структуры) **хранились внутри исполняемого кода приложения**. **Вторая** причина заключается в **отсутствии универсального инструмента, обеспечивающего доступ к данным**, хранящимся в любом типизированном файле. Таким образом, до содержащейся в файле информации способно добраться только его "родное" приложение.

Выявив причины первых неудач, разработчики программного обеспечения пришли к однозначному выводу: **программа и данные должны стать независимыми** (program-data independence). Но каким образом этого добиться? Достаточно просто – надо вывести описание структур из программы. Данные должны содержать не только информацию, но и ее характеристики. Прежде чем обратиться к информации, приложение знакомится с метаданными (так называется описание данных). выяснив используемые типы данных, их размерности, и если необходимо другие дополнительные параметры, программа без проблем получала доступ к основной информации. Описание данных программисты называют **системным каталогом** (*system catalog*).

Вынос описаний данных из состава приложения стал ключевым шагом, навстречу будущим базам данных.

> ***Определение***
> **База данных** – это совместно используемое пользователями единое хранилище логически связанных данных и их описаний.

База данных не жизнеспособна сама по себе, она нуждается в помощнике – программе обслуживания. Эта программа должна выполнять посредническую роль при организации общения конечного пользователя и находящихся в базе данных. Такая программа называется системой управления базами данных.

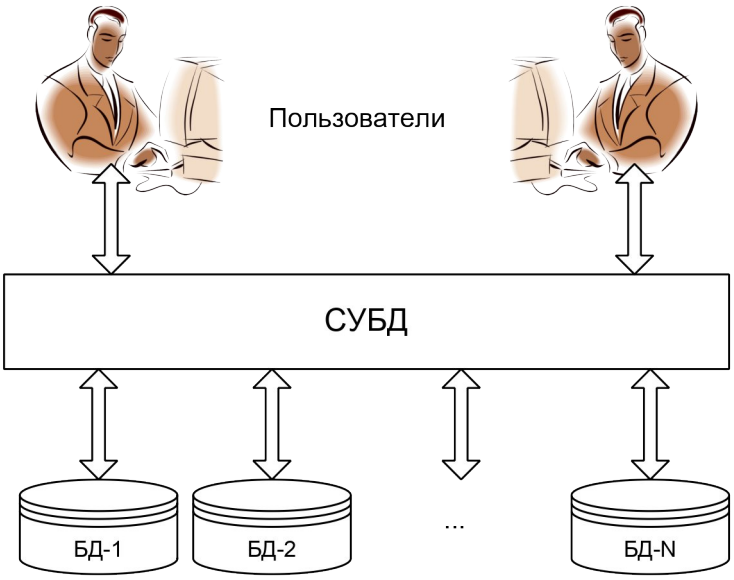



Рис. 2.1. Взаимодействие пользователя и БД

> ***Определение***
> **Система управления базами данных (СУБД)** – это программное обеспечение, позволяющее создавать БД, хранить и извлекать информацию из БД, а также обеспечивать контроль за доступом к БД.

На плечи СУБД возложено большое количество разнообразных задач. Пока мы упомянем лишь то, что СУБД дает пользователю абстрактное представление данных, скрывая особенности физического хранения данных и управления ими.

В качестве примеров современных реляционных СУБД отметим системы, работающие по технологии клиент-сервер: Oracle, Informix, InterBase, Microsoft SQL Server и однопользовательские системы Microsoft Access, SQLite.

## Реляционная таблица

Родоначальником реляционной модели данных считается Э.Ф.Кодд (E.F.Codd). Именно этот ученый в 1970 году опубликовал фундаментальную статью "Реляционная модель данных для больших совместно используемых банков данных". Анонсированная в статье идея реляционной модели основывается на теории множеств и логике предикатов. Фундаментальным понятием реляционной модели считается отношение. Отношения предназначены для хранения информации об объектах, представленных в базе данных. Пример отношения представлен на рисунке 2.2.

***Определение***
**Отношение** – двумерная таблица, состоящая из столбцов и строк.


Рис. 2.2. Пример отношения

В современной литературе при описании отношения зачастую встречаются разные определения. Стоит знать, что в качестве синонима “строка отношения” применяются термины “запись” и “кортеж”. Кроме термина “столбец отношения” можно услышать “атрибут” и “поле”. “Отношение” также имеет свой синоним – это “таблица”.

## Определение домена

Любая СУБД позволяет формировать столбец таблицы таким образом, чтобы он стал максимально удобным для своих данных. На стадии проектирования таблицы разработчик обязан уделить этому самое скрупулезное внимание, такой процесс называется **определением домена**.

***Определение***
**ДОМЕН** – это набор допустимых значений для одного или нескольких столбцов отношения.

Благодаря домену разработчик БД получает возможность централизованно определять смысл и источник значений, которые могут получать поля таблицы. Кроме предотвращения ввода некорректных данных домен поможет избежать семантически некорректных операций. Например, бессмысленного сравнения цены товара и номера телефона.

Характеристики любого столбца реляционной таблицы всегда определяется индивидуальными свойствами данных, которые мы намерены хранить в этом поле. Если столбец предназначен для фамилии или имени человека, то мы выберем текстовый тип данных и ограничим его 15-20 символами. Если речь идет о количественных значениях, то мы воспользуемся числовыми типами данных. В таблице 2.1 представлены основные типы данных, которые могут иметь поля в Microsoft Access.

*Таблица 2.1. Основные типы данных Microsoft Access*

| ***Тип данных*** | ***Назначение*** | ***Размер*** |
|---|---|---|
| Текстовый | Хранение текстовых данных. | До 255 символов |
| Поле MEMO | Большое количество неформатированных текстовых данных. | До 65 536 символов. |
| Числовой | Хранение целых и действительных значений. | 1, 2, 4 или 8 байт |
| Денежный | Денежные величины. | 8 байт |
| Дата/время | Значения дат и времени. | 8 байт |
| Счетчик | Автоинкрементное значение. | 4 байта |
| Логический | Булевы значения (истина/ложь). | 1 бит |

В отличие от всех языков программирования базы данных построены на 3-хзначной логике. Помимо значений `True` (истина) и `False` (ложь) мы можем столкнуться с определителем `NULL`. Определитель `NULL` воспринимается как неопределенное (или неизвестное) значение.

> **Определитель NULL** указывает, что значение поля в настоящий момент неизвестно или недопустимо для записи

Определившись с типом данных и их размерностью, программист может наложить на данные дополнительные ограничения:

Поставить признак обязательного поля. Если признак не установлен, то надо быть готовым к тому, что пользователь не станет заполнять поле данными. Таким образом, его содержимое станет неопределенным – `NULL`.

Указать значение, передаваемое в поле по умолчанию.

Ввести ограничения на допустимые значение (например, диапазон значений, которые разрешено передавать в поле).

Назначить маску ввода, определяющую жесткий порядок ввода и представления значений.

## Создание таблиц в среде Microsoft Access

Попробуем повторить простую задачу по учету студентов ВУЗа, но на этот раз не с помощью файловых систем, а на основе таблиц. В качестве инструмента воспользуемся услугами среды Access – программного продукта Microsoft предназначенного для построения реляционных баз данных малой степени сложности.

Запустите приложение Access. Воспользовавшись пунктом меню: **Файл – Создать**. создайте новый файл базы данных и сохраните его под любым именем, например “students.accdb”.

Access предлагает два способа создания таблиц (рис. 2.3):

1. В режиме конструктора.
2. Путем ввода данных.

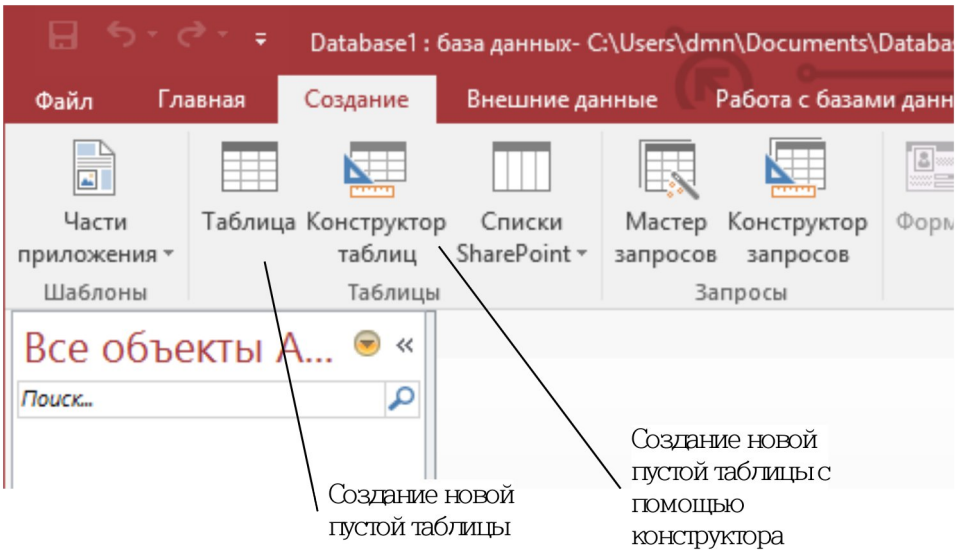



Рис. 2.3. Режимы создания таблиц в среде Access

Мы направимся по наиболее функциональному пути и выберем режим конструктора. Именно этот режим предоставляет пользователю максимальные возможности по формированию будущей таблицы. Двойной щелчок по строке “Создание таблицы в режиме конструктора” вызовет окно конструктора (рис. 2.4).

> ***Внимание!***
> При назначении имен столбцам и выборе названия таблицы старайтесь не применять символы национальных алфавитов.

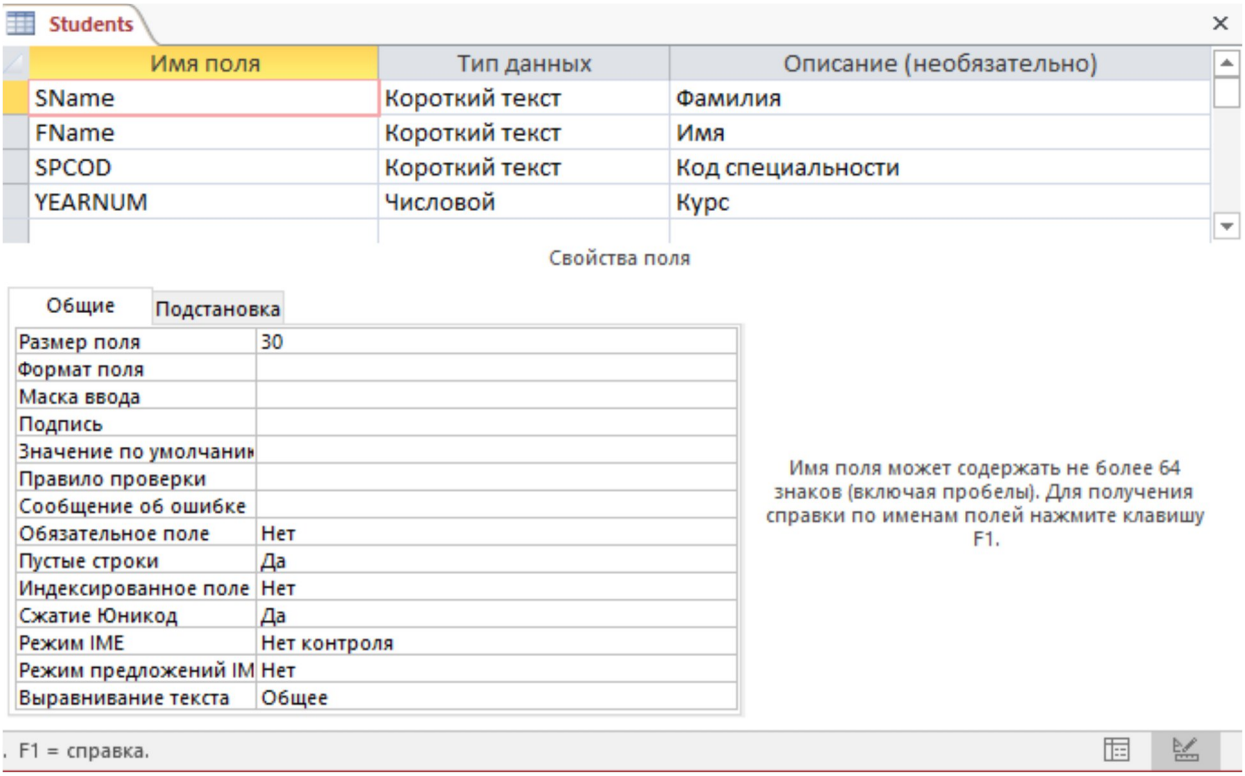



Рис. 2.4. Конструктор таблицы

В конструкторе назначаются имена полей (столбцов) будущей таблицы, определяются типы данных и их ограничения. Например, для хранения фамилии мы определяем поле “SName”, тип данных “Текстовый”, размер столбца – 30 символов.

Созданная нами таблица пока не может претендовать на высокое звание реляционной таблицы (R-таблицы). Для этого она должна пройти несколько стадий нормализации, но уже сейчас в ней присутствуют некоторые родовые черты R-таблиц:

1. Таблица поименована.
2. Каждое поле таблицы имеет уникальное имя.
3. Каждое поле содержит только атомарное (неделимое) значение.
4. Значения полей описываются доменами.

## Задание

В среде Microsoft Access в режиме конструктора создаете таблицу, предназначенную для хранения следующих данных (по вариантам):

1. Деловые знакомства (фамилия, имя, отчество, пол, название фирмы, должность, дата рождения, рабочий телефон, E-Mail).
2. Склад магазина (наименование товара, артикул, стоимость, количество, поставщик товара, номер телефона поставщика, дата поставки, место хранения (номер стеллажа)).
3. Телефонный справочник (фамилия, инициалы, номер телефона, адрес абонента).
4. Домашняя библиотека (автор, название произведения, жанр, издательство, год издания, количество страниц).
5. Домашняя фонотека (исполнитель, название концерта, год выхода, формат данных (CD, DVD, mp3), страна).
6. Список студентов (фамилия, имя, отчество, пол, специальность, номер студ. билета, дата рождения, дата поступления, контактный телефон).
7. Кинофильмы (Название, киностудия, режиссер, жанр, дата выхода на экран).
8. Расписание электричек на неделю (номер поезда, станция отправления, станция прибытия, день недели, время отправления, время прибытия).

Настройте доменные ограничения для каждого из столбцов таблицы, там, где это необходимо установите значения по умолчанию, разберитесь с возможностями “маски ввода”, признаком обязательности значения, возможности подстановки значений (режим “Поле со списком”), и т.д.

Заполните таблицу строками демонстрационных данных.

# 3. Ключи, организация связи между таблицами

*Вид занятия – лабораторная работа*
*Время занятия – 2 часа*
*Программное обеспечение – среда Microsoft Access*

Создаваемые нами на прошлом занятии таблицы нельзя назвать в полной мере реляционными. Они не удовлетворяют ряду требований, предъявляемых к реляционным отношениям. В 3 работе мы уделим внимание двум из них:

1. Обеспечению уникальности строк в R-таблице.
2. Устранению избыточности данных в R-таблице.

## Реляционные ключи

Реляционная модель требует, чтобы каждая отдельная строка отношения уникально идентифицировалась по значениям, содержащихся в ее столбцах.

Рассмотрим таблицу, предложенную на рисунке 3.1.

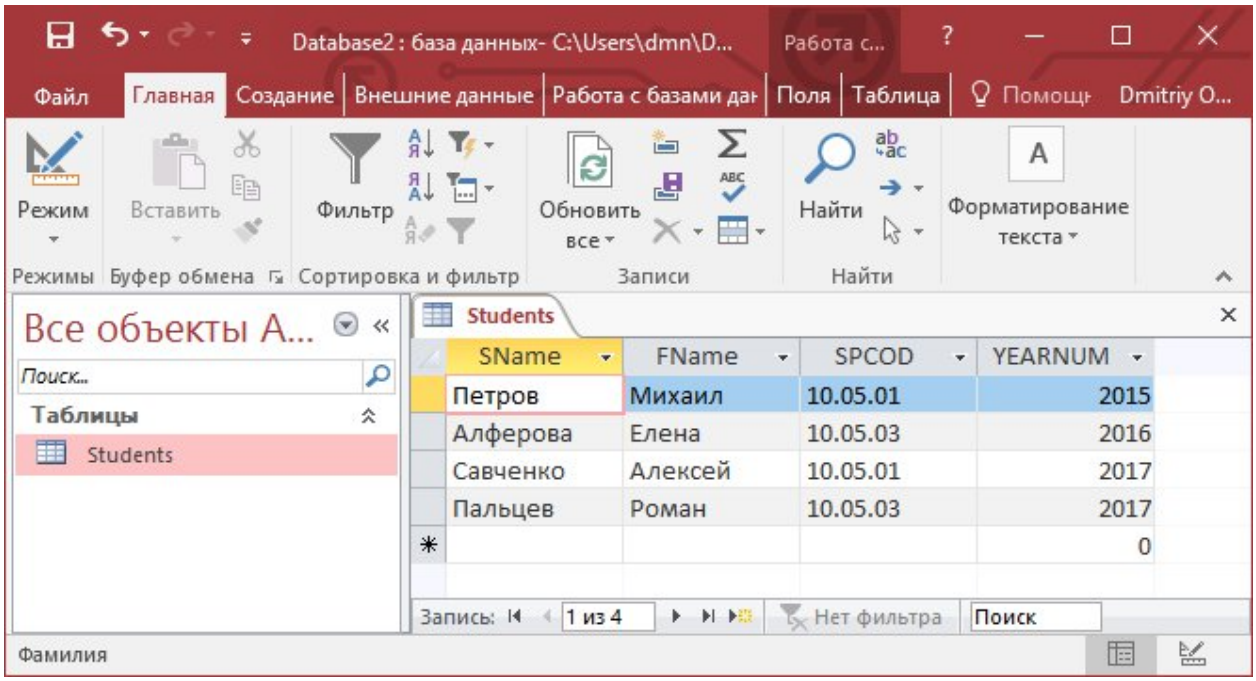



Рис. 3.1. Данные таблицы Students

В таблице содержится только 4 строки. Технически все эти записи уникальны, ведь нет ни одного повторения в содержащейся в строках информации. Но это не добавляет нам уверенности, в том, что такая благоприятная ситуация будет продолжаться и дальше. Можно ли с вероятностью в единицу исключить возможность поступления на один и тот же курс, на одну и ту же специальность двух человек с одинаковыми именами и фамилиями? Допустим, что кроме Петрова Михаила (с отчеством Алексеевич) в его же группе учится Петров Михаил Антонович. Внеся данные о втором Петрове в таблицу, мы сразу получим две абсолютно одинаковые строки. С этого момента никто не сможет отличить записи друг от друга. С точки зрения реляционной модели это недопустимо. Элементарный практический пример. Допустим, что второй Петров не сдал сессию и подлежит отчислению. Вопрос: какую строку из таблицы следует удалить?

Что делать, чтобы добиться уникальности каждой записи в таблице? Добавить в наше отношение поле отчества (LName)? Хорошее решение, но согласитесь, что все-таки, можно допустить и такое совпадение, что нам встретятся полные тески – Ивановы Иваны Ивановичи. Надо идти дальше – ввести в таблицу даты рождения, либо номер паспорта, а еще лучше то и другое. Одним словом, обеспечение уникальности процесс творческий, требующий опыта и интуиции.

***Определение***
**Суперключ (superkey) –** это столбец (множество столбцов), которые единственным образом идентифицирует запись в таблице

Вполне вероятна и обратная ситуация, что в одной и той же строке отношения содержится несколько полей (или групп полей), каждое из которых в отдельности уникальным образом идентифицирует запись в таблице. Это могут быть номер паспорта, индивидуальный номер налогоплательщика, номер мобильного телефона. В таком случае говорят о существовании нескольких **потенциальных ключей**.

Задача разработчика базы данных заключается в выборе (или создании) наиболее подходящего ключа в отношении и назначении его на роль первичного ключа таблицы.

> ***Определение***
> **Первичный ключ** – это потенциальный ключ, который выбран для уникальной идентификации записей внутри таблицы

Первичный ключ может включать одно поле таблицы или несколько ее полей. В последнем случае говорят о **составном первичном ключе**.

Многие СУБД позволяют использовать в качестве ключа поле автоинкрементного типа. Подобная возможность имеется и в Access, для этого предназначен счетчик. Для присвоения полю признака первичного ключа щелкните правой кнопкой мышки по имени поля и выберите пункт меню “ключевое поле”.

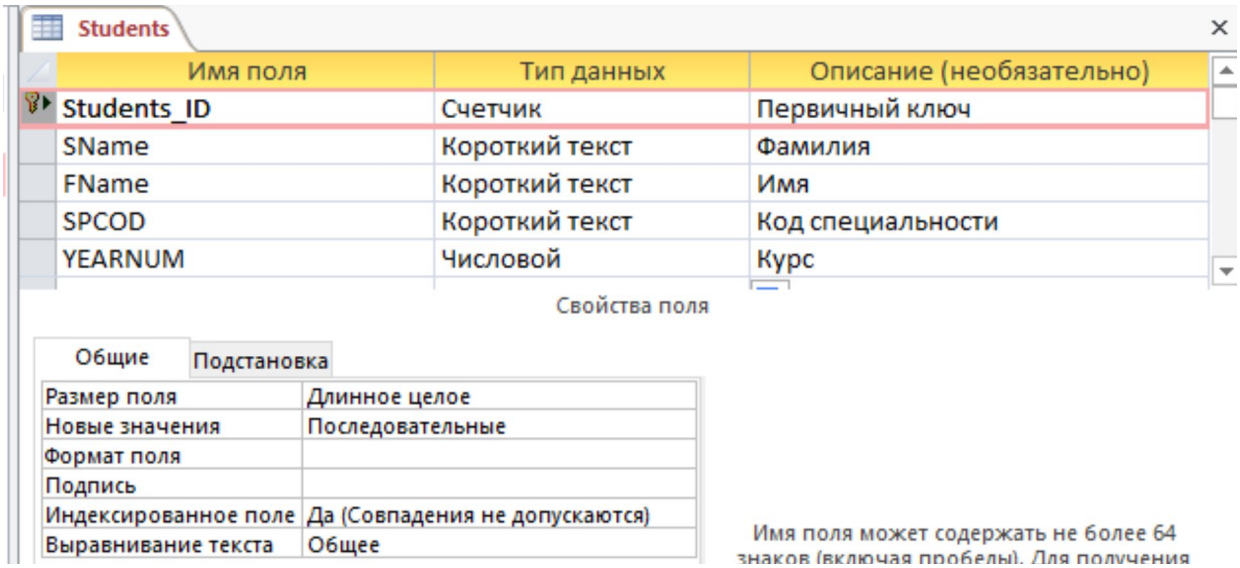



Рис. 3.2. Добавление первичного поля в таблицу

> При выборе названия поля, которое станет первичным ключом, используйте название таблицы и аббревиатуру “ID” (от англ. identification). Например: Students_ID.

Внеся необходимые дополнения в таблицу, сохраните ее. Вставьте в таблицу новую запись и посмотрите на поведение поля “Students_ID” – поле автоматически получило новое значение (предыдущее значение + 1). Таким образом, СУБД гарантирует, что каждая новая строка отношения получит очередной **уникальный идентификатор**. Даже если мы удалим последнюю строку в таблице, значение старого первичного ключа больше никогда не будет задействовано.

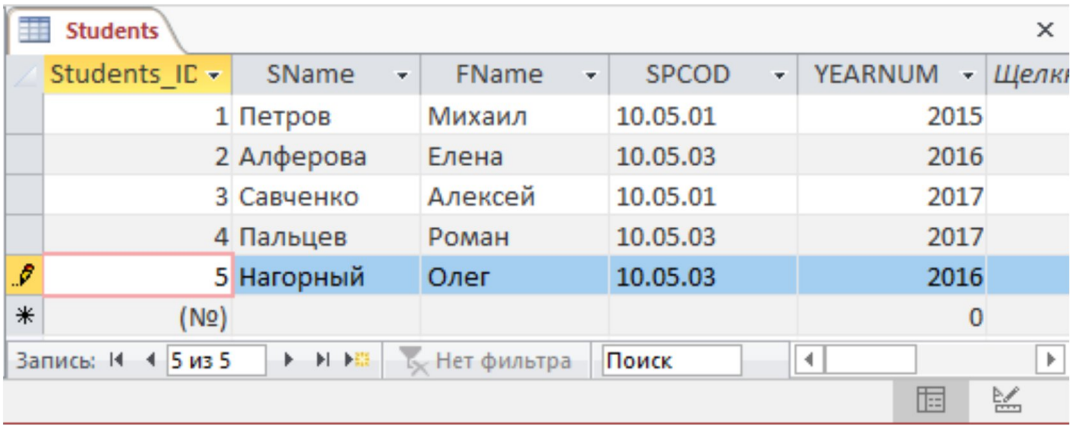



Рис. 3.3. Поведение поля-счетчика при добавлении новой записи

Рассказывая о ключах, мы пока не упомянули еще один важный вид ключа – внешний. О нем мы поговорим в следующем подразделе, в котором мы рассмотрим один из способов устранения избыточности в R-таблицах.

## Связь между таблицами

Внимательно изучите данные из таблицы Students. Не замечаете ли вы в ней очередного изъяна? Профессиональному разработчику БД достаточно одного взгляда на поле “SPCOD” чтобы сделать вывод о том, что наша таблица весьма неуважительно относится к дисковому пространству компьютера. И действительно, столбец, содержащий информацию о специальности, хранит многократно повторяющиеся значения: строка 10.05.01 (код специальности “Компьютерная безопасность”) встречается дважды, а строка 10.05.03 целых три раза. Наличие избыточных данных в таблице это один из верных признаков, что она не является реляционной.

Один из способов – разбиение таблицы с избыточными данными на две таблицы. В нашем случае за пределы таблицы Students выносится информация о специализации. Новая таблица будет содержать всего два поля:

1. Поле первичного ключа Spec_ID.
2. Поле названия специализации SpName.

Создайте эту таблицу и назовите ее “Spec” (рисунок 3.4).

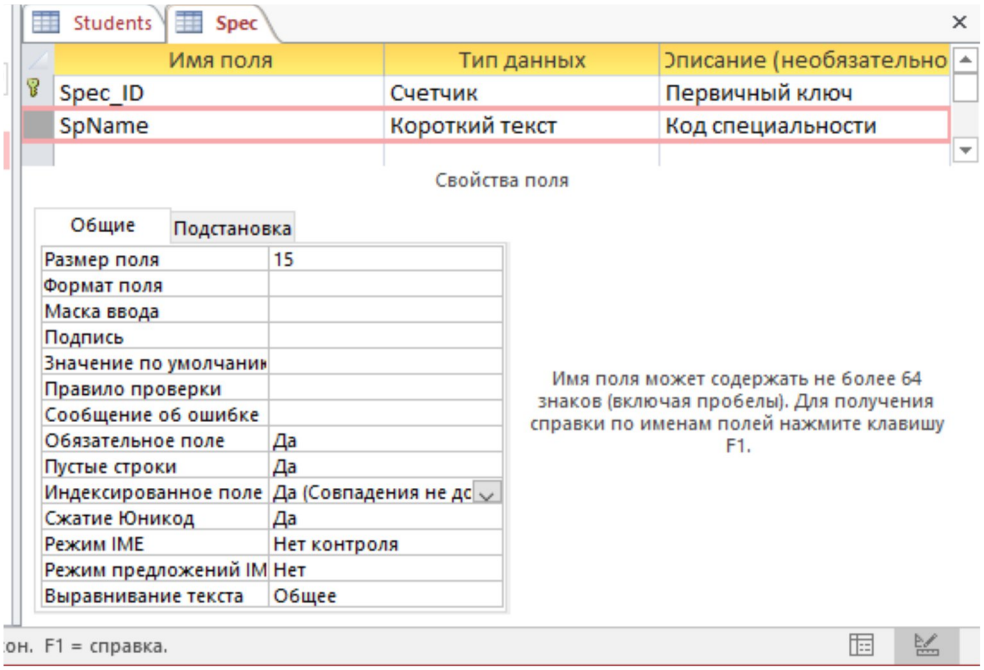



Рис. 3.4. Таблица специализации

Заполните вновь созданную таблицу названиями специальностей – “10.05.01” и “10.05.03”.

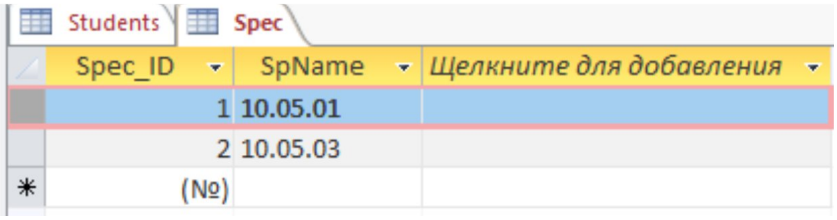



Рис. 3.5. Содержание таблицы специальностей

Наша таблица “Students” пока не способна воспринимать данные из вновь созданной справочной таблицы “Spec”. Для этого таблица студентов должна быть оснащена специальным полем, благодаря которому будет организовано взаимодействие с внешней таблицей. Такое поле называется **внешним ключом**, в него станут передаваться значения первичных ключей соответствующих специальностей.

> **Внешний ключ** – это столбец (или множество столбцов таблицы), которое соответствует первичному ключу некоторой (может быть, той же самой) таблицы.

Для создания внешнего ключа (рис. 3.6):

1. Откройте таблицу списка студентов в режиме конструктора.
2. Создайте числовое поле Spec_ID (размер поля “Длинное целое” – 4 байта). Это поле и будет внешним ключом.
3. Удалите из таблицы “Students” старое поле “SpCod”.

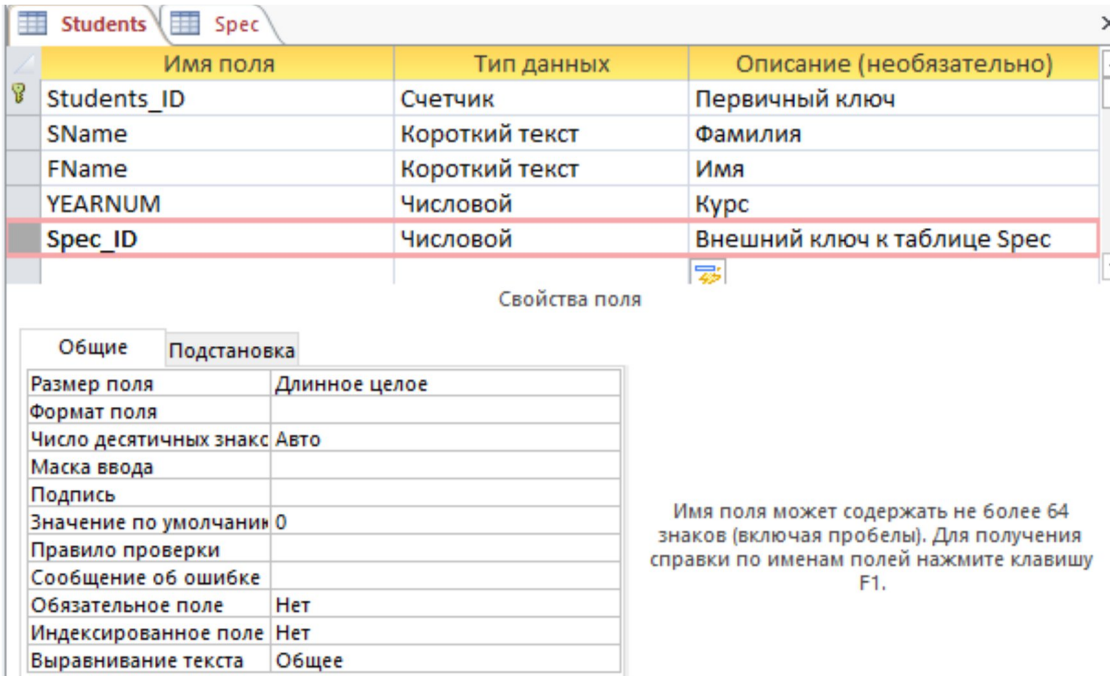



Рис. 3.6. Создание поля внешнего ключа в таблице “Students”

Подготовив поле внешнего ключа, нам необходимо связать обе таблицы. В Access этого результата можно достичь, воспользовавшись сервисной возможностью среды – схемой данных.

## Обеспечение целостности данных

При разработке таблиц и организации связи между таблицами разработчик БД обязан определить правила целостности данных. Различают два основных правила поддержания целостности, называемые: **целостностью сущностей** и **ссылочной целостностью**. Первое и второе правила имеют прямое отношения к порядку назначения первичного и внешних ключей.

> ***Определение***
> **Правило целостности сущностей**. В таблице ни один столбец, входящий в состав первичного ключа не может содержать неопределенных (NULL) значений.

Заложенный в правиле смысл абсолютно прозрачен. Первичный ключ — это минимальный идентификатор, который используется для уникального определения строк в таблицах. Допуская, что, хотя бы одно поле

составного ключа содержит NULL, мы разрушаем саму логику построения составного ключа – получается, что не все поля ключа необходимы для уникальной идентификации записи.

> ***Определение***
> **Правило ссылочной целостности**. Значение внешнего ключа таблицы должно либо соответствовать значению первичного ключа некоторой строки в базовой таблице, либо задаваться определителем NULL.

В соответствии с правилом ссылочной целостности считается допустимым создание записи с информацией о новом студенте с указанием определителя NULL вместо номера группы, в котором этот студент учится. Такая ситуация может иметь место в том случае, когда студент зачислен в институт, но еще не приписан к какому-то конкретному отделению.

## Построение схемы данных средствами Microsoft Access

Вызовите схему данных (пункт меню “Сервис – Схема данных”) и добавьте в ее обе наши таблицы. Удерживая в нажатом состоянии левую кнопку мышки, перетащите поле “Spec_ID” из таблицы “Spec” в таблицу “Students”. В появившемся окне “Изменение связей” включите опцию “Обеспечение целостности данных”. Связь с включенными условиями целостности данных обеспечивает соответствие каждого значения, которое вводится в столбец внешнего ключа, существующему значению в связанном столбце первичного ключа. В результате наших действий между таблицами будет организована связь “один-ко-многим” (рис. 3.7).

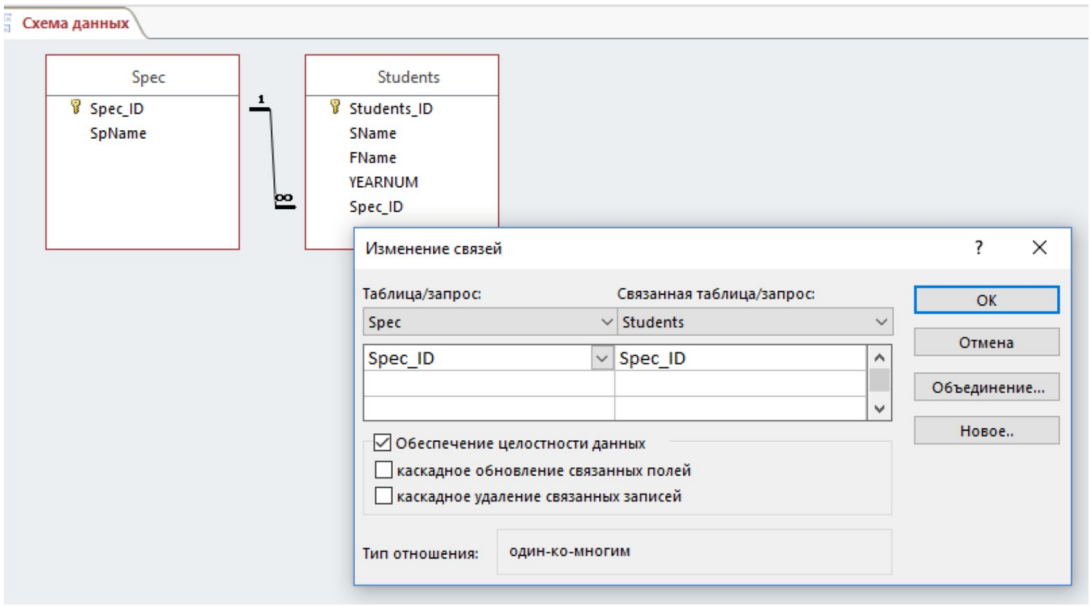



Рис. 3.7. Схема данных БД

В нашем случае связь “один-ко-многим” означает, что одна специальность из таблицы “Spec” может соответствовать многим студентам. Рассмотрим рисунок 3.8. Теперь вместо ввода названия специальности в поле внешнего ключа списка студентов достаточно передать его ключ. Значение 1 соответствует специальности “10.05.01”, 2 – “10.05.03”.

**Таблица Students**

| Students_ID | SName | FName | YearNum | Spec_ID |
|---|---|---|---|---|
| 1 | Петров | Михаил | 1 | 1 |
| 2 | Алфёрова | Елена | 2 | 2 |
| 3 | Савченко | Алексей | 4 | 1 |
| 4 | Пальцев | Роман | 4 | 2 |
| 5 | Нагорный | Олег | 2 | 2 |

**Таблица Spec**

| Spec_ID | SpName |
|---|---|
| 1 | 10.05.01 |
| 2 | 10.05.03 |

Рис. 3.8. Таблицы с данными

## Мастер подстановок

Вынеся повторяющиеся данные во внешнюю таблицу, мы избавились от избыточности, но утратили наглядность представления данных. Раньше пользователю было достаточно одного взгляда на таблицу студентов, чтобы узнать по какой специальности учится тот или иной человек. Теперь же увидев в столбце Spec_ID студента Петрова значение 1, необходимо обратиться к справочнику специальностей, чтобы выяснить, что единица соответствует компьютерной безопасности.

Попробуем сделать список студентов более приветливым. Для этого откройте таблицу студентов в режиме конструктора. выберите поле внешнего ключа Spec_ID. В нижней части окна откройте страничку “Подстановка”. На этой странице расположена всего одна строка “Тип элемента управления” – укажите в ней, что мы намерены работать с полем со списком. Сразу после этого действия на странице отобразятся дополнительные свойства поля (рис. 3.9).

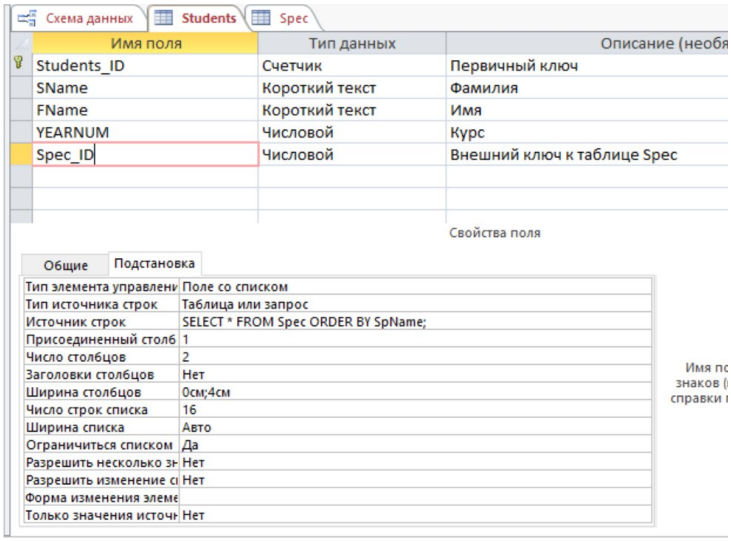



Рис. 3.9. Настройка поля Spec_ID таблицы студентов

В элементе “Источник строк” введите инструкцию на структурированном языке запросов.

```
SELECT * FROM Spec ORDER BY SpName;
```

Эта команда укажет, на то, что в качестве источника данных будет выступать весь перечень специальностей из таблицы, в дополнение ко всему список упорядочится по алфавиту.

Осталось внести два изменения. Укажите, что присоединяются два столбца (свойство “Число столбцов”) и размеры этих столбцов 0 и 4 сантиметра (свойство ширина столбцов). Закройте конструктор, не забыв сохранить изменения. Откройте таблицу Students в режиме редактирования данных. Благодаря мастеру подстановок вместо невыразительных значений ключей поле Spec_ID научилось отображать истинные названия специальностей (рис. 3.10).

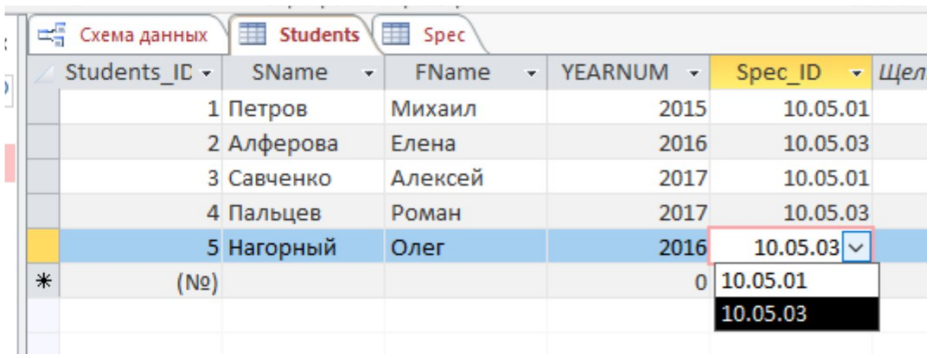



Рис. 3.10. Поведение поля с подстановкой

## Задание

Проанализируйте таблицу, созданную вами на лабораторной работе № 2. Определите ключевые поля, выявите и устраните повторяющиеся данные путем разбиения исходной таблицы на 2 (и более).

Изучите какие еще варианты полей позволяет создать Access.

# 4. Модель “сущность-связь” (ER-модель)

*Вид занятия – практическое занятие*
*Время занятия – 2 часа*
*Программное обеспечение – Microsoft Visio*

Проблема упрощения процесса проектирования базы данных, так чтобы, с одной стороны, модель была понятна и неспециалисту, а с другой – вполне устраивала и подготовленного программиста, не нова. Работа над этой проблемой вызвала к жизни целое семейство семантических моделей данных. На сегодняшний момент времени существует несколько эффективных решений, мы познакомимся с ключевым из них – высокоуровневой концептуальной моделью, разработанной Питером Ченом (Peter Pin-Shan Chen). Указанная модель известна под названием «сущность-связь» (Entity-Relationship), или просто ER-модель. Впервые она была анонсирована в марте 1976-го, тогда П. Чен опубликовал работу «The Entity-Relationship Model. Toward a Unified View of Data» в научном журнале «Transactions on Database Systems (TODS)».

***Замечание***
Модель «сущность-связь» относится к разряду концептуальных. Другими словами, она представляет общий взгляд на данные и позволяет нам на понятийном уровне разобраться с тем, что будет представлено в будущей базе данных. Ответ на вопрос, как данные должны быть реализованы на практике, надо искать у логической и физической моделей.

## Типы сущностей и атрибуты

Построение ER-модели начинается с выявления всех типов сущностей, подлежащих хранению в будущей базе данных. Различают две разновидности типов сущностей: слабую и сильную. Слабый тип находится в зависимом состоянии от сильной сущности, напротив, сильный тип вполне самостоятелен и ни от кого (или чего) не зависит. На диаграммах сильный тип изображается в виде прямоугольника с названием типа сущности внутри него, для обозначения слабого типа сущности контур прямоугольника рисуется двойной линией (рис. 4.1).

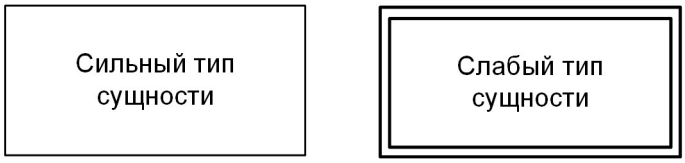



**Рис. 4.1.** Обозначение слабого и сильного типов сущности

Каждый тип сущности обладает некоторым набором атрибутов, в которых хранятся значения, описывающие конкретную сущность. Различают простые, составные, однозначные, многозначные и производные атрибуты. Главное отличие между простым и составным атрибутами в том, что простой состоит из одного компонента, а составной из нескольких. Примером составного атрибута может стать почтовый адрес, он включает группу простых атрибутов (индекс, страна, город, улица, дом). Однозначный атрибут содержит одно-единственное значение для сущности, например фамилия для типа сущности «Сотрудник». Многозначный атрибут допускает одновременное хранение нескольких значений, например у сотрудника может быть несколько телефонных номеров. Производный атрибут содержит значение, полученное на основе данных, хранящихся в других атрибутах. Например, возраст сотрудника можно легко вычислить, зная день его рождения и сегодняшнее число. Графическое представление атрибутов на ER-диаграмме вы найдете на рис. 4.2.

***Замечание***
В реляционных таблицах производные атрибуты редко когда превращаются в реальные столбцы таблицы, физически хранящие значения. Ведь нет смысла занимать драгоценную память под данные, которые можно получить путем элементарных вычислений.

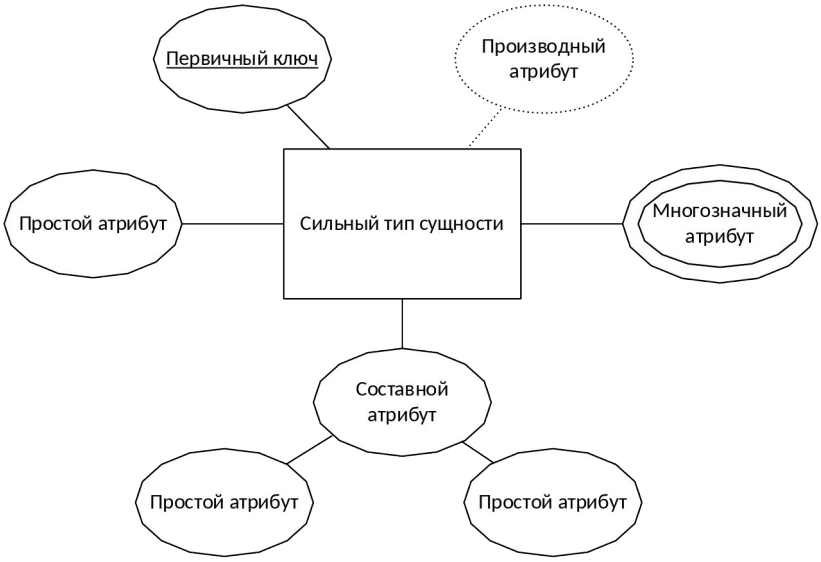



**Рис. 4.2.** Представление атрибутов на диаграммах ER-модели

Для изображения на схеме атрибута применяется эллипс, к своему типу сущности он присоединяется линией. Название атрибута записывается внутри эллипса. Если атрибут содержит идентификатор сущности (позднее он превратится в первичный ключ отношения), то название атрибута подчеркивается. Производный атрибут обводится пунктирной линией, многозначный – двойной.

> ***Замечание***
> При рисовании многозначного атрибута некоторые разработчики вместо черчения эллипса с двойным контуром рисуют обычный эллипс, но соединяют его с сущностью двойной линией. То же самое можно сказать и о производном атрибуте, только на этот раз эллипс соединяют с прямоугольником пунктирной линией.

Рассмотрим представленный на рис. 4.3 фрагмент ER-модели. Это диаграмма «Employee», описывающая тип сущности «сотрудник предприятия». Как видите, на диаграмме мы сумели представить все имеющиеся в нашем распоряжении разновидности атрибутов. Атрибут «Employee_id» выполняет функции первичного ключа сущности. Имя «FName» и фамилия «LName» являются ингредиентами составного атрибута «EmployeeName». Кроме этого, у нас имеется еще один составной атрибут «Address», он хранит почтовый адрес служащего. Обведенный двойной линией многозначный атрибут «PhoneNum» говорит нам о том, что у реального человека может быть несколько контактных телефонных номеров. Производный атрибут «Age» отражает возраст сотрудника, на схеме этот атрибут соединен с датой рождения «Birthday». Такая подробность подскажет разработчику, что расчет возраста должен вестись от даты появления на свет работника предприятия. Согласитесь, что для понимания сути предложенной диаграммы не требуется специальной подготовки, поэтому, нарисовав подобную модель будущей таблицы, можно без сомнений идти к заказчику базы данных.

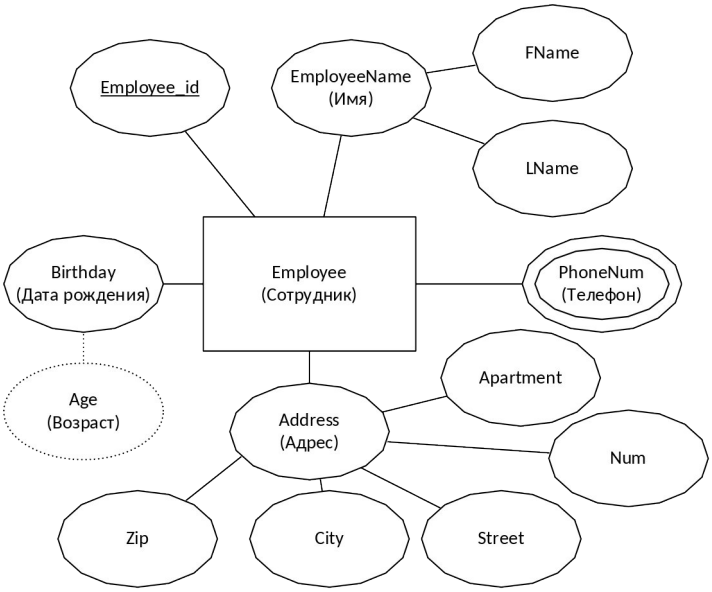



**Рис. 4.3.** Представление типа сущности на диаграммах ER-модели

## Связи в ER-модели

Сформировав полный перечень всех подлежащих учету типов сущностей и их атрибутов, создатель ER-модели переходит к очередному ответственному этапу. Теперь ему предстоит выявить все ассоциации между типами сущностей и на этой основе построить связи между ними. В предыдущей главе уже упоминалось, что явным признаком связи выступает глагол, который можно применить, характеризуя взаимоотношения между типами сущностей. Например, сотрудник **работает** в отделе, или самолет **выполняет** рейс.

Для того чтобы разработчик мог на схеме отразить смысловую нагрузку связи между сущностями, ее имя, а точнее глагол, показывающий характер взаимодействия между сущностями, записывается внутри ромба. С целью упрощения диаграммы допускается опускать атрибуты типов сущностей, ограничиваясь только первичными ключами (рис. 4.4).

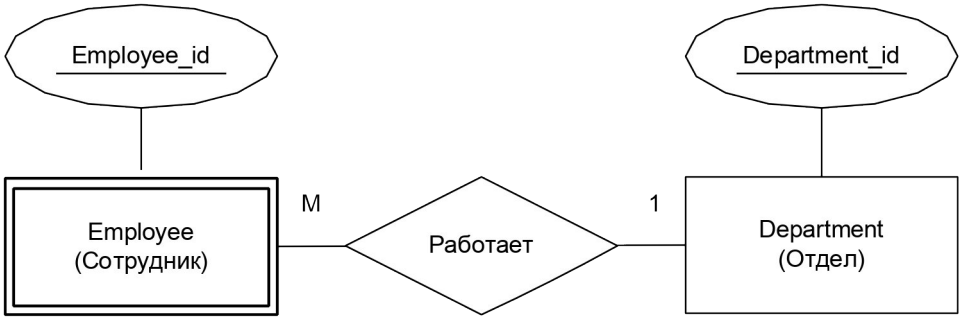



**Рис. 4.4.** Представление связи 1:M на диаграммах ER-модели

В ER-моделировании различают три типа связи между типами сущностей: «один к одному» (1:1), «один ко многим» (1:M) и «многие ко многим» (M:N). При появлении на диаграмме связи типа 1:1 следует задуматься, не является ли предполагаемый тип сущности всего лишь атрибутом. Исключение составляет обсужденный страницей ранее механизм супертипов и подтипов.

На рис. 4.4 представлена наиболее часто встречающаяся связь «один ко многим» – в одном отделе работает много сотрудников. На диаграмме отдел «Department» представлен как сильная сущность, сотрудник «Employee» – как слабая. Такое решение объясняется тем, что сотрудник находится в подчиненном отношении к отделу, в котором он трудится. Часто программисты для повышения информативности диаграмм при обозначении связи вместо использования символа «M» явным образом указывают мощность связи. Например, обозначение (1, 10) говорит о том, что минимальное значение мощности соответствует 1, а максимальное – 10 (другими словами, в отделах может работать от 1 до 10 сотрудников). Такие пояснения могут оказаться весьма полезными при практической разработке программного обеспечения, так как более подробно отражают бизнес-правила автоматизируемого учреждения или предприятия.

Проектируя ER-модель, нам следует особое внимание уделять связи «многие ко многим». Вполне реальна ситуация, когда несколько сотрудников одновременно выполняют несколько производственных поручений в заказе «Order» (рис. 4.6). Например, один и тот же автомеханик в рабочий день может получить заявки на обслуживание нескольких автомобилей, при этом отдельно взятый автомобиль может попасть в руки нескольких специалистов (допустим, на ремонт двигателя и на замену электропроводки).

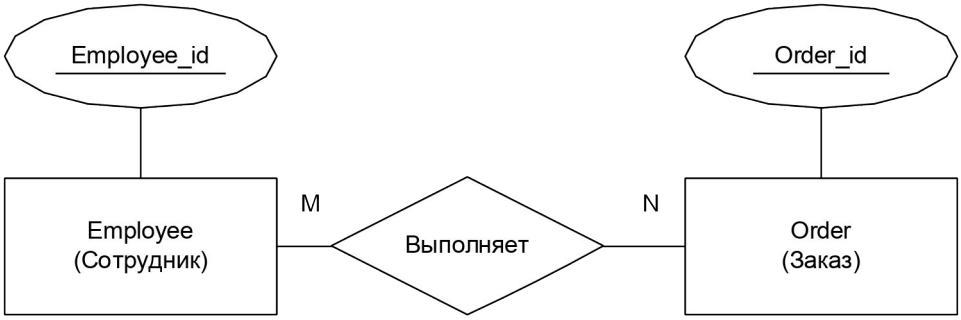



**Рис. 4.5.** Представление связи M:N на диаграммах ER-модели

Хотя связь M:N легко изобразить на диаграмме, ее далеко не просто реализовать физически, ведь реляционные базы поддерживают только отношение «один ко многим». Безусловно, рассматриваемая ситуация не безвыходная. Позднее, обсуждая процесс нормализации базы данных, мы очень подробно рассмотрим способ решения подобных задач на физическом уровне. А сейчас, пока мы находимся на концептуальном уровне моделирования, лишь запомним, что в тех случаях, когда между двумя типами

сущностей возникает связь «многие ко многим», разработчик БД создает искусственный тип сущности, выполняющий функции коммутатора между основными сущностями (рис. 4.7).

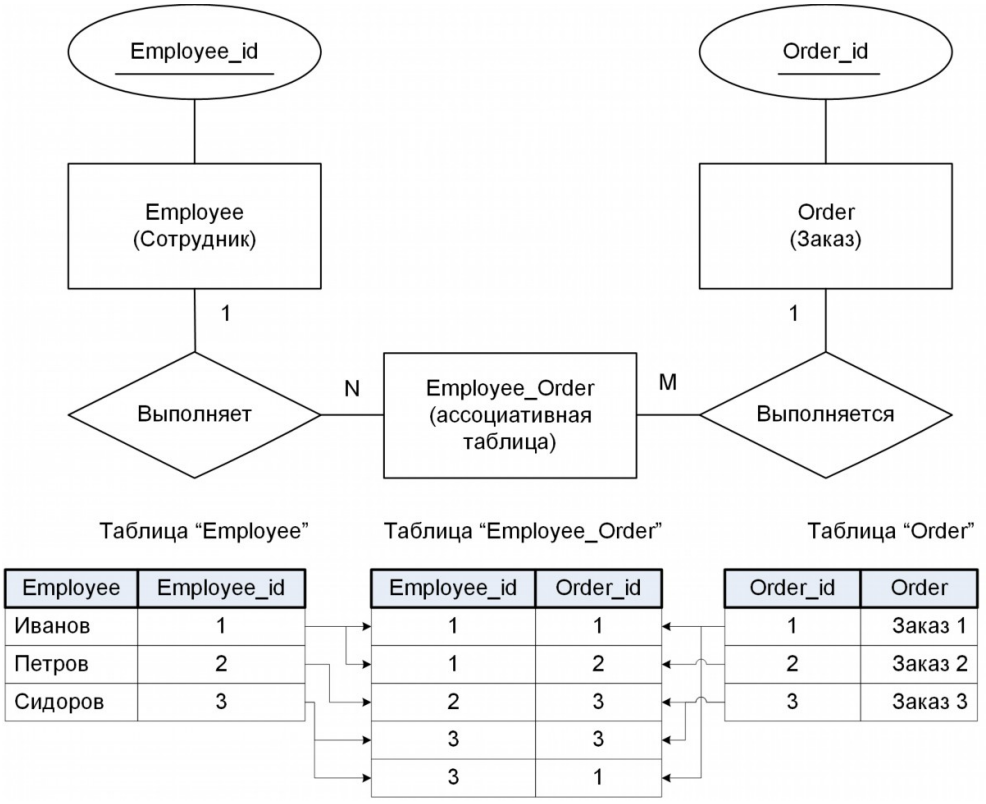



**Рис. 4.6.** Преобразование отношения M:N к двум отношениям 1:M

Дополнительный тип сущности разрывает связь M:N пополам, что позволяет нам трансформировать неподъемное для реляционных баз данных отношение «многие ко многим» в пару обычных связей «один ко многим». Вне зависимости от всех хитросплетений нашей ER-модели искусственно созданная сущность-коммутатор в любом случае будет содержать два атрибута для внешних ключей. Они предназначены для поддержания ассоциации между объединяемыми отношением «многие ко многим» типами сущностей, один атрибут коммутатора станет держать связь с расположенным на диаграмме слева типом «Employee», а другой – с правым типом сущности «Order». На диаграммах совсем необязательно опускаться до степени детализации рис. 4.7. Это может оказаться излишним с точки зрения наглядности схемы, поэтому большинство разработчиков ER-модели предпочитает компромиссное решение, оно представлено на рис. 4.8. Суть компромисса в том, что на диаграммах искусственный тип сущности изображается в виде ромба, вписанного в прямоугольник. Такое графическое решение, с одной стороны, указывает на то, что это будущая таблица, а с другой – напоминает, что своим появлением на свет таблица обязана необходимости организовать коммутацию между другими таблицами.

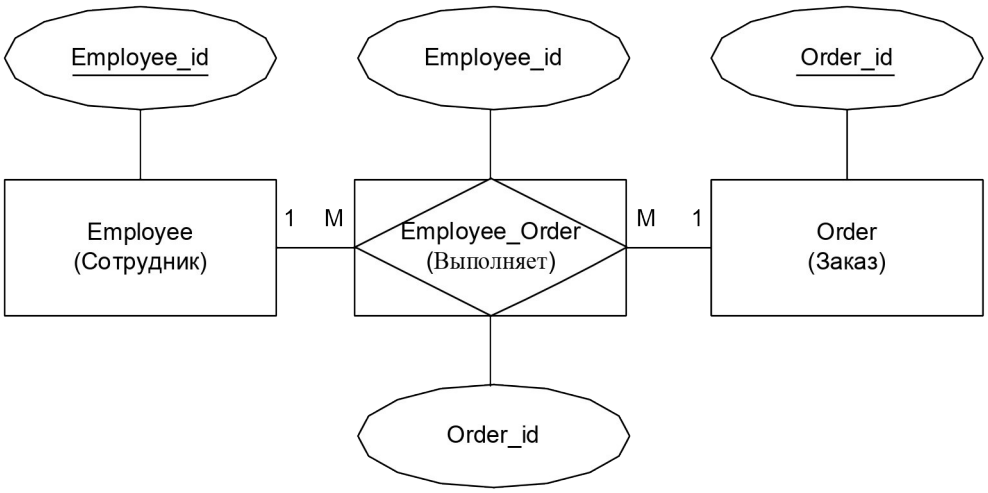



**Рис. 4.7.** Коммутирующий тип сущности на ER-диаграмме

## Сильные и слабые связи

Рассмотрим еще одну особенность взаимодействия между типами сущностей в базе данных – наличие сильных и слабых связей. Сильная связь обычно возникает между сильным и слабым типами сущностей в тех условиях, когда существование слабого типа невозможно без поддержки сильного. Например, слабая сущность «самолет» принадлежит сильной сущности «авиакомпания», или слабая сущность сотрудник не может трудиться на предприятии, не входя в штат какого-либо из отделов. Напротив, в той ситуации, когда присутствие связи между двумя типами сущности не обязательно, мы говорим о слабой связи. Слабые связи надо искать между типами сущностей, не находящимися в прямой зависимости друг от друга и, как следствие, способными жить автономно. Например, работа над некоторыми заказами может осуществляться не только в стенах нашего предприятия, но и совместно с каким-то соисполнителем (смежной фирмой). Несмотря на то что тип сущности «Accomplice» (соисполнитель) относится к разряду сильных, так как может существовать самостоятельно, связь между заказом и смежником оказывается слабой. Ведь нам не всегда нужны помощники, и большинство заказов мы выполняем только своими силами. Об этом нас информирует маленький кружок со стороны необязательной сущности (рис. 4.9).

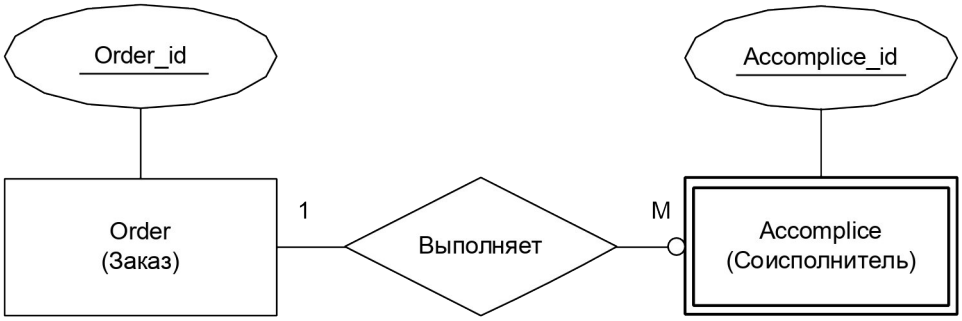



**Рис. 4.8.** Пример необязательной связи

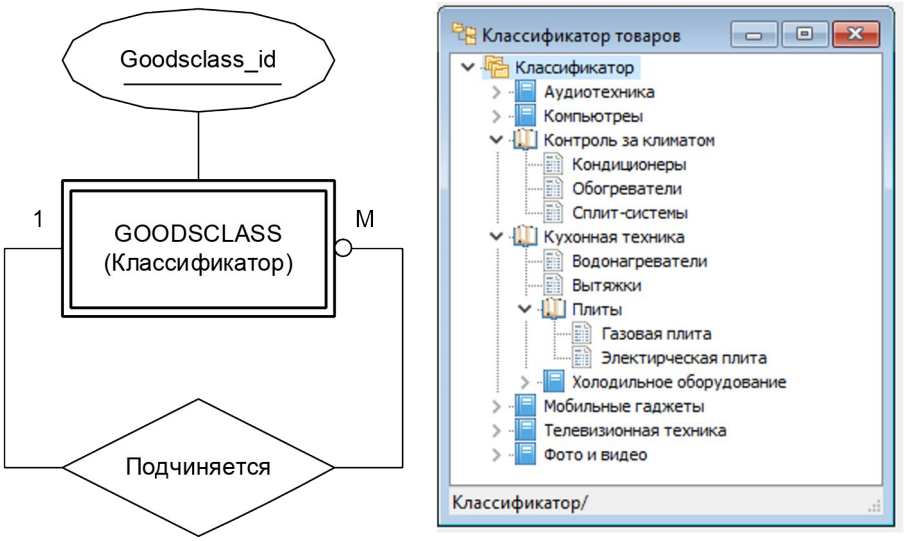
В последующем, на этапе создания реляционной таблицы «Order», кружок подскажет нам о том, что информация о соисполнителе заказа необязательна и поэтому поле внешнего ключа допускает вставку определителя NULL.

> ***Замечание***
> В некоторых нотациях ER-модели для обозначения сильной связи на диаграммах очерчивают ромб двойной линией.

## Рекурсивная связь

До сих пор на наших диаграммах мы сталкивались с наиболее часто возникающими в реальной жизни бинарными связями. Теперь поговорим об унарных связях, которые весьма эффективны при организации рекурсивных отношений внутри одной таблицы. Замыкание типа сущности на самого себя окажется весьма полезным при описании многоуровневых иерархических структур, подобных классификатору товаров, хранящихся на складе. Взгляните на рис. 4.10, на котором представлено решение такой задачи. Здесь тип сущности «Goodsclass» содержит названия категорий товаров, которыми торгует автоматизируемая фирма. Между категориями явно просматривается правило взаимодействия главный–подчиненный.

На практике, для построения иерархии, реальной реляционной таблице потребуется как минимум три столбца: поле первичного ключа «Goodsclass_id», поле внешнего ключа «Parent_id» и содержащее названия категорий текстовое поле «Goodsclass». Рекурсивная связь между записями таблицы обеспечивается за счет взаимодействия внешнего и первичного ключей, с этой целью поле внешнего ключа дочернего элемента хранит значение первичного ключа родительского узла. Если же идет речь о самом старшем элементе иерархии, который никому не подчинен (в нашем примере во главе дерева расположена папка «Классификатор»), то в его внешнем ключе окажется определитель NULL. Подтверждение моих слов вы обнаружите во фрагменте таблицы «Goodsclass», представленной в нижней части рис. 4.10. Например, узлу «Контроль за климатом» принадлежат дочерние узлы «Кондиционеры», «Обогреватели» и «Сплит-системы». В полях внешнего ключа всех трех подчиненных узлов вы найдете число 4, а это и есть значение первичного ключа родительского узла «Контроль за климатом».




Фрагмент данных таблицы “GOODSCLASS”

| Goodsclass_id | Parent_id | Goodsclass |
|---|---|---|
| 1 | NULL | Классификатор |
| 2 | 1 | Аудиотехника |
| 3 | 1 | Компьютеры |
| 4 | 1 | Контроль за климатом |
| 5 | 4 | Кондиционеры |
| 6 | 4 | Обогреватели |
| 7 | 4 | Сплит-системы |

**Рис. 4.9.** Представление унарной рекурсивной связи на диаграмме ER-модели

## Вариации ER-моделей

Модель Питера Чена вполне заслуживает лавры первого простого и одновременно эффективного инструмента моделирования БД. Для ее понимания вовсе не требуется глубоких знаний в области программирования и СУБД. Достаточно способности логически мыслить и немного терпения с крупицей интуиции. Благодаря своей успешности ER-модель Чена не только сразу снискала признательность у обычных программистов, но и вдохновила ряд специалистов на создание ряда схожих по идеологии инструментов проектирования БД.

Один из наиболее популярных нотаций ER-модели принадлежит уже знакомому нам по сетевой модели данных Чарльзу Бэчмэну. Бэчман несколько видоизменил графические обозначения связей на диаграммах сущность-связь. Связи в нотации Бэчмана также чертятся в виде линии, соединяющей типы сущностей, но на ней отсутствует ромб с названием связи. Вертикальная черта на конце линии связи означает, что эта сторона представляет отношение «один». На стороне «многие» линия связи расщепляется на три луча. Некоторая схожесть связи на стороне «M» с трехпалой птичьей лапкой привела к тому, что за ER-моделью Бэчмана прочно закрепилось название «Crow’s Foot model» (модель «воронья лапка») (рис. 4.12).

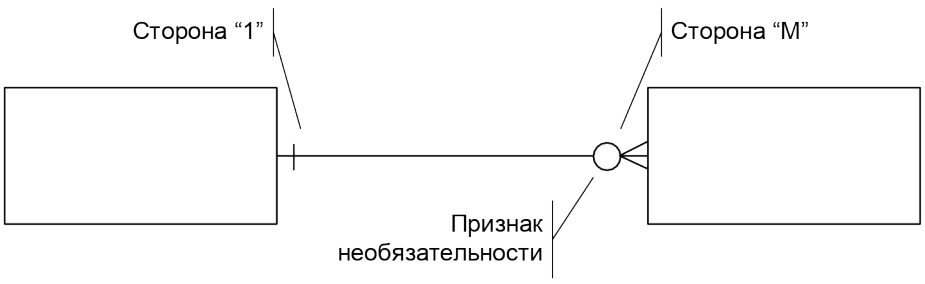



**Рис. 4.10.** Представление связи на модели «Crow’s foot»

На диаграммах «воронья лапка» атрибуты сущности рисуются не в виде отдельных эллипсов, а записываются в виде списка, под названием сущности. Благодаря чему модель приобрела более компактную форму представления.

Еще одна, заслуживающая внимания версия ER-моделей получилась в результате работы американской фирмы Hughes Aircraft. Создатели модели воспользовались наработками в области систем автоматизированного производства (integrated computer-aided manufacturing, ICAM), широко проводимых в США в 1970-х годах, и реализовали свою модель ICAM Definition, сокращенно IDEF. Несколько позднее модель IDEF была доработана и сегодня более широко известна под именем IDEF1X. IDEF1X впитала лучшие стороны моделей Чена и Бэчмэна.

Различными разработчиками создан внушительный перечень программного обеспечения, позволяющего автоматизировать этап концептуального моделирования БД. Эти продукты объединяют под понятием CASE-системы. Термин CASE (Computer-Aided Software Engineering) можно перевести как автоматизированное проектирование и создание программ. В качестве примеров CASE продуктов стоит упомянуть: ER/Studio (корпорации Embarcadero), ERwin Data Modeler (фирмы Platinum – CA), PowerDesigner (фирмы Sybase), Microsoft Visio. На рис. 4.13 представлен экранный снимок процесса построения ER-модели в редакторе MySQL Workbench. Здесь вы обнаружите диаграмму взаимодействия между сущностями БД «Склад», с которой мы еще не раз встретимся на страницах книги. Как видите, корни современных CASE-систем, специализирующихся на проектировании БД, также уходят в модель Питера Чена.

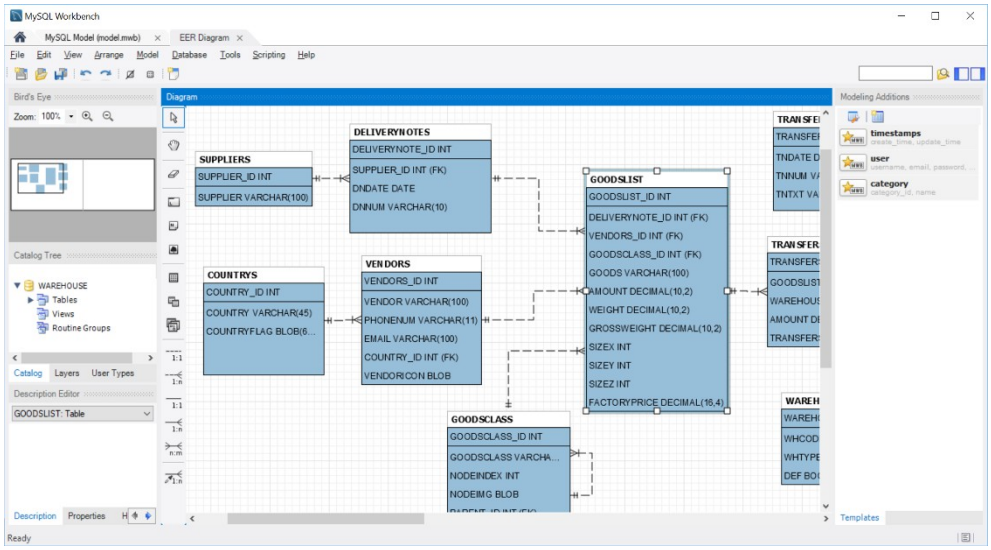



**Рис. 4.11.** Построение ER-модели в MySQL Workbench

Основная цель CASE-проектирования заключается не столько в автоматизации хода разработки БД, сколько в превращении трудоемкого и малопонятного для непосвященных «ритуала» кодирования в относительно более простой процесс логического проектирования. Разработчику зачастую даже не надо знать языки программирования, достаточно владеть методологией концептуального проектирования БД и уметь пользоваться мышкой... Как следствие CASE-средством сможет воспользоваться даже начинающий пользователь, что не только ускоряет, но и значительно удешевляет стоимость получаемого программного обеспечения. Ведь зачастую достаточно щелкнуть кнопкой, и из нарисованной концептуальной модели создается физическая БД! Но, с другой стороны, качество результирующего продукта выходящего из-под «пера» CASE-инструмента, и качество «ручной работы» профессионального программиста сегодня сравнивать не стоит.

## Задание

В соответствии с полученными вариантами разработайте ER-модель для будущей базы данных:

1. База данных поликлиники. Учету подлежат: больные (ФИО, домашний адрес, место работы), лечащий персонал (ФИО, специальность), дата посещения больным врача и поставленный диагноз, даты больничного.
2. База данных футбольных матчей. Учету подлежат: спортивный клуб, игроки команд (ФИО), дата матча, игравшие команды, забитые мячи, авторы голов.
3. База данных “Ремонт компьютеров”. Учету подлежат: клиенты (ФИО, домашний адрес), дата сдачи ПК в ремонт, неисправность, элементы и узлы, использованные для ремонта, стоимость элементов и узлов, стоимость работ, дата возврата ПК клиенту.

4. База данных “Бюро по туризму”. Учету подлежат: клиенты фирмы (ФИО, домашний адрес), список курортов с ценами и продолжительностью путевок, даты поездок клиентов на курорт.
5. База данных “Магазин видеофильмов”. Учету подлежат: название фильма, жанр, год выхода, киностудия, режиссер, цена, носитель (DVD/Видеокассета), число экземпляров.
6. База данных “Домашняя фонотека”. Учету подлежат: исполнитель, страна к которой принадлежит исполнитель, название альбома, год выхода альбома, музыкальный жанр альбома, наименование и продолжительность музыкальных композиций, входящих в альбом, формат хранения (CD, mp3, …).
7. База данных “Склад автомобильных запчастей”. Учету подлежат: название детали, количество деталей на складе, производитель, цена за единицу, в какой марке автомобиля применяется, поставщик детали, дата поставки.
8. База данных “Метеостанция”. Учету подлежат: названия станций, их географические координаты, персонал станций (ФИО, должность), каждая метеостанция ежечасно сохраняет данные о температуре окружающей среды.
9. База данных “Библиотека”. Учету подлежат: жанр книги, авторы книги, название книги, количество экземпляров, место хранения книги, читатель (ФИО и адрес), дата получения читателем, дата возврата книги.
10. База данных “Склад телевизоров”. Учету подлежат: поставщик, производитель телевизора, модель телевизора, дата поставки телевизора, дата списания телевизора со склада, не менее 5 характеристик телевизора (производитель, диагональ, вес, и т.п.), цена телевизора от поставщика, отпускная цена телевизора, количество телевизоров.
11. База данных “Автовокзал.” Учету подлежат: автобусы (марка, дата выпуска и государственный регистрационный номер), водители (ФИО, дата рождения, водительская категория, какой автобус закреплен), расписание движения автобусов (наименование населенного пункта, день и время отправления, день и время прибытия, стоимость билета, автобус выполняющий рейс).
12. База данных “Магазин одежды.” Учету подлежат: товар (название, размер, цена, материал, производитель), дата поступления, дата продажи.
13. База данных “Почтовое отделение” Учету подлежат: поступающая на почту заказная корреспонденция (письма, ценные бандероли и посылки). Дата поступления, дата выдачи, паспортные данные получателя корреспонденции.
14. База данных “Аптечный киоск” Учету подлежат: лекарства (наименование, дозировка, производитель, отпускная цена, количество упаковок, место хранения, выдача только по рецепту или нет, показания и противопоказания).
15. База базы данных “Расписание занятий в институте” Учету подлежат: специальность, учебная группа, учебные дисциплины (название и вид занятия), день недели, номера часов проведения занятий, преподаватель, номер аудитории
16. База данных “Гостиница” Учету подлежат: номера гостиницы (номер помещения, число спальных мест, наличие душа, туалета, холодильника, телевизора, сплит-системы, стоимость проживания), занятость номера (дата заселения, паспортные данные жильцов, дата освобождения номера)
17. База данных “Кинотеатр”. Учету подлежат: название кинофильма, жанр кинофильма, возрастные ограничения, дата и время киносеанса, название кинозала, номер посадочного места, цена билета.
18. База данных “Торговый зал”. Учету подлежат: название товара, отпускная цена товара, количество товара в зале, дата выпуска, срок годности. База данных должна предусматривать отпуск товара покупателю со списанием товара из зала.
19. База данных “Учет электроэнергии”. База учитывает количество потребленной электроэнергии в небольшом поселке и ежемесячно подготавливает индивидуальные квитанции об оплате. Учету подлежат: адрес дома, владелец дома, количество потребленной электроэнергии за месяц, стоимость за кВт/ч. Стоимость электроэнергии может изменяться!
20. База данных “Отдел кадров завода”. Учету подлежат: ФИО, рабочая специальность, квалификация (разряд), образование работника, дата трудоустройства, стаж работы, должностной оклад, надбавка (зависит от квалификации и стажа).
21. База данных “Художественный музей” Учету подлежат: автор картины, название картины, краткое описание сюжета, дата написания картины, художественное направление (реализм, абстракция, и т.д.), зал в котором выставлена картина.

**Примечания**

- Выполненное задание сдается в письменном (распечатанном) виде и заверяется подписью преподавателя.
- При построении ER-модели рекомендуется воспользоваться услугами специализированных редакторов. Например, ER/Studio входящего в поставку Embarcadero RAD Studio или Microsoft Visio.

# 5. Логическое проектирование и нормализация

*Вид занятия – лабораторное занятие*
*Время занятия – 2 часа*
*Программное обеспечение – среда Microsoft Access*

**Нормализация** преследует несколько целей. Наиболее важная из них – борьба с избыточностью данных. Устранение избыточности достигается не просто за счет рационального назначения размерности атрибутам таблиц, а за счет формирования эффективных взаимоувязанных табличных структур, объединяющих несколько отношений. Так что процесс нормализации выступает в роли основного элемента процесса проектирования БД.

> ***Замечание***
> Процесс нормализации и ER-моделирование выступают разными сторонами одной медали. Оба подхода приводят к созданию модели БД, с той лишь разницей, что ER-моделирование осуществляется на концептуальном уровне и основано на принципе от общего к частному (так называемый нисходящий подход). Нормализация производится на этапе логического проектирования и, в отличие от ER-моделирования, исповедует идеи восходящего подхода (от частного к общему).

Победа в борьбе за лишние байты не только приводит к минимизации расходов на хранение данных, но и устраняет ряд косвенных проблем, точнее аномалий, присущих неграмотно построенным таблицам. К таковым относятся **аномалии** вставки, редактирования и удаления данных.

## Первая нормальная форма

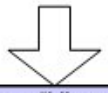
Самое худшее, что можно сделать с реляционной таблицей, – это разместить в ней **повторяющиеся группы данных**. Повторяющаяся группа появляется каждый раз, когда в одну ячейку таблицы мы пытаемся вставить больше одного значения.

Например, достаточно велико искушение в текстовом столбце классификатора товаров (допустим, для строки с какой-нибудь микроволновой печью) написать следующее: «Бытовая техника; Кухонная техника; Мелкая бытовая техника; Печи микроволновые». Но это ошибка! Таблица не должна содержать таких данных – каждое поле таблицы должно быть неделимым.

> ***Замечание***
> Под атомарностью значения понимается, с одной стороны, его целостность, а с другой – неделимость. Разделение данных на «атомы» впоследствии позволит нам вооружить БД дополнительным сервисом, в частности разнообразными способами сортировки данных и расширенными методами фильтрации и поиска данных.

Что такое **неделимое поле** и как достичь неделимости на практике? Предположим, что мы работаем над базой данных домашней библиотеки (рис. 5.1). Взгляните на ненормализованный вариант таблицы Library. Она состоит из двух полей: Writer – автор книги и Books – название произведения. В этой таблице представлены сразу две ошибки, связанные с неделимостью поля. Наиболее грубая из них – в поле Books. Обратите внимание, что атрибут поля одновременно содержит названия двух и более произведений. Это неверно – для каждого произведения должна быть отведена отдельная строка таблицы. Вторая ошибка связана с полем Writer, где мы храним имя, отчество и фамилию писателя. В таких целях целесообразно сформировать три отдельных поля: имя, отчество и фамилия. Причин тому несколько. Представьте себе, что при вводе информации в колонку Write пользователь вместо «Лев Николаевич Толстой» ввел «Толстой Лев Николаевич». С точки зрения человеческой логики это абсолютно одинаковые значения, но компьютер воспримет их как разные, что сразу отразиться на работе программы. Вторая причина – разделив колонку Writer на три (по смыслу уже неделимых) колонки мы значительно упростим задачу по сортировке и выборке данных средствами SQL запросов. Например, теперь мы получим возможность отобрать записи авторов без отчества, и т.п. Заметьте, что приведенная к первой нормальной форме таблица приобрела явную избыточность, но на этом этапе это не страшно.

| Ненормализованная таблица “Library” | |
|---|---|
| **Writer** | **Books** |
| Лев Николаевич Толстой | Война и Мир<br>Анна Каренина |
| Александр Сергеевич Пушкин | Евгений Онегин<br>Сказка о царе Салтане |
| Александр Дюма | Три мушкетёра<br>Десят лет спустя<br>Граф Монте-Кристо |



| Таблица “Library” приведённая к 1NF | | | |
|---|---|---|---|
| **FName** | **LName** | **Name** | **Books** |
| Лев | Николаевич | Толстой | Война и мир |
| Лев | Николаевич | Толстой | Анна Каренина |
| Александр | Сергеевич | Пушкин | Евгений Онегин |
| Александр | Сергеевич | Пушкин | Сказка о царе Салтане |
| Александр | | Дюма | Три мушкетёра |
| Александр | | Дюма | Десять лет спустя |
| Александр | | Дюма | Граф Монте-Кристо |

Рис. 5.1. Создание неделимых полей

Теперь поговорим о повторяющейся группе. **Повторяющаяся группа** представляет собой поле, которое повторяется внутри строки, с целью хранения нескольких значений для атрибута. Допустим, что мы разрабатываем базу данных, предназначенную для хранения данных о температуре воздуха, причем измерения производятся ежедневно, каждый четный час. На рисунке 5.2 изображена классическая ошибка начинающего программиста. Для хранения даты в таблице «temperature» используется поле `TmprDate`, а для хранения температуры поле Tmpr. В одном и том же поле заносится несколько значений, разделенных точкой с запятой. Несостоятельность такого подхода при проектировании таблицы не вызывает и сомнения. Представьте себе, как бы выглядела таблица температур, если бы измерения проводились каждые четверть часа, или каждую минуту… Можете поверить, что даже если повторяющихся групп не много, то они все равно затруднят построение запросов и форматирование отчетов, поэтому посмотрим, как от них избавиться. Вернемся к рисунку 5.2. Для того, чтобы преобразовать таблицу температур к 1NF нам потребовалось создать таблицу, состоящую из трех столбцов – даты, времени и температуры. И здесь (как и в случае с приведением к 1NF таблицы авторов) мы получили некоторую избыточность, но зато все логично.

| Ненормализованная таблица “temperature” | |
|---|---|
| **TmprDate** | **Tmpr** |
| 01.01.2003 | -10.5; -10.5; -11.0; ... |
| 02.01.2003 | -12.0; -13,0; ... |

| “temperature” приведена к 1NF | | |
|---|---|---|
| **TmprDate** | **TTime** | **Tmpr** |
| 01.01.2003 | 00:00 | -10.5 |
| 01.01.2003 | 02:00 | -10.5 |
| 01.01.2003 | 04:00 | -11.0 |
| ... | ... | ... |
| 02.01.2003 | 20:00 | -12.0 |
| 02.01.2003 | 22:00 | -13.0 |

Рис. 5.2. Пример устранения повторяющейся группы

***Определение***

Таблица приведена к первой нормальной форме, если в ней отсутствуют повторяющиеся группы и все значения, хранимые в ней, неделимы (атомарны).

## Вторая нормальная форма

Этап приведения таблицы ко второй нормальной форме может начинаться только после того, как таблица была приведена к 1NF.

***Определение***

Таблица приведена ко второй нормальной форме, если она соответствует 1NF и в ней нет частичных зависимостей, т. е. нет неключевых столбцов, зависящих от части первичного ключа.

Другими словами, у таблицы 2NF все не ключевые поля полностью зависят от первичного ключа или (в случае составного первичного ключа) от каждого поля первичного ключа. Таким образом любое, хранящееся в не ключевом столбце таблицы, значение однозначно определяется значением первичного ключа.

| “Library” | | | | |
|---|---|---|---|---|
| **Library_ID** | **FName** | **LName** | **SurName** | **Books** |
| 1 | Лев | Николаевич | Толстой | Война и мир |
| 2 | Лев | Николаевич | Толстой | Анна Каренина |
| 3 | Александр | Сергеевич | Пушкин | Евгений Онегин |
| 4 | Александр | Сергеевич | Пушкин | Сказка о царе Салтане |
| 5 | Александр | | Дюма | Три мушкетёра |
| 6 | Александр | | Дюма | Десять лет спустя |
| 7 | Александр | | Дюма | Граф Монте-Кристо |

Рис. 5.3. Приведение таблицы к 2NF

Итак, на втором этапе приведения таблицы к реляционному виду необходимо создание первичного ключа. Рассмотрим очередной пример. А для этого предлагаю вернуться к таблице Library. Для преобразования этой таблицы ко второй нормальной форме просто дополняем ее столбцом – Library_ID (рис. 5.3). В качестве первичного ключа разработчики баз данных очень часто используют поля автоинкрементного типа. В этом случае при вставке новой записи, значение ее первичного ключа получает приращение на одну единицу, что гарантирует уникальность этого ключа.

## Третья нормальная форма

Визитной карточкой БД, таблицы которой не приведены к 3 нормальной формой, выступают транзитивные зависимости. Под **транзитивной зависимостью** понимается такая зависимость между атрибутами одного отношения, когда один из атрибутов связан с другим через атрибут-посредник A → B → C.

Например (рис. 5.4) таблица склада прячет в себе несколько транзитивных зависимостей. Очевидный пример транзитивности – «Поставщик». Указанный атрибут непосредственно связан лишь с атрибутами, описывающими накладную («Номер накладной» и «Дата накладной»), а все остальные атрибуты связаны с «Поставщик» транзитивно.

| **Товары на складе** | |
|---|---|
| PK | ПКлюч |
| | Поставщик<br>Номер накладной<br>Дата накладной<br>Производитель<br>Наименование<br>Количество<br>Цена за ед.<br>Склад<br>... |

**Рис. 5.4.** Таблица «Склад» в состоянии 2NF

Транзитивная зависимость A → B → C устраняется одним-единственным способом – за счет выделения атрибутов B → C (или A → B) в другую таблицу. Связи между такими таблицами строятся по ключевым столбцам: первичный ключ главной таблицы соединяется с внешним ключом подчиненной таблицы. В результате атрибут «Поставщик» выделяется в отдельную таблицу «Поставщики», то же самое происходит и с атрибутами накладной – они переносятся в таблицу «Приходные накладные» (рис. 5.5).

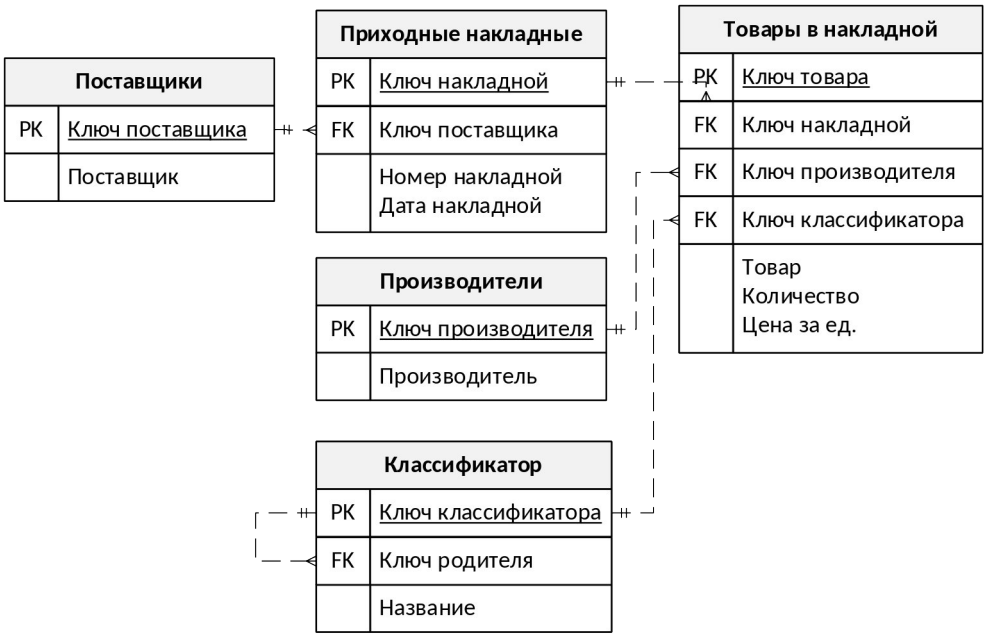



**Рис. 5.5.** Фрагмент БД с таблицами, приведенными к 3NF

***Определение***
Таблица приведена к третьей нормальной форме, если она соответствует 2NF и в ней отсутствуют транзитивные зависимости.

## Нормальная форма Бойса-Кодда

Помимо 3NF, специалисты выделяют усиленную разновидность 3NF – нормальную форму Бойса-Кодда 3NFBC (Boyce-Codd). Смысл усиления в том, что во время формулирования первоначальных требований к 3NF Кодд не предусмотрел вероятность того, что в нормализуемом отношении может существовать более одного потенциального ключа, указанные ключи окажутся составными и эти ключи станут обладателями хотя бы одного общего атрибута.

Вероятность совместного возникновения перечисленных событий крайне невысока, но все-таки не исключена. Поэтому и предусмотрено более строгое определение 3NFBC. Прежде чем мы с ним познакомимся, вспомним, что левая часть символической записи A → B называется детерминантом функциональной зависимости. А теперь дошла очередь и до определения.

***Определение***
Таблица приведена к третьей нормальной форме Бойса-Кодда, когда детерминанты всех ее функциональных зависимостей являются потенциальными ключами.

Вспоминать о нормальной форме Бойса-Кодда стоит только в ситуации, когда в таблицах мы активно применяем составные ключи. Если же при проектировании БД разработчик опирается на идею искусственных (например, автоинкрементных), состоящих из одного атрибута ключей, о существовании 3NFBC можно забыть

## Четвертая нормальная форма

Если третья нормальная форма призвана для борьбы с транзитивными зависимостями, то героиня этой части главы 4NF состоит в конфронтации с другим злом реляционной модели – многозначными зависимостями. **Многозначная зависимость –** это не что иное, как связь «многие ко многим».

Итак, база данных, соответствующая четвертой ступени нормализации, обязана избавиться от многозначных зависимостей между атрибутами отношений. Но каким образом? Ведь реляционная модель специализируется только на связях «один к одному» (1:1) и «один ко многим» (1:M).

Для реализации связи «многие ко многим» (M:N) приходится идти на хитрость. Она заключается во введении дополнительной таблицы, которая разделит связь M:N на две 1:M. Мы так уже поступали при изучении ER-модели. Возвратитесь к рисункам 4.6 и 4.7, на них приведен классический пример связи M:N.

***Определение***
Таблица приведена к четвертой нормальной форме, если она соответствует 3NF и в ней отсутствуют многозначные зависимости (многие ко многим).

## Пятая нормальная форма

Для обычного разработчика БД пятая нормальная форма представляет скорее теоретический, нежели практический интерес. 5NF требует обеспечения беспрепятственной возможности перестройки данных в нормализованных таблицах. Приведение таблицы к высшей степени нормализации – крайне редкий случай. Это действие имеет смысл, только если таблица содержит так называемые зависимые сочетания.

> ***Определение***
> **Зависимые сочетания** – это свойство декомпозиции, которое вызывает генерацию ложных строк при обратном соединении декомпозированных отношений с помощью операции естественного соединения.

## Задание

На основе разработанной вами на предыдущем занятии ER-модели базы данных, создайте физическую модель данных. При формировании таблиц добейтесь, чтобы таблицы были нормализованы до четвертой нормальной формы. Реализуйте базу данных в среде Microsoft Access. Заполните БД демонстрационными данными (не менее 10 строк на каждую из таблиц).

# 6. Язык SQL, оператор SELECT

*Вид занятия – лабораторное занятие*
*Время занятия – 4 часа*
*Программное обеспечение – среда Microsoft Access*

Инструкции SELECT являются едва ли не самым важным элементом языка SQL, ведь именно для выбора данных изначально и создавался язык запросов (в переводе с англ. «select» означает «выбрать»). Исходным материалом для операций выборки выступают одна или несколько таблиц и представлений. Результатом выполнения SQL-запроса, основанного на инструкции SELECT, всегда будет отношение. В интерактивном режиме оно просто выводится на экран компьютера, но при желании содержимое полученного отношения даже может быть отредактировано и сохранено (правда, не средствами SELECT).

Содержимое отношения, полученного в результате запроса, определяется программистом, в простейшем случае оно может полностью повторять исходные данные, может представлять собой объединение нескольких таблиц, может представлять собой определенную выборку из исходных данных. Ко всему прочему запросы SELECT активно используются при создании ряда ключевых объектов БД (представлений, хранимых процедур и триггеров).

Базовая синтаксическая конструкция запроса выглядит следующим образом:

```
SELECT [DISTINCT|ALL]
    { имя поля [AS псевдоним] [,…]
     |функция_агрегирования [AS псевдоним] [,…]
     |выражение для вычисления значения [AS псевдоним] [,…]
     |спецификатор.* }
FROM
   {имя_таблицы [AS псевдоним] [,…]
   |имя_представления [AS псевдоним][,…]}
[WHERE условия отбора]
   [GROUP BY имя поля [,…]] [HAVING условие]
   [ORDER BY имя поля [ASC | DESC] [,…]]
```

> ***Замечание***
> В предложенной конструкции не отражен порядок описания запроса, охватывающего несколько таблиц и представлений. Многотабличные запросы SELECT мы рассмотрим немного позднее.

Запросы на выборку данных всегда начинаются с ключевого слова SELECT, затем следует перечень столбцов, которые мы планируем увидеть в результате выполнения запроса. В перечень, кроме названий столбцов таблиц, могут входить агрегирующие функции, выражения и просто константы. За списком столбцов следует еще одно обязательное слово FROM, а за ним – имена таблиц или представлений, из которых будет производиться отбор данных.

В самом простейшем случае запрос должен включать в себя операторы SELECT и FROM, имя таблицы и спецификатор «*» (листинг 6.1).

**Листинг 6.1. Выборка всех данных из таблицы**

```
SELECT * FROM suppliers;
```

> ***Замечание***
> **В качестве опорной БД для изучения SQL используется БД “Склад” (см. приложение 1)**.

Символ «*» говорит о том, что в результирующее отношение попадут все столбцы из таблицы поставщиков suppliers. Это не всегда желательный способ организации запроса, так как выборка всех столбцов значительно повышает объем запрашиваемых данных и замедляет выполнение запроса. Поэтому рекомендуется быть более конкретным и указывать минимально необходимый набор столбцов, допустим так, как предложено в листинге 6.2.

**Листинг 6.2. Выборка отдельного столбца из таблицы**

```
SELECT supplier FROM suppliers;
```

Единственная строка кода потребует, чтобы в результирующем наборе данных присутствовал только один столбец с именем поставщика. Если понадобится выбрать несколько столбцов, то их имена разделяются запятыми (листинг 6.3).

**Листинг 6.3. Выборка двух столбцов из таблицы**

```
SELECT dnnum, dndate FROM deliverynotes;
```

Если таблица содержит повторяющиеся строки, которые вы не хотите видеть в результирующем отношении, то сразу после инструкции `SELECT` следует вставить предикат `DISTINCT` (листинг 6.4).

**Листинг 6.4. Исключение дубликатов строк**

```
SELECT DISTINCT vendor FROM vendors;
```

Обратный предикат `ALL` указывает на то, что мы намерены получить все строки данных, включая и дубликаты. Впрочем, в явном применении `ALL` необходимости нет, так как это режим по умолчанию.

> ***Замечание***
> В большинстве диалектов SQL с помощью инструкции `SELECT` можно вычислять значения без обращения к какой-либо таблице. Проверьте это утверждение, отправив на выполнение запрос `SELECT 2+2`.

## Псевдонимы имен столбцов и таблиц

В тех случаях, когда запрос достаточно сложен и содержит длинные названия столбцов и таблиц, а также при использовании агрегирующих функций или столбцов, появившихся в результате выполнения выражений, программисты могут ввести псевдонимы столбцов и таблиц. Для этого применяется ключевое слово `AS` (листинг 6.5).

**Листинг 6.5. Создание искусственного столбца с псевдонимом**

```
SELECT CONCAT('№ ', dnnum,' от ', dndate) AS dinfo FROM deliverynotes;
```

В представленной выше строке запроса мы описываем выражение, объединяющее номер и дату накладной, и присваиваем этому столбцу псевдоним «`dinfo`». Обратите внимание на то, что в примере для конкатенации строк мы задействовали функцию `CONCAT()`. Однако это не догма, например в InterBase для соединения строк используют оператор «`||`».

В SQL предусмотрена возможность определения псевдонимов и для таблиц, в таком случае ключевое слово `AS` нам не понадобится – псевдоним просто указывается после названия таблицы (листинг 6.6). Псевдонимы таблиц позволяют сократить объем кода и способны повысить его наглядность.

**Листинг 6.6. Создание псевдонимов для таблиц**

```
SELECT * FROM goodslist g, transfers t
WHERE t.goodslist_id=g.goodslist_id;
```

Запрос SQL может быть адресован сразу нескольким таблицам и даже таблицам, физически расположенным в разных базах данных. В таких случаях при перечислении столбцов перед собственно именем следует предварительно указывать имя БД и имя таблицы (листинг 6.7).

**Листинг 6.7. Фрагмент инструкции SELECT с полным именем столбца таблицы**

```
SELECT warehouse.goodlists.goods, warehouse_2.goodlists.amount, …
```

Сразу после предложения `FROM` может следовать не только таблица, но и альтернативный источник данных, например представление.

## Порядок сортировки, ORDER BY

Полученный в результате выполнения запроса набор строк может быть упорядочен. Порядок сортировки данных определяется при помощи команды `ORDER BY` и перечня столбцов, принимающих участие в сортировке. Пример, представленный в листинге 6.8, осуществит сортировку строк таблицы `vendors` по имени поставщика.

**Листинг 6.8. Сортировка таблицы поставщиков по одному столбцу**

```
SELECT vendor, phonenum FROM vendors ORDER BY vendor;
```

В порядке сортировки допускается указать сразу несколько столбцов, разделяя их запятыми.

По умолчанию записи упорядочиваются по возрастанию алфавита (от А до Я). Если необходимо осуществить сортировку в порядке убывания, то после имени столбца поставьте оператор DESC (листинг 6.9).

**Листинг 6.9. Сортировка данных в порядке убывания даты**

```
SELECT * FROM deliverynotes ORDER BY dndate DESC;
```

Вне зависимости от используемых в запросе предикатов и инструкций, фраза ORDER BY всегда должна быть последним элементом в предложении SELECT.

# Условие отбора данных, предложение WHERE

В результате выполнения запроса пользователь рассчитывает получить не все данные, а какую-то логическую выборку только необходимых в данный момент строк. Например, товары для кухни, товары, поступившие на склад за последний месяц, или товары, пользовавшиеся наибольшим спросом в предыдущем квартале. Для того чтобы ограничить набор выводимых данных, необходимо в конструкцию SELECT добавить предложение WHERE. Совместно с WHERE применяется следующий перечень типов ограничений.

1. Сравнение. Проводится сравнение результатов вычисления одного выражения с другим. Сравнения осуществляются с помощью операторов, >, <, =, >=, <=, <>. Результат сравнения может принимать значения: TRUE, FALSE и UNKNOWN.
2. Попадание в диапазон. Проверяется, попадает ли результат вычисления выражения в определенный диапазон значений.
3. Соответствие шаблону. Проверяется, соответствует ли некоторое строковое значение заданному шаблону.
4. Неопределенность. Содержит ли поле неопределенное значение NULL.
5. Проверка существования. Проверяется факт существования значения в выходных результатах вложенных подзапросов к другим отношениям БД.

Все перечисленные ограничения описываются сразу после инструкции WHERE. В одном запросе допускается комбинировать несколько ограничений, с этой целью задействуются логические операторы AND, OR и NOT.

## Сравнение

Сравнение – наиболее распространенное условие ограничения результирующего набора данных (листинг 6.10).

**Листинг 6.10. Выборка строк с указанием минимальной даты**

```
SELECT * FROM deliverynotes WHERE dndate>='01/01/2021';
```

Приведенное выражение выберет информацию о контрактах, заключенных начиная с 1-го января 2021 года.

Другой вариант более сложного запроса (листинг 6.11) возвратит данные о товарах, габаритный размер которых не превышает 1 метра по осям X, Y и Z (при условии что в столбцах, отвечающих за хранение размеров, за единицу измерения принят миллиметр).

**Листинг 6.11. Сложное условие выборки на основе AND**

```
SELECT * FROM goodslist WHERE
              SizeX<=1000 AND SizeY<=1000 AND SizeZ<=1000;
```

В листинге 6.13 для объединения трех условий мы использовали ключевое слово AND (логическое «И»). Кроме этого, в тексте SQL запроса можно использовать условие OR (логическое «ИЛИ»). Например, с помощью запроса из листинга 6.12 товаровед сможет узнать, есть ли на складе продукция с весом брутто более 100 Кг или габаритами более 1 кубометра.

**Листинг 6.12. Сложное условие выборки на основе AND и OR**

```
SELECT goodslist.*, (SizeX*SizeY*SizeZ) AS volume
FROM goodslist
WHERE amount>0 AND (grossweight>100 OR SizeX*SizeY*SizeZ>1000000000);
```

***Замечание***

Если необходимо построить запрос на основе комбинации нескольких условий AND и OR, то следует пользоваться круглыми скобками. Условия, заключенные в круглые скобки, должны выполняться совместно.

## Попадание в диапазон, BETWEEN

Предикат BETWEEN осуществляет контроль за тем, чтобы интересующий нас результат входил в определенный диапазон значений.

```
<значение> BETWEEN <начало диапазона> AND <окончание диапазона>
```

Или наоборот, чтобы результат не принадлежал определенному диапазону.

```
<значение> NOT BETWEEN <начало диапазона> AND <окончание диапазона>
```

Один из примеров применения BETWEEN представлен в листинге 6.13.

**Листинг 6.13. Ограничения диапазона дат**

```
SELECT * FROM deliverynotes WHERE dndate
                        BETWEEN '2018/01/01' AND '2018/12/31';
```

Пример вернет все контракты, заключенные в 2018 году начиная с первого января и заканчивая последним днем декабря.

## Соответствие шаблону, LIKE

Если вы столкнулись со столь сложным условием выборки, что с ним оказываются не способны справиться ни STARTING WITH, ни CONTAINING, то обязательно обратите внимание на предикат LIKE, который определяет условия поиска в форме шаблона.

```
<значение> [NOT] LIKE <шаблон подстроки>
[ESCAPE служебный символ подстановки]
```

Например, для выбора всех поставщиков, в имени которых есть символ «о», мы задействуем запрос из листинга 6.14.

**Листинг 6.14. Поиск поставщиков, в имени которых есть символ «о»**

```
SELECT * FROM suppliers WHERE supplier LIKE '%о%';
```

Результат выполнения представленного запроса проиллюстрирован на рис. 6.2.

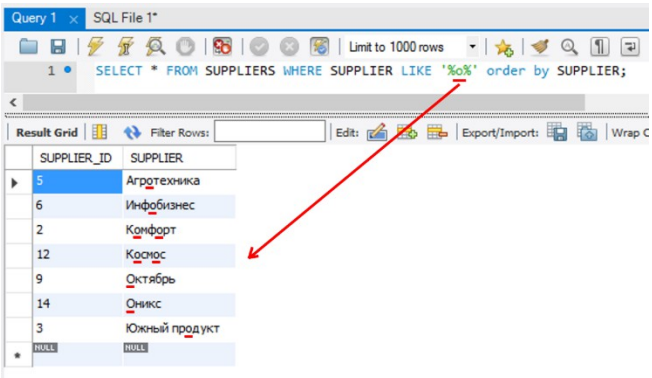



**Рис. 6.1.** Применение LIKE в запросе SELECT

В шаблоне подстроки, помимо набора интересующих нас символов, допускается применять два трафаретных символа:

1) символ подчеркивания «_», он используется вместо любого одиночного символа в выражении;
2) символ процента «%», заменяющий последовательность любых символов.

Например, для отбора всех поставщиков, у которых во второй позиции названия есть символ «к» и название заканчивается на «о», выражение должно выглядеть так: `'_к%о'`.

## Выбор неопределенных или определенных значений

Нам уже знаком предикат `IS NULL`, предназначенный для проверки факта отсутствия или наличия значений в столбце отношения. Посмотрим, как он будет работать в предложении `WHERE`.

Пример из листинга 6.15 возвратит все кортежи, у которых в столбце `WEIGHT` отсутствует значение.

**Листинг 6.15. Выборка строк с неопределенным значением в столбце**

```
SELECT * FROM goodslist WHERE weight IS NULL;
```

Если вы, наоборот, не намерены выбрать строки, в которых имеется неопределенность, то запрос должен выглядеть иначе, чем в листинге 6.16.

**Листинг 6.16. Выборка строк с определенным значением в столбце**

```
SELECT * FROM deliveries WHERE amount IS NOT NULL;
```

> ***Замечание***
> При проверке значения на неопределенность не стоит повторять хрестоматийную ошибку начинающих разработчиков БД – не используйте конструкции, подобные следующим: `WHERE X=NULL` или `WHERE X<>NULL`.

## Вложенные запросы и проверка существования

Инструкция `SELECT` допускает вложение в запрос неограниченного количества подзапросов, благодаря которым можно создавать весьма сложные многоярусные конструкции выборки данных. Простейшая форма подзапроса предложена в примере 6.17.

**Листинг 6.17. Подзапрос**

```
SELECT * FROM SUPPLIERS WHERE SUPPLIER_ID=
                    (SELECT SUPPLIER_ID FROM DELIVERYNOTES
                     WHERE DNDATE BETWEEN '01.01.2018' and '31.01.2018');
```

Здесь мы пытаемся узнать данные поставщика, с которым был заключен договор на поставку продукции в январе 2018 года. Однако предложенный подход не пользуется популярностью у программистов, потому что предназначен для возврата всего одной строки.

Поэтому обычно вложенные подзапросы обладают более сложным синтаксисом и создаются на основе специальных предикатов: `IN`, `EXISTS`, `ALL`, `ANY`, `SINGULAR` и `SOME`. Благодаря перечисленным предикатам программист получает возможность проверить факт существования определенного значения в выходном наборе данных вложенного подзапроса.

## Подзапрос IN

Оператор подзапроса `IN` производит проверку, входит ли результат вычисления в заданное множество. Множество может описываться как поэлементно, так и быть результатом другого, вложенного подзапроса.

```
<значение> [NOT] IN (подзапрос SELECT|элементы множества)
```

Например, в листинге 6.18 мы осуществляем выборку производителей продукции, соответствующих элементам заданного множества.

**Листинг 6.18. Выборка строк, соответствующих элементам множества**

```
SELECT * FROM VENDORS WHERE VENDOR IN ('Asus', 'Intel', 'IBM');
```

В основной запрос разрешено вкладывать подзапрос, он следует за инструкцией `WHERE`. Подзапрос описывается благодаря уже знакомому нам оператору `SELECT`. Допустим, что нам требуется узнать, имеются ли в таблице `VENDORS` производители, продукция которых никогда не поступала на склад. Решение поставленной задачи представлено в листинге 6.19.

**Листинг 6.19. Пример вложения подзапроса с помощью предиката IN**

```
SELECT * FROM VENDORS
WHERE VENDORS_ID NOT IN (SELECT VENDORS_ID FROM GOODSLIST);
```

Предикат IN определяет, будут ли значения строки обнаружены в наборе значений, который либо определен, либо получен по результатам табличного подзапроса. Обратная задача решается при удалении ключевого слова NOT.

Представленный выше пример принадлежности ко множеству одновременно демонстрирует порядок создания вложенных подзапросов. Язык SQL не лимитирует количество вложений, поэтому механизм вложенных подзапросов является весьма мощным оружием при построении сложных отчетов.

### Проверка существования EXISTS

Задача предиката EXISTS заключается в проверке факта существования строк, удовлетворяющих определенному критерию. Предикат может использоваться только совместно с подзапросами.

```
[NOT] EXISTS (SELECT * FROM <имя_таблицы> WHERE <условие отбора>)
```

Результатом выполнения предиката окажется не какой-то выходной набор данных, а просто булево значение TRUE или FALSE. Логическая истина получится в том случае, если в отобранном подзапросом наборе данных присутствует хотя бы одна запись. Ключевое слово NOT EXISTS использует обратные правила обработки.

Листинг 6.20 предлагает один из вариантов использования предиката EXISTS.

**Листинг 6.20. Пример проверки существования**

```
SELECT * FROM VENDORS WHERE EXISTS
(SELECT VENDORS_ID FROM GOODSLIST WHERE AMOUNT>0
                    AND GOODSLIST.VENDORS_ID=VENDORS.VENDORS_ID);
```

В рассмотренном примере мы получим список производителей, чьи товары имеются на нашем складе. Запрос будет выполнен очень быстро за счет того, что подзапрос выборки поставщиков не стремится пройти по всем строкам таблицы GOODSLIST, а просто находит первую соответствующую критерию поиска строку в запросе и возвращает TRUE.

> ***Замечание***
> Основное достоинство предиката существования – высокая скорость выполнения подзапроса, и если вы хотите сформировать запрос, основанный на предположении о существовании каких-то данных, то EXISTS станет вашим лучшим помощником.

## Агрегирующие функции

Агрегирующие (обобщающие) функции в качестве исходных параметров принимают значения, указанные в запросе (после слова SELECT), и вычисляют результат. Как правило, эти функции применяются в запросах группировки с помощью команды GROUP BY. Наиболее распространенные функции представлены в табл. 6.1.

***Таблица 6.1.*** *Агрегирующие функции*

| Функция | Описание |
|---|---|
| COUNT | Возвращает количество строк. Тип возвращаемого значения – целое число.<br>`SELECT COUNT(*) FROM имя_таблицы;` |
| AVG | Вычисляет среднее арифметическое для указанных элементов. Используется только для полей цифровых типов данных. Тип возвращаемого значения – вещественное число.<br>`SELECT AVG(имя_столбца) FROM имя_таблицы;` |
| SUM | Вычисляет сумму значений. Применяется только для цифровых типов данных.<br>`SELECT SUM(имя_столбца) FROM имя_таблицы;`<br>Будьте внимательны. Если в результате вычисления суммы итоговое число превысит максимальное значение используемого типа данных, то возникнет ошибка |
| MAX | Возвращает наибольшее из всех значений.<br>`SELECT MAX(имя_столбца) FROM имя_таблицы;` |
| MIN | Возвращает наименьшее из всех значений.<br>`SELECT MIN(имя_столбца) FROM имя_таблицы;` |

> ***Замечание***
> Все агрегатные функции работают с единственным полем таблицы и возвращают единственное значение. Функции COUNT, MIN и MAX применимы как к числовым, так и к текстовым полям. Функции SUM и AVG

сохраняют работоспособность только при обслуживании числовых полей. Совместно с перечисленными функциями разрешено применять квантификатор DISTINCT.

# Группировка данных GROUP BY

Предложение GROUP BY предназначено для осуществления группировки выходных строк по какому-либо признаку, например по равенству значений каких-либо столбцов

```
GROUP BY имя_столбца1 [,имя_столбца2, ...];
```

В группирующих запросах зачастую применяются рассмотренные чуть ранее в табл. 6.1 агрегирующие функции.

***Замечание***

В результате выполнения группирующего запроса для каждой отдельной группы создается одна-единственная группирующая строка.

Допустим, что мы хотим узнать, сколько всего единиц продукции каждого из наименований поступило на склад по накладной с определенным первичным ключом. Для этого нам следует обратиться к таблице GOODSLIST и, используя агрегирующую функцию SUM(), построить группирующий запрос 6.21.

**Листинг 6.21. Группировка по наименованию товара**

```
SELECT GOODS, SUM(AMOUNT) AS SUM_AMOUNT  FROM  GOODSLIST
WHERE DELIVERYNOTE_ID=100
GROUP BY GOODS;
```

Обратите внимание на то, каким образом мы присвоили имя результату, возвращенному агрегирующей функцией, – для этого мы воспользовались предикатом **AS**. Еще одно замечание, касающееся особенностей построения группирующих запросов, – все имена столбца, приведенные в списке группирующего запроса SELECT, должны присутствовать и во фразе GROUP BY. Единственное исключение делается для полей, обрабатываемых агрегирующей функцией.

## Дополнительная фильтрация группы строк, HAVING

Предикат HEAVING используется после группировки (предложения GROUP BY) и предназначен для дополнительной фильтрации уже сформированных групп строк. Поведение элемента HEAVING подобно предложению WHERE, но он работает не со всеми строками таблиц, а исключительно со строками результирующего набора. Фильтрация может осуществляться с помощью агрегирующих функций (COUNT, SUM, AVG, MAX и MIN). Для примера несколько модифицируем текст предыдущего запроса и научим его выводить информацию только о накладных с товарами, чья суммарная стоимость превышает 1000 (листинг 6.22).

**Листинг 6.22. Применение предиката HAVING**

```
SELECT GOODS, SUM(AMOUNT * FACTORYPRICE) AS SUM_PRICE FROM GOODSLIST
GROUP BY GOODS
HAVING SUM(AMOUNT * FACTORYPRICE)>1000;
```

***Замечание***

Условие отбора HAVING должно основываться на агрегирующей функции, если это не так, то вместо HAVING следует воспользоваться предложением WHERE.

# Соединение таблиц в запросе SELECT

Рассмотренный в начале главы способ применения инструкции SELECT раскрывает только одну из сторон процесса выборки данных из таблицы и представляет собой частный случай. Значительно чаще возникает необходимость объединения информационных потоков из двух и более таблиц. Для этого в рамках инструкции SELECT задействуют ряд дополнительных синтаксических структур, позволяющих работать одновременно с несколькими таблицами. В SQL предусмотрено три базовых способа построения запросов ко множеству таблиц (рис. 6.3).

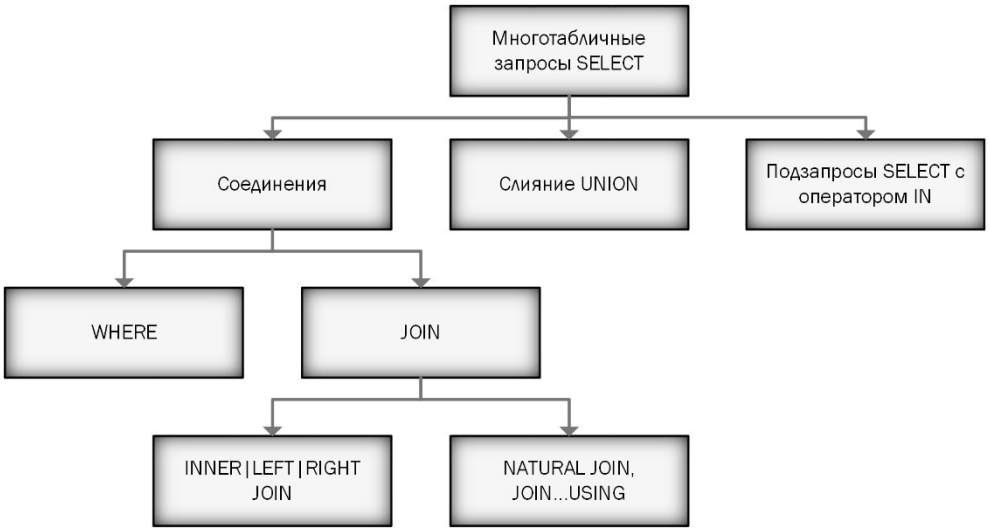



**Рис. 6.2.** Классификация многотабличных запросов

Нам уже известен порядок построения подзапроса с применением оператора `IN`, этот способ выборки нам уже встречался (листинг 6.17). Так что нам предстоит сосредоточить свое внимание на соединениях, осуществляемых с помощью ключевых слов `WHERE`, `JOIN`, и слиянии таблиц `UNION`.

## Внутреннее соединение WHERE

Простейшим способом соединения двух и более отношений является использование инструкции `WHERE` с определением столбцов, применяемых для соединения. Это один из самых старых способов объединения таблиц в реляционных базах данных – он появился на свет вместе с первым стандартом SQL.

Для выполнения объединения после ключевого слова `FROM` необходимо перечислить имена соединяемых таблиц, разделяя их запятыми. Например, мы хотим соединить таблицы поставщиков `SUPPLIERS` и таблицу договоров о поставке `DELIVERYNOTE` (листинг 6.23). Указанные таблицы объединяются по ключевым полям `SUPPLIER_ID` (в таблице поставщиков это первичный ключ, в таблице договоров – внешний).

**Листинг 6.23. Внутреннее соединение таблиц с помощью предложения WHERE**

```
SELECT SUPPLIERS.SUPPLIER_ID, SUPPLIERS.SUPPLIER,
DELIVERYNOTES.DELIVERYNOTE_ID, DELIVERYNOTES.DNNUM, DELIVERYNOTES.DNDATE
FROM SUPPLIERS, DELIVERYNOTES
WHERE DELIVERYNOTES.SUPPLIER_ID=SUPPLIERS.SUPPLIER_ID;
```

Тип представленного в листинге 6.27 соединения называют внутренним соединением, таблицы объединились на основе точного равенства между значениями, хранящимися в двух столбцах, – «внешний ключ = первичный ключ». Основная особенность внутреннего соединения в том, что из упомянутых в запросе таблиц в результирующее отношение попадут исключительно те строки, у которых имеются «напарники». Это проиллюстрировано рис. 6.4 – запрос возвратит только поставщиков «Салют» и «Радуга», ведь именно с ними были заключены контракты. Остальные упомянутые в таблице `SUPPLIER` поставщики в выходное отношение не попадают.

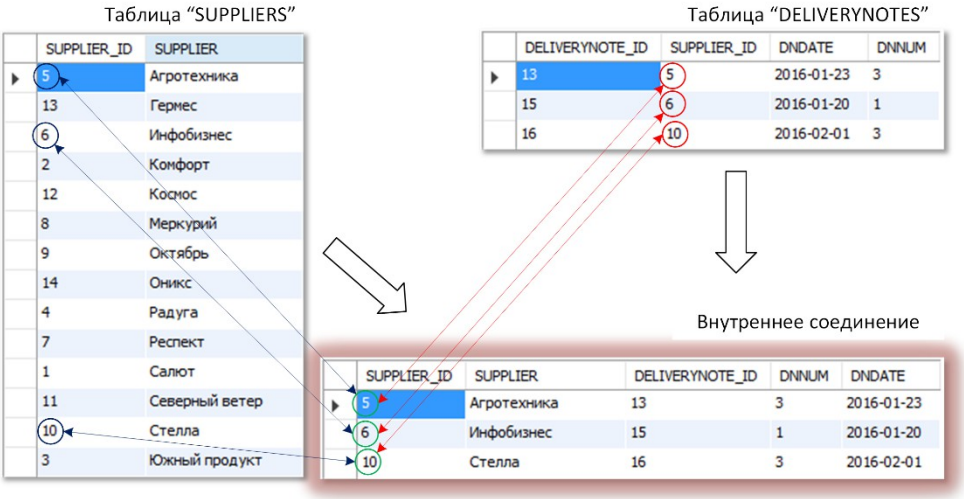



**Рис. 6.3.** Пример внутреннего соединения двух таблиц

Хотя метод построения многотабличных запросов, основанный на `WHERE`, поддерживается и сегодня, разработчику БД целесообразно отдавать предпочтение более совершенному (появившемуся вместе со стандартом SQL-92) способу соединения таблиц с применением ключевого слова `JOIN`.

## Соединение JOIN

С выходом стандарта SQL-92 вместо соединения с помощью предложения `WHERE` стали использовать синтаксические конструкции, опирающиеся на ключевое слово `JOIN`. В SQL таких конструкций несколько. Во-первых, предусмотрено классическое внутреннее соединение:

```
SELECT столбец1 [, столбец2 ...] | *
FROM <левая_таблица> [INNER | CROSS] JOIN <правая_таблица>
[ON <правило соединения>]
[WHERE <условие отбора>];
```

Как видите, в данном случае необходимо задействовать отдельный оператор `JOIN` или дополнять его необязательным ключевым словом `INNER`, подчеркивающим тот факт, что речь идет именно о внутреннем соединении. С помощью `INNER JOIN` можно добиться результата, абсолютно идентичного полученному ранее за счет `WHERE` (листинг 6.23). Для этого достаточно немного изменить структуру запроса `SELECT` (листинг 6.24).

**Листинг 6.24. Внутреннее соединение таблиц с помощью предложения INNER JOIN**

```
SELECT Sp.SUPPLIER, Dn.*
FROM SUPPLIERS Sp
INNER JOIN DELIVERYNOTES Dn ON Dn.SUPPLIER_ID=Sp.SUPPLIER_ID
ORDER BY Sp.SUPPLIER;
```

Во-вторых, в SQL осуществлена поддержка так называемого внешнего соединения, на этот раз программисту следует явным образом указать, какого рода соединение он рассчитывает получить – левое (`LEFT`), правое (`RIGHT`).

```
SELECT столбец1 [, столбец2...] | *
FROM <левая_таблица> {LEFT | RIGHT} JOIN
     <правая таблица> [ON <правило соединения>]
[WHERE <условие отбора>];
```

При написании кода запроса важную роль играет последовательность соединения таблиц, именно поэтому были введены понятия «левая» и «правая» таблицы. По правилам построения запроса `JOIN` имя левой таблицы следует сразу после ключевого слова `FROM`, имя правой таблицы – после `JOIN`.

***Замечание***

Левое внешнее соединение `LEFT JOIN` возвратит все (даже несвязанные) строки левой таблицы и дополнит их связанными строками правой таблицы.

Рассмотрим листинг 6.25, в котором в качестве левой мы задействуем таблицу поставщиков `SUPPLIER`, а на роль правой назначим таблицу договоров `DELIVERYNOTES` на поставку товаров на склад.

**Листинг 6.25. Левое соединение таблиц**

```
SELECT Sp.SUPPLIER, Dn.*
FROM SUPPLIERS Sp
LEFT JOIN DELIVERYNOTES Dn ON Dn.SUPPLIER_ID=Sp.SUPPLIER_ID
ORDER BY Sp.SUPPLIER;
```

Результат, возвращаемый запросом левого внешнего соединения, будет существенно отличаться от внутреннего. На этот раз (рис. 6.5) в выходное отношение оказались включены все строки из левой таблицы `SUPPLIERS`. И это произошло, даже несмотря на то что с большинством поставщиков пока не заключены контракты (об этом свидетельствуют многочисленные неопределенные значения `NULL` в столбцах `DELIVERYNOTE_ID`, `DNNUM`, `DNDATE`). С правой таблицей `DELIVERYNOTES` все обстоит по-прежнему – в результирующее отношение попали только строки, имеющие соответствие по ключу.

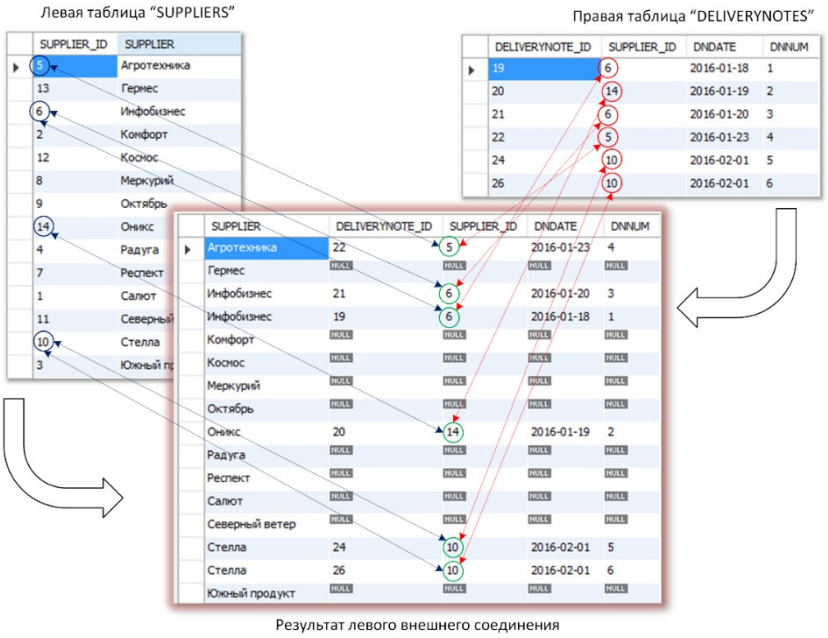



**Рис. 6.4.** Пример левого внешнего соединения двух таблиц

Порядок выполнения правого внешнего соединения `RIGHT JOIN` практически повторяет алгоритм левого, с той лишь разницей, что на этот раз выводу подлежат все записи из правой таблицы и только соответствующие им строки из левой. Пример использования `RIGHT JOIN` представлен в листинге 6.26.

**Листинг 6.26. Правое соединение таблиц**

```
SELECT G.GOODSLIST_ID, G.GOODS, G.AMOUNT, V.VENDORS_ID, V.VENDOR
FROM  GOODSLIST G
RIGHT JOIN VENDORS V ON V.VENDORS_ID=G.VENDORS_ID;
```

В рассматриваемом примере объединятся таблицы производителей `VENDORS` и товаров `GOODSLIST`. При описании операции соединения `RIGHT JOIN` таблица производителей объявляется правой, поэтому в итоговое отношение попадут все имеющиеся в ней строки. Что касается левой таблицы `GOODSLIST`, то на этот раз в выборку попадут только записи, имеющие связь со строками из правой таблицы.

***Замечание***

В ряде СУБД поддерживается полное (`FULL JOIN`) объединение, которое возвращает все строки из левой и правой таблиц.

## Соединение NATURAL JOIN и JOIN…USING

Стандарт SQL постоянно совершенствуется, предлагая разработчикам СУБД вектор развития структурированного языка запросов. По рекомендациям стандарта появились дополнительные синтаксические конструкции JOIN:

`NATURAL JOIN` – естественное соединение;

`JOIN … USING` – соединение с явным указанием столбца.

Суть конструкции естественного соединения `NATURAL JOIN` заключается в том, что SQL самостоятельно выбирает способ соединения таблиц. В первую очередь проверяются одноименные столбцы в левой и правой таблицах, если таковые отсутствуют, то соединение осуществляется по однотипным столбцам (как вы понимаете, в последнем случае результат, скорее всего, не будет соответствовать вашим ожиданиям).

Автор книги при проектировании связей между таблицами всегда в подключаемой таблице создает столбец внешнего ключа с тем же именем, что и у столбца первичного ключа в главной таблице. Поэтому проблем с натуральным соединением (листинг 6.27) в демонстрационной базе данных не возникнет.

**Листинг 6.27. Натуральное соединение таблиц NATURAL JOIN**

```
SELECT * FROM suppliers NATURAL JOIN deliverynotes
WHERE dndate BETWEEN '01.01.2016' AND '31.01.2016';
```

Результат выполнения натурального соединения иллюстрирует рис. 6.5. Для сравнения на нем, кроме экранного снимка `NATURAL JOIN`, предложен снимок запроса `INNER JOIN`, как видите, результаты идентичны – натуральное соединение отработало корректно.

Соединение с явным указанием столбца `JOIN … USING` можно рассматривать как доработку естественного соединения `NATURAL JOIN`. В данном случае исключается неверная трактовка имен соединяемых столбцов (листинг 6.32).

**Листинг 6.28. Соединение таблиц JOIN … USING**

```
SELECT * FROM suppliers
JOIN deliverynotes USING (supplier_id)
WHERE dndate BETWEEN '01.01.2016' AND '31.01.2016';
```

И вновь мы получаем корректное итоговое отношение, это утверждение подтвердит рис. 6.5.

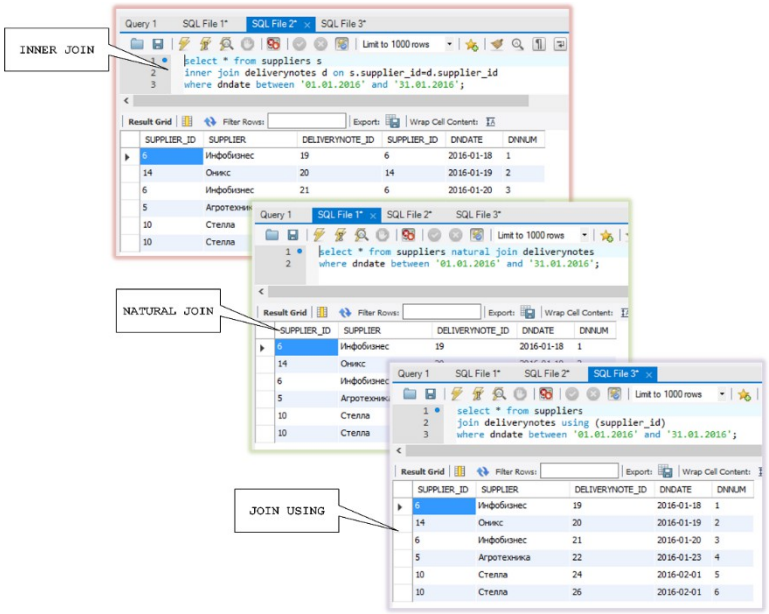



**Рис. 6.5.** Сравнение результатов внутреннего и натурального соединения двух таблиц

***Замечание***

По своей сути соединения `NATURAL JOIN` и `JOIN … USING` выступают просто сервисными расширениями традиционных конструкций `[INNER | LEFT |RIGHT] JOIN` и предназначены для упрощения труда программиста только в случае, если при проектировании таблиц строго соблюдались правила именования столбцов первичного и внешнего ключей.

## Соединение большого количества таблиц

При необходимости в запросе SELECT теоретически допускается соединять бесконечное число таблиц. В нашем примере мы рассмотрим вариант запроса SQL, формирующий единое отношение из 7 таблиц демонстрационной БД (листинг 6.29).

**Листинг 6.29. Внутреннее объединение нескольких таблиц**

```
SELECT * FROM GOODSLIST GL
JOIN GOODSCLASS GC    ON GC.GOODSCLASS_ID=GL.GOODSCLASS_ID
JOIN VENDORS V        ON GL.VENDORS_ID=V.VENDORS_ID
JOIN DELIVERYNOTES D ON D.DELIVERYNOTE_ID=GL.DELIVERYNOTE_ID
JOIN SUPPLIERS S      ON S.SUPPLIER_ID=D.SUPPLIER_ID
JOIN TRANSFERS T      ON T.GOODSLIST_ID=GL.GOODSLIST_ID
JOIN WAREHOUSES W     ON W.WAREHOUSES_ID=T.WAREHOUSES_ID;
```

***Внимание!***
Порядок объединения таблиц имеет критическое значение для скорости выполнения запроса. Если таблицы соединяются в правильном порядке, то общее число обрабатываемых строк будет меньше. Описывая запрос, следует выполнять сначала максимально ограничивающий поиск, чтобы отфильтровать как можно большее число строк на ранних фазах выполнения запроса с соединениями.

## Слияние UNION

Слияние необходимо в случае, когда требуется объединить результаты нескольких запросов в одну результирующую таблицу. Для этих целей предназначено выражение UNION.

```
SELECT ...
UNION [ALL | DISTINCT] SELECT ...
[UNION [ALL | DISTINCT] SELECT ...]
```

Допустим, что у нас имеется две идентичные по структуре таблицы VENDORS_OLD и VENDORS, хранящие сведения о производителях товаров. Для того чтобы получить суммарную информацию, стоит воспользоваться оператором UNION (листинг 6.30).

**Листинг 6.30. Слияние UNION**

```
SELECT * FROM VENDORS_OLD
UNION
SELECT * FROM VENDORS;
```

***Замечание***
Слияние UNION может осуществляться только у совместимых по объединению отношений.

Допустимо объединять и разные отношения, но при условии совпадения количества и типа данных объединяемых столбцов.

По умолчанию запрос UNION удаляет повторяющиеся строки из результирующего отношения (этому режиму соответствует необязательный ключ DISTINCT), однако если их следует сохранить, то стоит позвать на помощь предикат ALL (листинг 6.31).

**Листинг 6.31. Слияние UNION с сохранением дубликатов**

```
SELECT * FROM VENDORS_OLD
UNION ALL
SELECT * FROM VENDORS;
```

***Внимание!***
При слиянии таблиц с помощью UNION из результирующего отношения по умолчанию исключаются строки-дубликаты.

### Задание

С помощью оператора SELECT разработайте группу запросов к своей базе данных. В результате работы у вас должны быть созданы запросы позволяющие:

1. сортировать данные во всех отдельных таблицах БД;
2. выбирать необходимые строки и столбцы из отдельных таблиц;

3. объединяющие все пары таблиц, состоящие в отношении “один ко многим”;
4. объединяющие все таблицы, состоящие в отношении “многие ко многим”;
5. суммарный запрос, отображающий все данные из всех таблиц БД.

Защита лабораторной работы осуществляется путем выполнения, полученного у преподавателя индивидуального задания по реализации инструкции SELECT.

# 7. Язык SQL, операторы манипулирования данными

*Вид занятия – лабораторное занятие*
*Время занятия – 2 часа*
*Программное обеспечение – среда Microsoft Access*

На занятии мы продолжим работу с подмножеством SQL – языком манипулирования данными DML (Data Manipulation Language). DML осуществляет операции вставки, редактирования и удаления данных и базируется на инструкциях INSERT, UPDATE и DELETE.

## Вставка, инструкция INSERT

Назначение инструкции `INSERT` – вставка в таблицу одной или нескольких строк. Практически все диалекты SQL на все 100 % поддерживают синтаксическую конструкцию, рекомендованную стандартом.

```
INSERT INTO {имя таблицы | имя представления}
{имя столбца[,…]}
{DEFAULT VALUES | VALUES (Значение1[,…]) | инструкция SELECT}
```

Имя пополняемой таблицы указывается после ключевых слов `INSERT INTO`. С помощью `INSERT` допускается вставка новой записи и в представление, но при условии, что представление обслуживает одну таблицу. Сразу за именем таблицы следует перечень столбцов, в которые мы собираемся внести новую информацию. Если список столбцов окажется неполным, то в оставшиеся поля записывается значение по умолчанию. Если, в свою очередь, значение по умолчанию не было определено заранее (на этапе создания таблицы) и на столбец наложено ограничение `NOT NULL`, то работа оператора прерывается. Еще одним поводом для приостановки операции вставки новой записи может послужить попытка поместить данные в столбец, который самостоятельно генерирует новое значение (например, в столбцы автоинкрементного типа).

Рассмотрим наиболее распространенную форму применения инструкции `INSERT…INTO`, в которой нам требуется добавить новую строку в таблицу производителей `VENDORS` (листинг 7.1).

**Листинг 7.1. Вставка новой строки с помощью INSERT…INTO**

```
INSERT INTO VENDORS
(VENDOR, PHONENUM, EMAIL, COUNTRY_ID)
VALUES
('SAMSUNG','88002000001','support@SAMSUNG.com', 12);
```

В результате таблица производителей товаров пополнится новой записью. Заметьте, что в тексте запроса отсутствует обязательное значение для столбца первичного ключа таблицы. Дело в том, что задача вставки значения первичного ключа автоинкрементного типа обычно решается автоматически.

> ***Замечание***
> В команде на добавление новой строки перечисление столбцов не является обязательным, если вы осуществляете вставку данных во все столбцы. Но в этом случае последовательность передаваемых значений должна четко соответствовать физическому порядку столбцов таблицы.

При реализации команды `INSERT` некоторые разработчики СУБД расширили стандартные возможности SQL. Например, MySQL и DB2 позволяют за одно обращение к команде `INSERT` вставлять в таблицу несколько строк (листинг 7.2).

**Листинг 7.2. Вставка нескольких строк с помощью INSERT…INTO**

```
INSERT INTO COUNTRYS
(COUNTRY)
VALUES
('Австрия'), ('Бельгия'), ('Южная Корея'), ('Япония');
```

Такое решение весьма удобно, особенно когда требуется добавить все строки в таблицу в рамках единственной транзакции.

Стандартом SQL:2003 рекомендуется производителям доработать свои диалекты SQL так, чтобы они позволяли вставлять в целевую таблицу строки, извлекаемые инструкцией SELECT из другой таблицы. Пример решения этой задачи раскрывает листинг 7.3.

**Листинг 7.3. Вставка записей INSERT…SELECT**

```
INSERT INTO SUPPLIERS
(SUPPLIER)
(SELECT SUPPLIER FROM SUPPLIERS_2);
```

Условием корректности выполнения такой команды должно быть соответствие типов столбцов целевой и донорской таблиц.

# Модификация, инструкция UPDATE

Вам никогда не хотелось все исправить? Если да, то сейчас мы поговорим о самой главной команде в нашей с вами жизни – инструкции редактирования данных UPDATE.

```
UPDATE {имя таблицы | имя представления}
SET {{имя_столбца=
      выражение | значение | NULL | DEFAULT | ARRAY | ROW=строка}[,…]}
[WHERE условие отбора | WHERE CURRENT OF имя курсора]
```

Минимальный код, позволяющий за один присест изменить все записи в таблице, выглядит так, как представлено в листинге 7.4.

**Листинг 7.4. Модификация значения столбца для всех строк таблицы**

```
UPDATE VENDORS SET EMAIL='VENDOR@MAILOPERATOR.COM';
```

Предложенную строку вряд ли стоит когда-нибудь применять в реальных проектах. Почему? Хотя бы потому, что она одним махом поменяет адреса электронной почты всех производителей товара на адрес VENDOR@MAILOPERATOR.COM. Как вы понимаете, в данном случае такой поступок не вполне корректен.

Неоспоримое удобство инструкции UPDATE заключается в возможности практически мгновенного изменения данных не только в отдельной, но и в нескольких строках или даже во всей таблице. Допустим, что существует некая таблица SALES, в одном из полей которой хранится значение заработной платы сотрудников компании. Представленный пример разработан специально для тех, кто хочет попробовать проявить себя на поприще благотворительности (листинг 7.5).

**Листинг 7.5. Удвоение зарплаты**

```
UPDATE SALES SET SALARY=SALARY*2 WHERE DEPARTMENT_ID=1;
```

Благодаря единственной строке кода мы умудрились повысить зарплату всем сотрудникам отдела, идентифицированного ключом DEPARTMENT_ID=1, ровно в 2 раза. Если после этого вас не уволят, то, скорее всего, вы не программист, а руководитель предприятия…

Если мы ошиблись очень сильно и нам необходимо одновременно внести изменения в несколько полей, то следует разделять пары «имя поля = значение» запятыми (листинг 7.42).

**Листинг 7.6. Исправление значений в двух столбцах таблицы**

```
UPDATE GOODSLIST
SET WEIGHT=100, GROSSWEIGHT=110
WHERE GOODSLIST_ID=2567;
```

При присваивании нового значения столбцу с помощью предложения SET мы можем не только передавать в столбец константу или результат выражения, но и значение по умолчанию, для этого надо воспользоваться ключевым словом DEFAULT.

***Замечание***
Во многих СУБД редактирование данных осуществляется следующим образом: строка со старой информацией помечается как удаленная, а в таблицу вставляется новая строка, с обновленными данными. А для того чтобы не нарушить целостность данных, в эту запись переносится старое значение первичного ключа.

# Удаление, инструкция DELETE

Инструкция `DELETE` специализируется на удалении строк из таблиц данных.

```
DELETE FROM имя_таблицы
[{WHERE предикат}]
| {WHERE CURRENT OF имя_курсора}
```

***Замечание!***
В большинстве СУБД, в том числе и у MySQL, операция удаления не приводит к немедленному физическому стиранию строки. Вместо этого специальный служебный бит удаления строки помечается соответствующим образом, после чего сервер начинает полагать, что строки не существует.

Минимальная команда включает инструкцию `DELETE FROM` и имя таблицы (листинг 7.7).

**Листинг 7.7. Удаление всех строк из таблицы**

```
DELETE FROM ARTICLE
```

Такая команда полностью очистит таблицу, что в обычных случаях не применяется, так как скорее напоминает вредительство. Для того чтобы осуществить адресное удаление строк, необходимо после предиката `WHERE` описать условие удаления (листинг 7.8).

**Листинг 7.8. Составное условие удаления строк из таблицы**

```
DELETE FROM DELIVERYNOTES
WHERE DNNUM>100 AND DNDATE BETWEEN '2021-01-10' AND '2021-01-15';
```

Как видите, условия удаления могут быть достаточно сложными, в данном примере мы удалили только те записи из таблицы, которые хранят сведения о договорах с номером больше 100 и были заключены в январе 2021 года.

## Задание

Применительно к созданной вами БД разработайте запросы, позволяющие:

очищать таблицы;

заполнять таблицы данными;

редактировать определенную строку в таблице;

удалять определенную строку в таблице.

Защита лабораторной работы осуществляется путем выполнения, полученного у преподавателя индивидуального задания по реализации инструкций INSERT, UPDATE и DELETE.

# 8. Язык SQL, операторы определения данных

*Вид занятия – лабораторное занятие*
*Время занятия – 2 часа*
*Программное обеспечение – среда Microsoft Access*

Язык определения данных DDL (Data Definition Language) в нацелен на решение вопросов создания `CREATE` и удаления `DROP` различных объектов базы данных – таблиц `TABLE`, представлений `VIEW`, индексов `INDEX`, курсоров `CURSOR`, определений доменов `DOMAIN`, и т.д.

## Основные типы данных SQL-92

Каждое поле в реляционной таблице базы данных SQL характеризуется определенным типом данных. Основные типы данных практически всех СУБД эквивалентны друг другу. Указание типа данных в SQL понадобится в первую очередь при создании таблиц. С достаточной степенью условности можно говорить о существовании семи основных типов данных SQL (по крайней мере, такое разделение предусматривает стандарт):

1. `CHARACTER STRING` (текст);
2. `NATIONAL CHARACTER` (символы национальной кодировки);
3. `BIT STRING` (двоичные данные);
4. `APPROXIMATE NUMERIC` (число, имеющее приближенное значение);
5. `EXACT NUMERIC` (точное число);
6. `DATETIME` (дата и время);
7. `INTERVAL` (период времени).

Первые два типа данных представляют собой обычный текст, с той лишь разницей, что `NATIONAL CHARACTER` способен работать с национальным набором символов.

*Таблица 8.1. Текстовые данные*

| ***Спецификация*** | ***Описание*** |
|---|---|
| `CHARACTER [n]`<br>или<br>`CHAR [n]` | Тип данных предназначен для создания текстовой строки фиксированной длины. Количество символов в строке определяется в квадратных скобках после указания типа данных – *n*. Если в поле типа `CHAR` помещается текстовое значение меньшего размера, то оставшиеся позиции символов забиваются пробелами. |
| `CHARACTER VARYNG [n]`<br>или<br>`VARCHAR [n]` | Текстовая строка переменной длины. Максимальный размер строки определяется в квадратных скобках. Преимущество такого типа данных над типом `CHAR` заключается в том, что здесь пустые позиции не заполняются пробелами и соответственно таблица требует меньшего размера памяти. |
| `NATIONAL CHARACTER [n]`<br>или<br>`NCHAR [n]` | Строка специального (национального) символьного набора фиксированной длины. |
| `NATIONAL CHARACTER VARYING [n]`<br>или<br>`NCHAR VARYING [n]` | Строка национального символьного набора переменной длины. Национальные типы данных описывают символы в системе UNICODE, в этом случае для хранения одного символа требуется два байта. |

Тип данных **двоичная** (битовая) **последовательность** предназначен для хранения последовательностей двоичной информации. Такой тип данных часто называют Binary Large Objects (BLOB). Тип данных достаточно универсален и позволяет хранить как простейшие логические данные, так и огромные изображения, звуковые файлы и т.д.

*Таблица 8.2. Двоичные последовательности*

| ***Спецификация*** | ***Описание*** |
|---|---|
| `BIT [n]` | Битовая последовательность фиксированной длины. Аргумент *n* устанавливает длину последовательности в битах. Если аргумент отсутствует (установлен в 1), то тип данных используется для создания полей логического типа (Да/Нет). |

| | |
|---|---|
| BIT VARYNG [n] | Битовая последовательность переменной длины. Максимальное значение битовой последовательности указывается в аргументе *n*. |

На базе битовых последовательностей в различных СУБД реализованы свои собственные типы данных. Как уже упоминались ранее, эти типы данных умеют хранить большие объемы форматированного и неформатированного текста, изображения, OLE объекты, файлы мультимедиа и др.

Тип данных **округленные числа** предназначен для определения чисел с плавающей точкой, осуществляющий хранение числа в научном формате (мантисса плюс порядок) и имеет аргумент точность(n), но в отличие от типа numeric не имеет масштаба. По этой причине APPROXIMATE NUMERIC склонен к ошибке округления и предназначен для хранения значений не требующих высокой точности.

*Таблица 8.3. Округленные числа*

| ***Спецификация*** | ***Описание*** |
|---|---|
| REAL | Точность и предел значений определяется автоматически СУБД. Как правило, занимает в памяти 6 байт и может хранить число в интервале от -3,4E-38 до +3,4E+38 с точностью до 7 цифр после запятой |
| FLOAT [(n)] | Параметром n определяется (но не обязательно) минимальное значение точности. Как правило, занимает в памяти 8 байт (если значение n не указано). Тип данных способен хранить число в интервале от 1,7E-308 по +1,7E-308 с точностью до 15 знаков |
| DOUBLE PRECISION | Точность определяется автоматически СУБД, но должна превышать точность REAL |

Тип данных **точные числа** предназначен для хранения чисел, точно соответствующих их цифровой записи. При задании такого типа данных необходимо указать два аргумента: «точность» (*n*) и «масштаб» (*m*). Точность задает общее число значащих цифр, используемых при отображении числа. Масштаб определяет число значащих цифр справа от десятичной точки. Безусловно должно соблюдаться условие: точность больше масштаба (*n*>*m*). Аргумент «масштаб» не является обязательным и если его не указывать, то считается равным 0.

*Таблица 8.4. Точные числа*

| ***Спецификация*** | ***Описание*** |
|---|---|
| NUMERIC [(n[,m])] | Точное число, описываемое аргументами n и m. |
| DECIMAL [(n[,m])]<br>или<br>DEC [(n[,m])] | В отличие от NUMERIC реально хранит число с большей точностью, чем определено в аргументе *m*. Другими словами говорят, что NUMERIC задает реальное значение точности, а DECIMAL – минимальное значение точности. |
| INTEGER<br>или<br>INT | Тип данных, предназначен для хранения целых чисел. Это частный случай типа данных NUMERIC, у которого масштаб установлен в 0, а точность определена возможностями СУБД. Как правило, занимает в памяти 4 байта. |
| SMALLINT | Тип данных предназначен для хранения малых целых чисел. Также, как и INTEGER это частный случай типа данных NUMERIC. Как правило, занимает в памяти 2 байта. |

Количество байт, необходимых для хранения NUMERIC и DECIMAL в памяти компьютера, состоит в прямой зависимости от размерности параметра *n*. Например, если вы хотите хранить число из 15-16 значащих цифр потребуется 8 байт, 34-36 цифр – 16 байт.

Практически во всех СУБД для работы с денежными величинами на базе NUMERIC реализован свой собственный специализированный тип данных. Такие типы данных хранят действительные числа с точностью до 4-го знака после запятой.

По сути, тип данных **дата и время** (DateTime) представляет собой структуру с отдельными полями для хранения даты и времени.

*Таблица 8.5. Дата и время*

| ***Спецификация*** | ***Описание*** |
|---|---|
| DATE | Тип данных включает три поля:<br>YEAR (год) – от 0001 до 9999<br>MONTH (месяц) – от 01 до 12 |

| | |
|---|---|
| | DAY (день) – от 01 до 31<br>Формат записи: “yyyy–mm–dd”.<br>Полное число позиций (вместе с разделителями) 10. |
| TIME [(n)] | Тип данных включает три поля:<br>HOUR (год), MINUTE (минута), SECOND (секунда). Если не указан аргумент точности (*n*), то полное число позиций (вместе с разделителями) равно 8, а формат записи: “hh:mm:ss”. Если вы укажите аргумент точности, то получите возможность работать с долями секунд. |
| TIMESTAMP [(n)] | Комбинация типов данных DATE и TIME. |

Типы данных TIME и TIMESTAMP обладают еще одним аргументом – WITH TIME ZONE. Аргумент предназначен для определения зонального времени.

Тип данных **интервал** предназначен для задания интервала времени между двумя значениями времени. Основное назначение Interval – осуществление приращения (уменьшения) переменной типа DateTime. Структура типа данных Interval состоит из таких же полей, что и DateTime и, кроме того, обладает знаком, для того чтобы указать направление смещения времени.

## Язык определения данных – DDL

Язык определения данных решает задачи создания и удаления объектов базы данных. По стандартам SQL-92 к таким объектам относятся:

- базы данных (схемы);
- домены;
- таблицы;
- индексы;
- представления;
- курсоры.

Каждый объект в базе однозначно описывается его именем и имеет владельца. Подавляющее число операторов DDL начинается с ключевых слов CREATE (создать) или DROP (удалить).

### Базы данных (схемы)

В SQL под схемой понимается поименованная коллекция связанных между собой объектов базы данных. Определение схемы приводит к созданию собственно физической базы данных, так что с точки зрения SQL между понятиями «схема» и «база данных» можно поставить знак равенства.

Для управления БД в структурированном языке запросов SQL предусмотрено три базовые инструкции:

CRETATE DATABASE – создание нового экземпляра БД;

ALTER DATABASE – модификация существующего экземпляра БД;

DROP DATABASE – удаление БД.

Перечисленные инструкции в том или ином виде вы можете встретить практически во всех современных СУБД. В Informix Universal Server, MySQL, SQL Server, PostgreSQL используется инструкция CREATE DATABASE. Изначально в СУБД DB2 и Oracle схема данных формировалась по умолчанию во время развертывания сервера, и все создаваемые таблицы принадлежали одной схеме. Позднее Oracle отказался от идеи общей схемы и стал позволять создавать отдельные экземпляры БД, для этого в состав диалекта SQL этой СУБД была введена инструкция CREATE DATABASE.

Для получения полного представления о создании БД рассмотрим несколько примеров создания новой БД «WAREHOUSE» на различных диалектах SQL. Популярные СУБД MySQL и PostgreSQL удовлетворятся одной строкой.

**Листинг 8.1. Инструкция для создания БД в MySQL**

```
CREATE DATABASE 'WAREHOUSE';
```

При работе с СУБД InterBase (Firebird) допускается воспользоваться следующим примером (листинг8.2).

**Листинг 8.2. Инструкция для создания БД в InterBase**

```
CREATE DATABASE 'd:\data\warehouse.ib'
USER 'SYSDBA' PASSWORD 'masterkey';
```

В простейшем случае для SQL Server компании Microsoft также достаточно передать инструкцию CREATE DATABASE и имя создаваемой БД, но для создания новой базы с настройками файлов данных и журнала транзакций одной-двух строк кода явно недостаточно (листинг 8.3).

**Листинг 8.3. Инструкция для создания БД в SQL Server**

```
CREATE DATABASE WAREHOUSE
ON
   (NAME = warehouse_dat,
    FILENAME = 'c:\data\warehouse.mdf',
    SIZE = 50MB,
    MAXSIZE = 200MB,
    FILEGROWTH = 5MB)
LOG ON
   (NAME = warehouse_log,
    FILENAME = 'c:\data\warehouse.ldf',
    SIZE = 10MB,
    MAXSIZE = 50MB,
    FILEGROWTH = 5MB)
```

В этом примере мы определили не только параметры БД, но и журнала транзакций.

Несмотря на всеобщее признание команды CREATE DATABASE производителями СУБД, в стандарте SQL она в явном виде отсутствует, а вместо нее предусмотрена следующая синтаксическая конструкция:

```
CREATE SCHEMA <имя_схемы> [AUTHORIZATION имя владельца]
            [DEFAULT CHARTERSET набор символов по умолчанию]
            [PATH <символьный путь к файлам БД> ]
            [ <дополнительные инструкции>... ]
```

В минимальной нотации достаточно только указания названия будущей БД. При необходимости можно определить путь к файлам создаваемой схемы и задать дополнительные опции, связанные с вопросами авторизации и кодировкой символов (если последняя должна отличаться от назначенной по умолчанию).

```
CREATE SCHEMA 'WAREHOUSE' AUTORIZATION 'DBAdmin';
```

Процесс уничтожения схемы особым изыском не отличается. Практически во всех диалектах SQL имеется инструкция:

```
DROP DATABASE <имя_базы_данных>;
```

В свою очередь, стандарт SQL опирается на команду DROP SCHEMA

```
DROP SCHEMA <имя_схемы> [CASCADE|RESTRICT];
```

По умолчанию (ключевое слово RESTRICT) инструкция полагает, что удалению подлежит БД, не содержащая никаких объектов, и если это не так, то ее выполнение команды отменяется. Если следует обеспечить удаление схемы с каскадным уничтожением всех принадлежащих ей таблиц, представлений и других объектов, используем RESTRICT.

## Домены

Если вы видите, что для корректного описания допустимых значений, передаваемых в то или иное поле таблицы, возможностей типа данных явным образом не хватает, то это признак того, что пора воспользоваться услугами доменов. Если типизация отвечает за физический формат хранения данных, то в дополнение к этому домен позволяет накладывать дополнительные логические ограничения на значения, передаваемые в столбец таблицы. Поэтому механизм доменов существенно обогащает наши возможности по определению ограничений.

Для описания домена SQL предусматривает следующую конструкцию:

```
CREATE DOMAIN <имя_домена> [ AS ] <предопределенный тип данных>
[ <значение по умолчанию> ]
[ <ограничения домена>[,...] ]
[ <правила сравнения> ]
```

При определении доменных правил огромную пользу может принести грамотное применение инструкции CHECK – благодаря ей программист сможет описать ограничения, накладываемые на вводимые пользователем значения. Допустим, что нам следует создать домен, ограничивающий ввод значений диапазоном от 1 до 100 и по умолчанию принимающий значение 1, тогда следует написать строки, предложенные в листинге 8.4.

**Листинг 8.4. Домен-диапазон**

```
CREATE DOMAIN DOMAIN_SMALLRANGE AS INTEGER
DEFAULT 1
CHECK (VALUE BETWEEN 1 AND 100);
```

Созданный домен сохраняется в системном каталоге БД, поэтому его можно применять при определении атрибутов множества таблиц.

## Таблицы

После создания базы данных и доменов можно переходить к созданию основного объекта реляционной БД – таблиц. Рассмотрим средства подъязыка DDL, нацеленные на решение этой задачи:

```
CREATE [ {GLOBAL | LOCAL} TEMPORARY] TABLE имя_таблицы
  ({определение поля | [ограничение таблицы]}., …
  [ON COMMIT {DELETE | PRESERVE} ROWS]):
  определение поля ::= имя_поля {имя домена | тип данных [размер]}
                                [ограничение поля…]
[DEFAULT значение по умолчанию]
[COLLATE имя сравнения]
```

Минимальная конструкция, позволяющая создать простейшую таблицу, должна включать операторы CREATE TABLE, имя создаваемой таблицы и перечень столбцов с их типами и размерами. В результате мы получим пустую таблицу. В следующем примере (листинг 8.5) создается таблица PEOPLES, включающая пять столбцов. Первый столбец целочисленный, второй, третий и четвертый столбцы – символьные, и последний столбец предназначен для хранения даты.

**Листинг 8.5. Создание простейшей таблицы**

```
CREATE TABLE PEOPLES
(IndNum  int,
 Surname char(30),
 FName  char(20),
 LName  char(20),
 BDay   date);
```

В таблице должно быть одно или несколько определений столбцов. Определение столбца обязательно должно включать имя, тип данных и при необходимости размер. В различных диалектах SQL могут различаться названия типов столбцов и формат хранения данных (например, даты и времени). По умолчанию атрибуты таблицы допускают помещение в них значения NULL.

В соответствии со стандартом SQL для таблицы можно задать одно или несколько ограничений (табл. 8.1).

***Таблица 8.6.*** *Ограничения столбцов таблицы*

| **Ключ** | **Описание** |
|---|---|
| NOT NULL | Запрет на вставку в столбец неопределенного значения NULL |
| UNIQUE | Значение столбца должно быть уникальным |
| PRIMARY KEY | Признак первичного ключа. Значение поля должно быть уникальным, оно не может содержать NULL, в таблице это ограничение может использоваться только один раз |
| CHECK | Ограничение-проверка на допустимое значение. В скобках за оператором CHECK указывается предикат, проверяющий допустимость значения |
| FOREIGN KEY и REFERENCES | Оба ограничения весьма похожи, единственное различие между ними в том, что FOREIGN KEY – ограничение внешнего ключа для таблицы, а REFERENCES – ссылка на столбец со значениями подстановки. В случае установки ограничения для таблицы сразу за FOREIGN KEY в скобках необходимо указать перечень столбцов, относящихся к ключу. В остальном синтаксис одинаков. |

| | |
|---|---|
| | Ограничения внешнего ключа требуют, чтобы все значения, присутствующие во внешнем ключе, соответствовали значениям родительского ключа (обеспечение ссылочной целостности) |

В любой реляционной таблице должна существовать возможность уникальной идентификации отдельной строки, для этой цели предназначен первичный ключ. Для включения столбца в состав первичного ключа используется ключевое слово `PRIMARY KEY`. Продемонстрируем это на примере таблицы поставщиков `SUPPLIERS` (листинг 8.8).

**Листинг 8.6. Создание таблицы поставщиков**

```
CREATE TABLE SUPPLIERS (
  SUPPLIER_ID int NOT NULL AUTO_INCREMENT,
  SUPPLIER varchar(100) NOT NULL,
  PRIMARY KEY (SUPPLIER_ID),
  UNIQUE KEY SUPPLIER_UNIQUE (SUPPLIER)
);
```

В представленном примере на роль первичного ключа назначен столбец `SUPPLIER_ID`.

## Внешние ключи и связи между таблицами

В реляционных базах данных возникающее между таблицами отношение создается посредством ограничения внешнего ключа. Данное ограничение обеспечивает правило существования строк в подчиненной таблице, защищая ее от попыток вставить записи, несовместимые с выбранной моделью.

Изучите рис. 8.1, на котором представлены таблицы поставщиков `SUPPLIERS` и приходных накладных `DELIVERYNOTES`. Каким образом реализовать поддержку ссылочной целостности для таблицы `DELIVERYNOTE` на языке SQL?

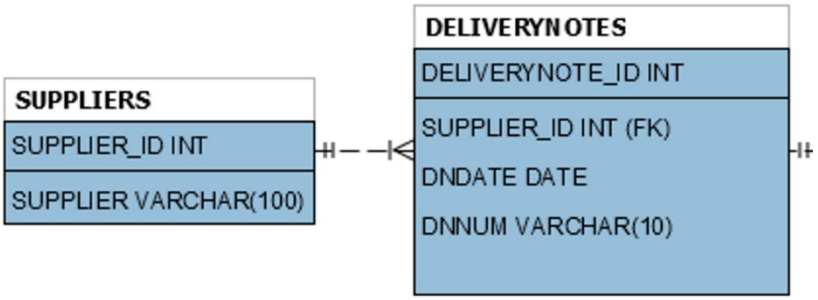



**Рис. 8.1.** Связь между таблицами

Минимальная конструкция определения внешнего ключа в большинстве диалектов SQL выглядит примерно так:

```
FOREIGN KEY (имя_столбца,…)
REFERENCES имя_таблицы (имя_столбца, ...)
[ON DELETE {CASCADE | SET NULL | NO ACTION | RESTRICT | SET DEFAULT}]
[ON UPDATE {CASCADE | SET NULL | NO ACTION | RESTRICT | SET DEFAULT}]
```

Первое и самое главное условие для построения связи между таблицами заключается в наличии однотипных столбцов первичного ключа в родительской таблице и внешнего ключа в дочерней. Раз в нашем примере в качестве первичного ключа во всех таблицах задействуется целочисленный тип, то столбцы внешнего ключа также должны быть целочисленными. В противном случае связь окажется создать невозможно.

Второе условие – столбцы первичного ключа главной таблицы и соответствующие им столбцы внешнего ключа подчиненной таблицы должны быть проиндексированы. В случае первичного ключа индексация производится автоматически, нам достаточно только поставить признак `PRIMARY KEY`. В случае внешнего ключа индекс создается в момент обработки инструкции `FOREIGN KEY`.

Затем в соответствии с требованиями минимальной нотации к описанию связи сразу после `FOREIGN KEY` следует указать имя столбца (или столбцов) внешнего ключа дочерней таблицы. Имя главной таблицы и название столбца первичного ключа следуют после ключевого слова `REFERENCES` (листинг 8.7).

**Листинг 8.7. Создание таблицы с внешним ключом**

```
CREATE TABLE DELIVERYNOTES (
  DELIVERYNOTE_ID int NOT NULL AUTO_INCREMENT,
  SUPPLIER_ID int NOT NULL,
  DNDATE date NOT NULL,
  DNNUM varchar(10) NOT NULL,
  PRIMARY KEY (DELIVERYNOTE_ID),
  FOREIGN KEY (SUPPLIER_ID) REFERENCES SUPPLIERS (SUPPLIER_ID)
);
```

Таким образом, за счет применения FOREIGN KEY и REFERENCES в инструкции создания таблицы накладных DELIVERYNOTES мы реализовали отношение 1:M (один поставщик – много накладных) между таблицами точно так, как предложено в ER-модели.
Расширенный синтаксис позволяет в момент описания связи настроить правила ссылочной целостности, которые определят поведение дочерней таблицы при изменении (ON UPDATE) или удалении (ON DELETE) связанных данных в родительской. В момент изменения значения первичного ключа и в момент удаления значения первичного ключа у родительской таблицы сервер осуществляет контроль ссылочной целостности данных. Если при создании внешнего ключа в инструкции SQL мы никак не конкретизировали поведение механизма контроля целостности, то они по умолчанию устанавливаются в режим RESTRICT, остальные доступные варианты работы триггеров отражены в табл. 8.7.

***Таблица 8.7.*** *Поведение дочерней таблицы при изменении/удалении значения PK*

| Действие | Описание |
|---|---|
| CASCADE | Изменение значения первичного ключа приводит к автоматическому изменению соответствующих значений внешнего ключа |
| SET NULL | При изменении значения или удалении первичного ключа все значения в связанных с внешним ключом колонках устанавливаются в NULL. В результате в дочерней таблице появляются «брошенные» строки, утерявшие связь с соответствующей записью из главной таблицы |
| RESTRICT | Режим по умолчанию. Внешний ключ воспринимает только значения первичного ключа или NULL. Попытки поместить в него какие-либо другие значения будут отвергаться |
| NO ACTION | То же самое, что и RESTRICT |

## Ограничения на значения столбцов

Установка значения по умолчанию производится при помощи оператора DEFAULT. Таким образом обеспечивается автоматический ввод в столбец некоторого значения, если его явным образом не укажет пользователь в момент вставки новых данных (листинг 8.8).

**Листинг 8.8. Создание таблицы с простейшими ограничениями**

```
CREATE TABLE TABLE1
(COLUMN1 INT DEFAULT 0,
 COLUMN2 FLOAT NOT NULL,
 COLUMN3 NUMERIC(9,2), CHECK (COLUMN3>-100 AND COLUMN3<100),
 COLUMN4 FLOAT NOT NULL, CHECK (COLUMN4<COLUMN2));
```

В этих же строках вы найдете оператор CHECK, позволяющий наложить определенные ограничения на вводимые в поле данные. Благодаря CHECK третий столбец – принимает только значения из диапазона от –100 до 100, а четвертый столбец таблицы должен содержать число, не превышающее значение, хранимое во втором столбце.

Листинг 8.9 демонстрирует еще один аспект применения значений по умолчанию в InterBase (FireBird).

**Листинг 8.9. Столбец со значением по умолчанию**

```
CREATE TABLE DELIVERYNOTES (
  DELIVERYNOTE_ID INTEGER PRIMARY KEY,
  …
  INSERTEDBY VARCHAR(50) DEFAULT CURRENT_USER)
```

В представленном примере таблица DELIVERYNOTES дополнена столбцом INSERTEDBY, в который по умолчанию будет передаваться содержимое контекстной переменной CURRENT_USER, в которой InterBase хранит имя текущего пользователя. Такое решение позволит нам контролировать, кто именно вносил в таблицу данные об очередном заказе.

> ***Внимание!***
> В большинстве СУБД DEFAULT гарантирует попадание назначенного разработчиком БД значения в заданный столбец только в одном случае – при осуществлении операции вставки новой записи (инструкция INSERT). Это произойдет в ситуации, когда пользователь не предложил взамен неопределенности NULL никакого значения. Если же запись уже существует и подвергается редактированию (инструкция UPDATE), то пользователь запросто сможет отправить в столбец NULL. Это замечание следует учитывать при проектировании таблиц и при необходимости строить еще один эшелон обороны от некорректности данных, например добавляя ограничение NOT NULL или описывая дополнительные правила на уровне триггеров.

# Индексы

Система индексирования используется во всех СУБД без исключения, но в стандартном SQL синтаксическая конструкция создания индексов отсутствует. Вместе с тем диалекты SQL большинства производителей весьма схожи и придерживаются примерно следующей упрощенной конструкции:

```
CREATE [UNIQUE] [ASCENDING|DESCENDING] INDEX имя_индекса
ON имя_таблицы (имя_столбца [, … n]);
```

В случае создания индекса с опцией UNIQUE БД прекращает допускать ввод повторяющихся значений в столбец (столбцы), включенный в определение индекса (листинг 8.10).

**Листинг 8.10. Индекс для одного столбца таблицы с проверкой уникальности**

```
CREATE UNIQUE INDEX ind_SUPLIER ON SUPPLIERS(SUPLIER);
```

Довольно часто приходится создавать составные индексы, то есть индексы, состоящие из двух и более столбцов таблицы (листинг 8.11).

**Листинг 8.11. Индекс для нескольких столбцов таблицы**

```
CREATE INDEX ind_VENDOR ON VENDORS (vendor, email);
```

В последнем примере индекс строится по столбцам с именем и электронным адресом производителя товаров.

Допускается создавать индексы, осуществляющие сортировку данных в обратной последовательности, например в SQL Server, InterBase, FireBird это делается при помощи ключа DESCENDING (листинг 8.12).

**Листинг 8.12. Индекс по убыванию значений для SQL Server**

```
CREATE DESCENDING INDEX ind_SUPPLIER_DESC ON SUPPLIERS(SUPPLIER);
```

В MySQL аналогичная задача решается немного по-другому (листинг 8.23).

**Листинг 8.13. Индекс по убыванию значений для MySQL**

```
CREATE INDEX ind_SUPPLIER_DESC ON SUPPLIERS(SUPPLIER DESC);
```

# Изменение индекса

Перенастройка индекса производится инструкцией ALTER INDEX. Особенности команды зависят от эксплуатируемой СУБД, например InterBase, FireBird, SQL Server, Oracle умеют отключать индекс (листинги 8.14 и 8.15) .

**Листинг 8.14. Отключение и включение индекса в InterBase**

```
ALTER INDEX ind_VENDOR INACTIVE;
/* блок операций с данными */
ALTER INDEX ind_VENDOR ACTIVE;
```

**Листинг 8.15. Отключение индекса в SQL Server**

```
ALTER INDEX ind_VENDOR ON VENDORS DISABLE;
```

Такая операция может пригодиться, когда в короткий срок времени большей части данных, хранящихся в таблице `VENDORS`, грозит удаление или изменение. Мы временно отключаем индекс, чтобы приостановить процесс индексирования и не перегружать сервер баз данных. Если вы предположили, что для включения индекса в SQL Server потребуется команда «Enable», – вы ошиблись, в Transact-SQL для этой задачи применяется инструкция `ALTER INDEX REBUILD`. Однако если вы работаете в Oracle, то «Enable» – как раз то что надо. В InterBase включением/отключением ведают ключевые слова `ACTIVE/INACTIVE`.

После подключения индекса СУБД самостоятельно реорганизует его и приведет в соответствие с текущими данными проиндексированных столбцов таблицы.

> ***Внимание!***
> Если вы администрируете большую, активно эксплуатируемую многопользовательскую базу данных, то периодически стоит заставлять СУБД перестроить все индексы. Подобные операции следует проводить в период наименьшего использования БД (в выходные и праздничные дни или в ночное время), предварительно не забывая создать полную копию БД и журналов транзакций.

# Удаление индекса

Синтаксис удаления индекса включает в себя ключевые слова `DROP INDEX` и непосредственно имя таблицы и индекса, разделенные точкой.

```
DROP INDEX имя_индекса;
```

В некоторых диалектах SQL (листинг 8.16), кроме имени индекса, следует указать имя таблицы, которой он принадлежит.

**Листинг 8.16. Удаление индекса в MySQL**

```
DROP INDEX ind_SUPPLIERS_SUPPLIER ON SUPPLIERS;
```

## Задание

Используя изученные на занятии операторы SQL, подготовьте последовательность инструкций SQL (скрипт SQL) позволяющих:

1. Создать все таблицы и связи между ними в вашей БД (см. задание ЛР № 4).
2. Предложите инструкции SQL, позволяющие реструктурировать имеющиеся таблицы (добавлять или удалять поля, изменять характеристики полей).
3. Предложите инструкции позволяющие создать все необходимые индексы для вашей БД

# 9. Разработка отчетов БД в среде Access

*Вид занятия – лабораторное занятие*
*Время занятия – 2 часа*
*Программное обеспечение – среда Microsoft Access*

Отчеты (reports) предоставляют наиболее гибкий и одновременно наглядный способ отображения информации, содержащейся в базе данных. Отчеты позволяют выбрать из базы данных требуемую пользователем информацию и оформить ее в виде документов, которые можно просмотреть и напечатать.

Средствами отчета невозможно внести изменения в содержащиеся в БД данные.

## Подготовка отчета в среде Access

В среде Access отчет может создаваться как на основе таблиц, так и на базе заранее подготовленных запросов. Допустим, что нам требуется подготовить отчет по списочному составу студентов нашего ВУЗа. Разработаем простой объединяющий запрос (листинг **9.1**).

**Листинг 9.1. Запрос к БД Access**

```
SELECT Students.*, Spec.*
FROM Spec
INNER JOIN Students ON Spec.Spec_ID = Students.Spec_ID
ORDER BY SName, FName;
```

Сохраните вновь созданный запрос под именем “StudentsList”, в списке объектов выделите только что созданный элемент и щелкните по кнопке “**Создание — Отчет**” на панели управления базой данных. В результате перейдя в окно предварительного просмотра вы получите простейший вариант бумажного отчета (рис. 9.1).

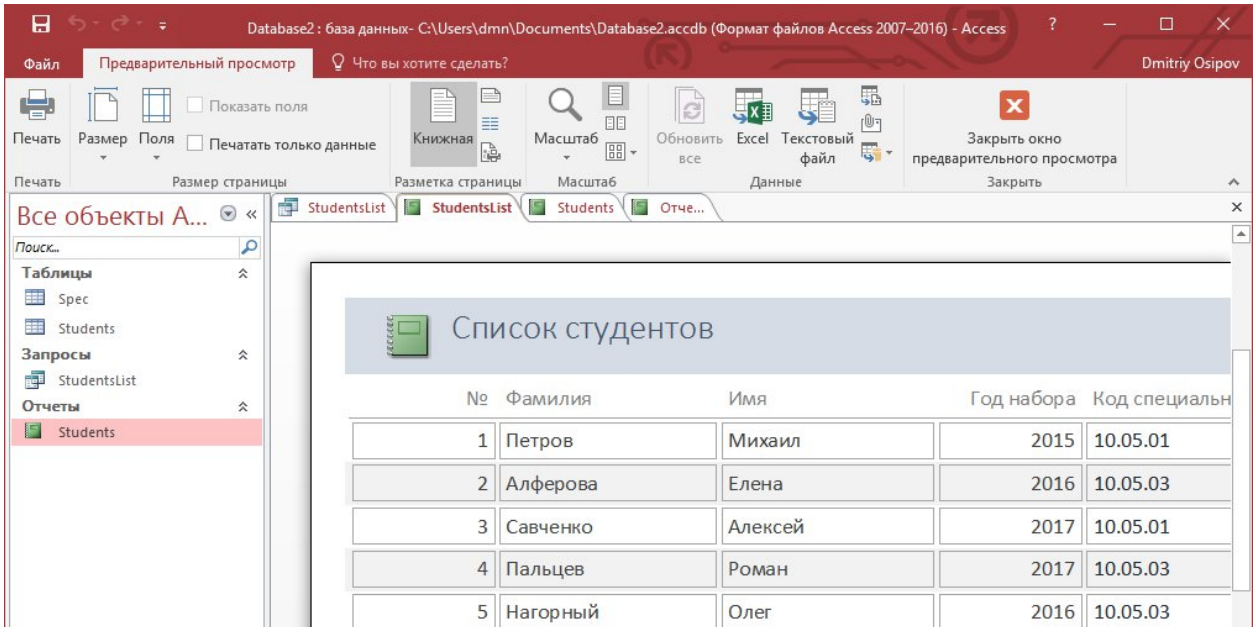



*Рис. 9.1. Отчет созданный Access в режиме по умолчанию*

Другим вариантом создания отчетов в Access является использование специального помощника — мастера отчетов. Для доступа к нему воспользуйтесь кнопкой “**Создание — Мастер отчетов**” (рис. 9.2)

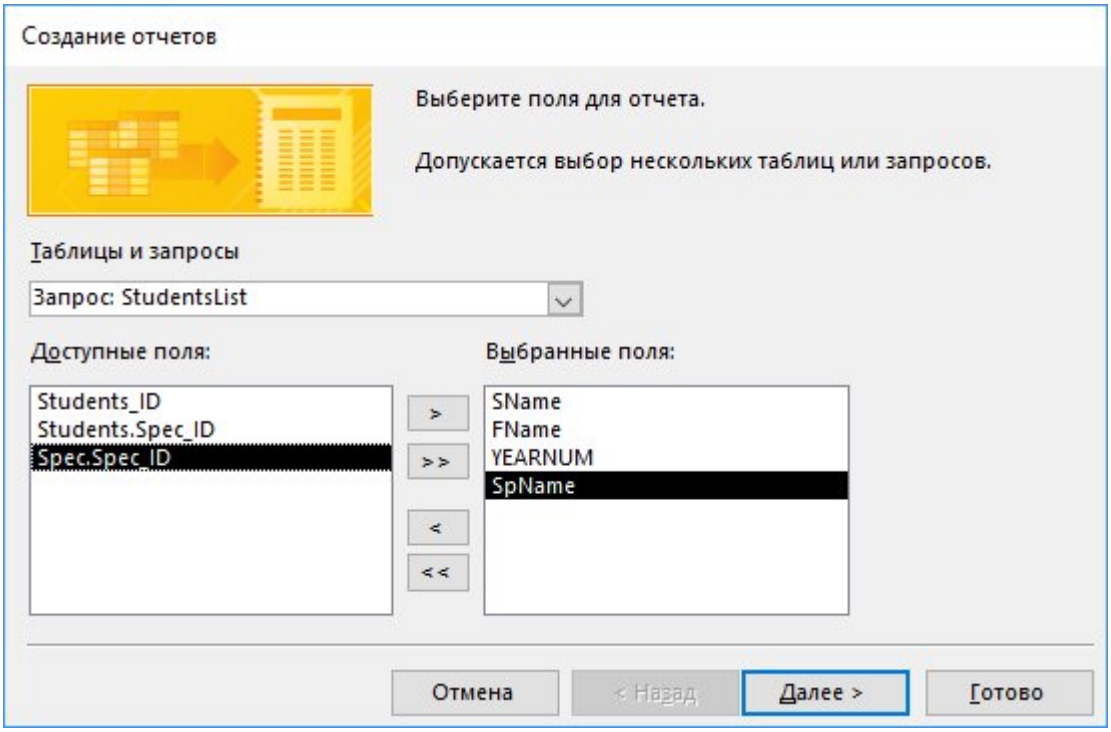



*Рис. 9.2. Первое окно диалога мастера отчетов*

Для создания профессионального отчета вам потребуется значительно больше усилий. Для этого потребуется поработать вручную в **Конструкторе отчетов**. Отчет сразу “рассыпается” на пару десятков самостоятельных элементов управления. Здесь и обычные компоненты-метки (предназначенные для вывода вспомогательных надписей) и компоненты-поля (отвечающие за получение информации из результирующего набора данных).

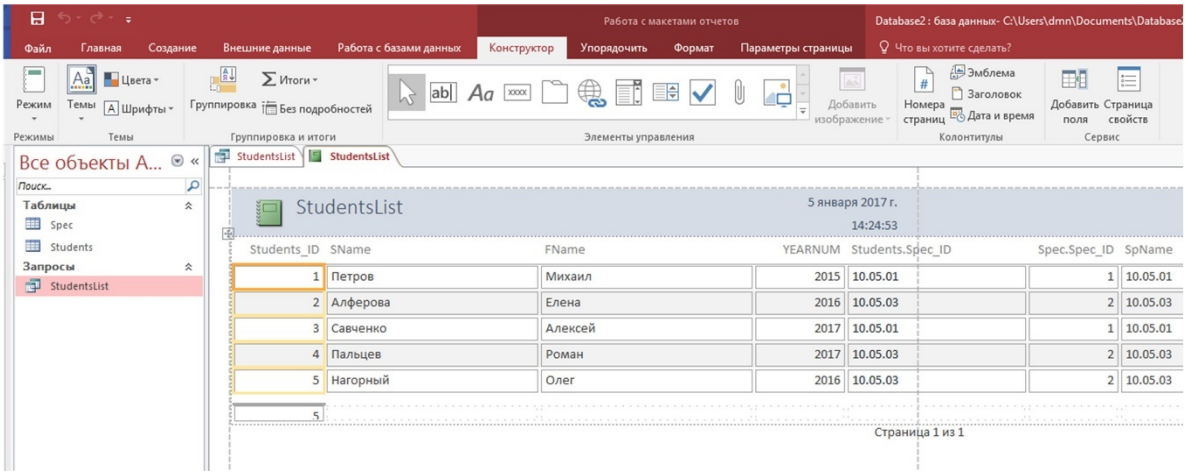



*Рис. 9.3. Отчет в режиме конструктора*

Для настройки характеристик отдельного элемента выведите на экран его свойства (пункт “Свойства” контекстного меню). В рамках редактора свойств вы сможете управлять всеми параметрами элемента управления.

## Задание

В соответствии с полученными заданиями подготовьте 3-4 отчета. Основной отчет должен объединять в себе информацию из всех таблиц БД. Один из отчетов должен быть создан с помощью мастера отчетов Access.

**Важно**! Подготовленные вами отчеты должны быть готовы к печати на листах бумаги формата А4.

# 10. Разработка интерфейса приложения в среде Access

*Вид занятия – лабораторное занятие*
*Время занятия – 2 часа*
*Программное обеспечение – среда Microsoft Access*

В задачи разработчика баз данных входит не только создание ключевых объектов БД (например, таблиц и запросов), но и разработка интуитивно понятного интерфейса, позволяющего обычному пользователю (не имеющего ни малейшего представления об особенностях физической организации данных) просматривать, добавлять, редактировать и удалять данные.

## Создание форм для ввода данных в Microsoft Access

Среди всех сред программирования, специализирующихся на работе с БД Microsoft Access предоставляет едва ли ни самый простой и удобный способ создания интерфейса доступа к базам данных. Здесь предусмотрено несколько способов создания форм. К простейшим из них относится применение средств автоматического создания форм на основе таблицы или запроса (рис. 10.1).

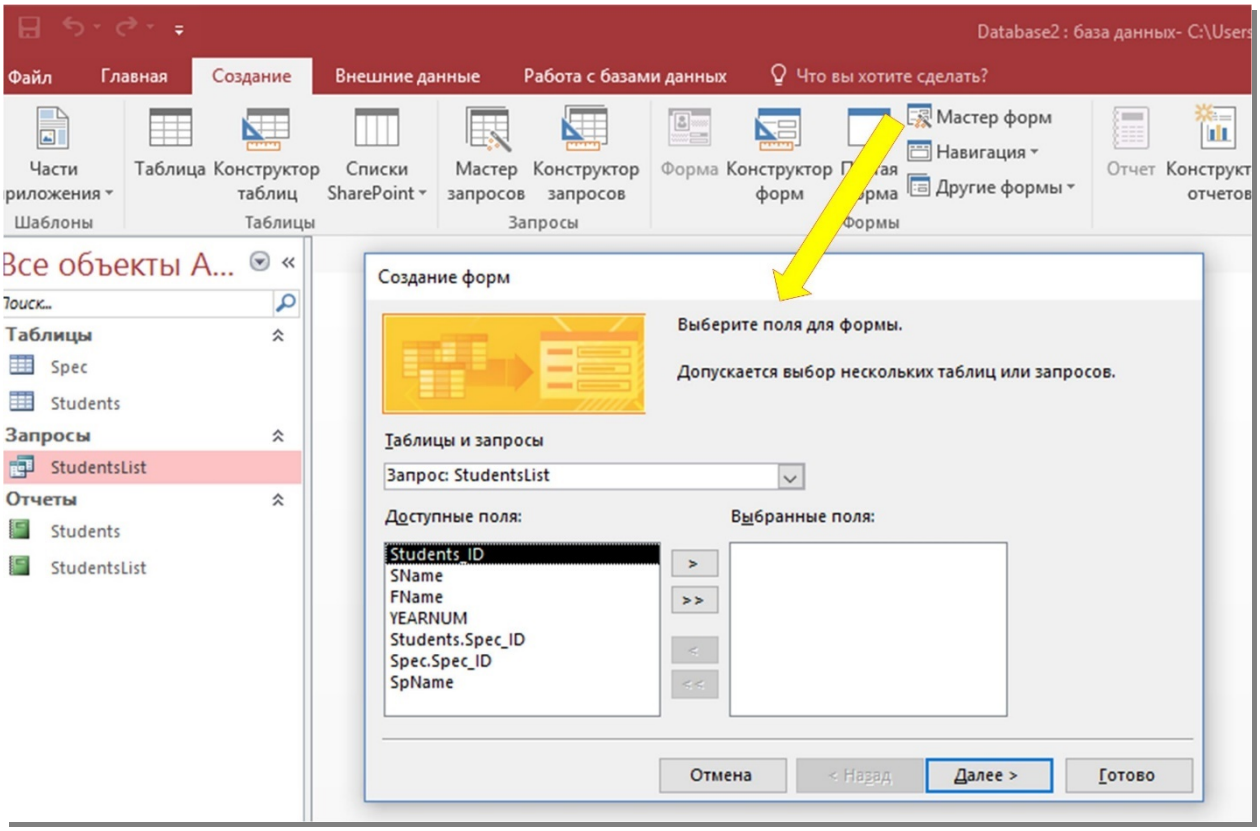



*Рис. 10.1. Мастер создания новой формы*

Вернемся к нашему проекту по учету студентов ВУЗа (лабораторные работы 2 и 3) и создадим форму для таблицы студентов “Students”.

1. Выберите таблицу “ Students ” в списке объектов БД.
2. Перейдите на страницу “Создание” среды проектирования.
3. Раскройте список **Другие формы** и выберите в нем элемент **Разделенная форма**.

Автоматически созданная форма далеко не безукоризненна, с точки зрения конечного пользователя у нее как минимум четыре недостатка (рис. 10.3). **Во-первых**, подписи полей не отражают их назначения. **Во-вторых**, на форме присутствует поле первичного ключа, причем создается впечатление, что оно доступно для редактирования, а в этом нет необходимости. **В-третьих**, размеры строк ввода чрезмерны велики. **В-четвертых**, название формы не отражает ее назначения.

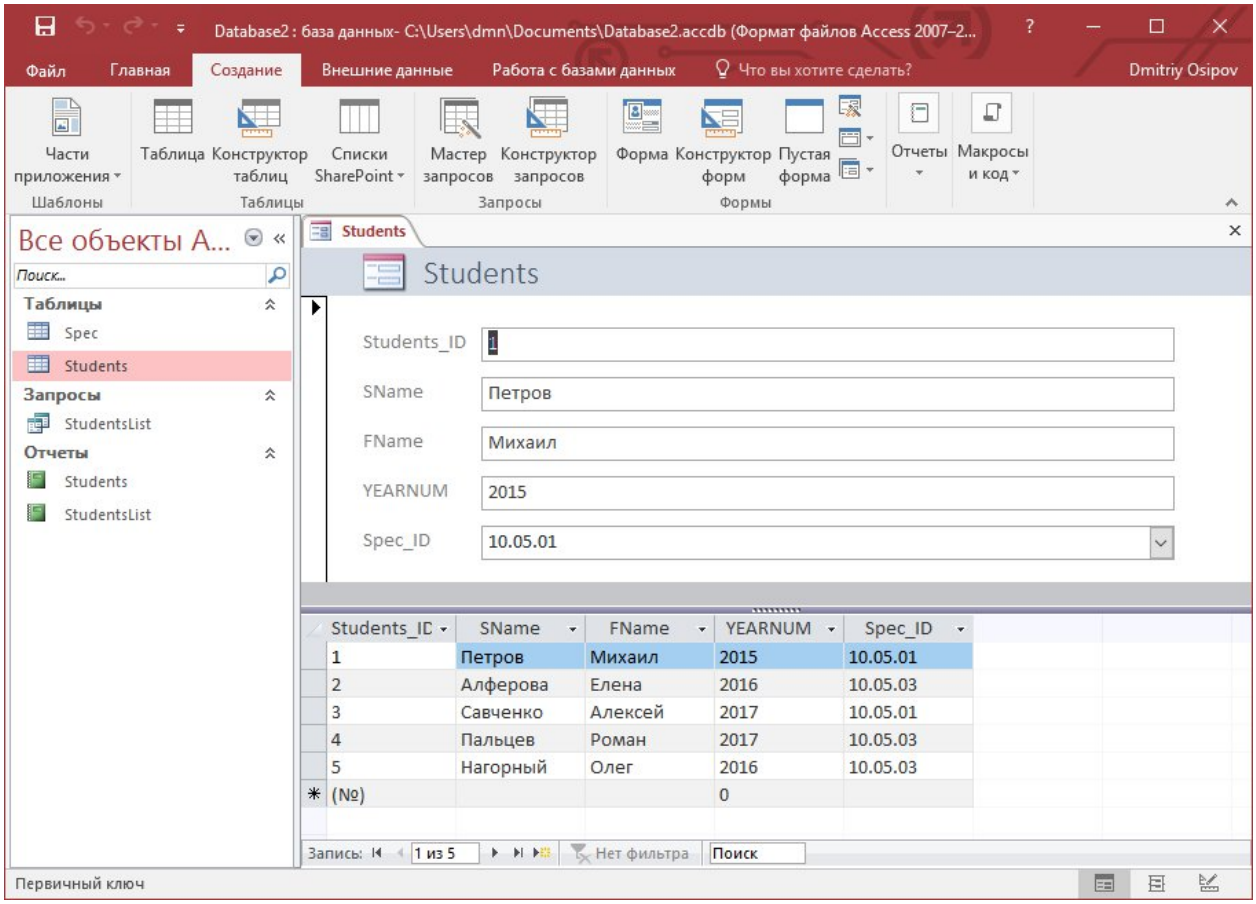



*Рис. 10.2. Автоматически созданная форма Students*

Для внесения исправлений и доработок в форму необходимо перейти в режим конструктора. Для этого перейдите на вкладку **Главная**, нажмите на кнопку “Режим” на панели инструментов, и в открывшемся списке выберите элемент “Конструктор”. В режиме конструктора вы сможете управлять всеми элементами управления, расположенными на форме (рис. 10.4). Для этого достаточно обратиться к контекстному меню компонента и выбрать пункт “Свойства”.

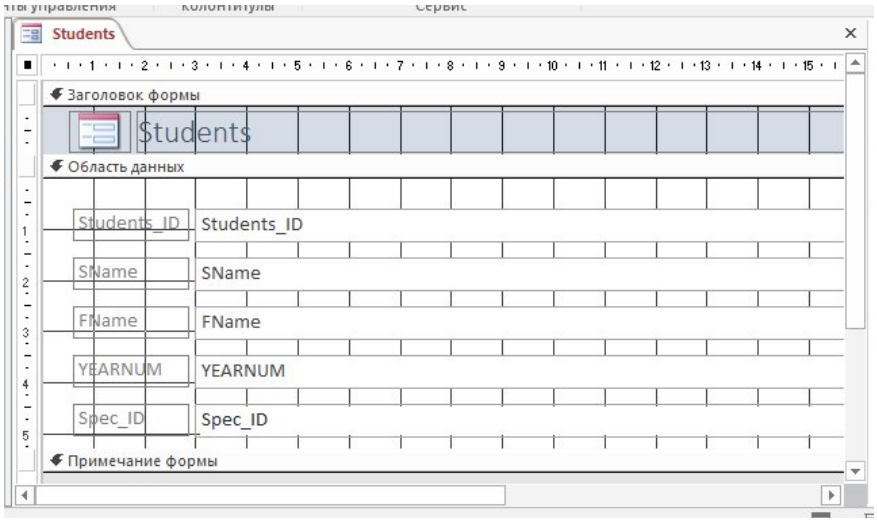



*Рис. 10.3. Форма в режиме конструктора*

## Рекомендации по проектированию пользовательского интерфейса

При разработке пользовательского интерфейса следует руководствоваться рядом рекомендаций. Ниже перечислены основные из них:

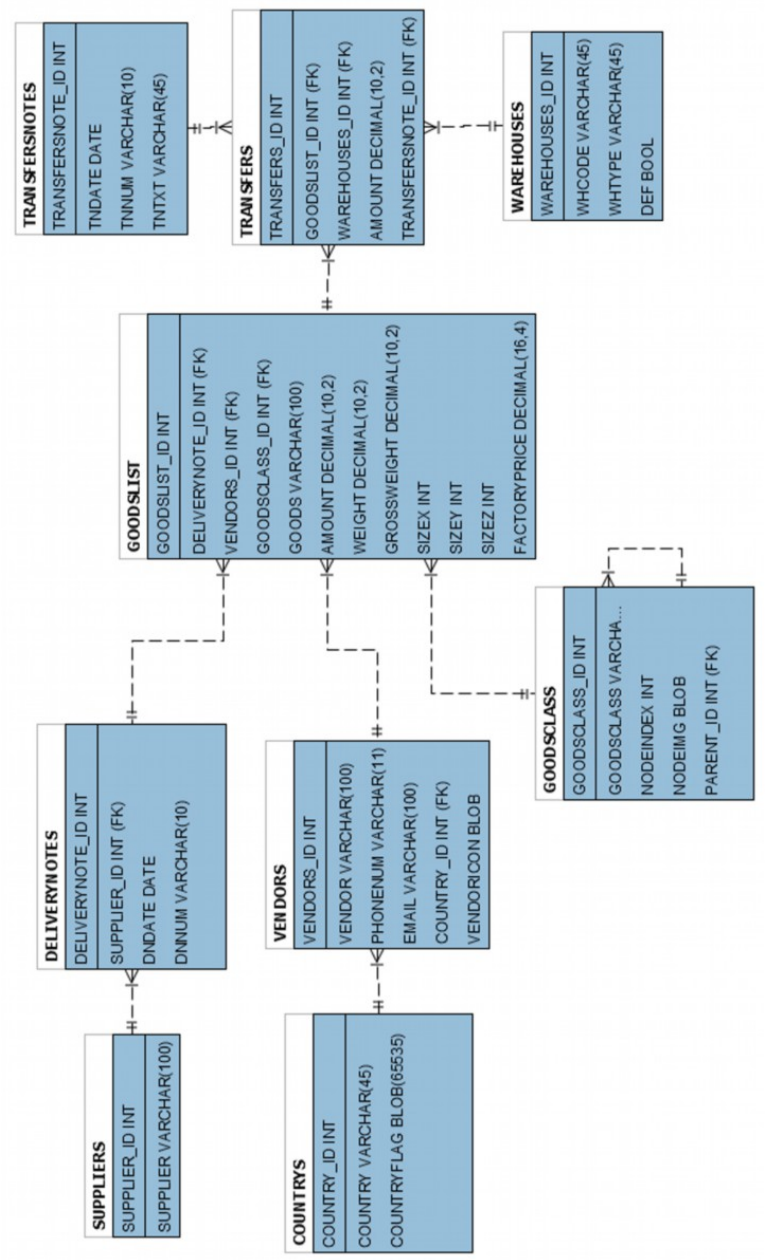
- название формы должно отражать ее назначение;
- последовательность полей и их группировка должны быть логически обоснованы;

- названия полей легко узнаваемы;
- форма должна иметь привлекательный вид;
- используется только общепринятая терминология и сокращения;
- визуальное выделение пространства и границ полей ввода данных;
- удобные средства перемещения курсора;
- наличие средств исправления отдельных ошибочных символов и целых полей;
- наличие сообщений об ошибках при вводе недопустимых значений;
- выделение обязательных для ввода данных полей;
- средства вывода пояснительных сообщений с описанием полей;
- средства вывода сообщения об окончании заполнения формы.

## Задание

Применяя различные подходы к проектированию рабочих форм создайте не менее 5 форм для ввода, редактирования и удаления данных для своего проекта БД. Разберитесь с назначением основных элементов управления, которые разрешено располагать на форме. При проектировании интерфейса форм опирайтесь на рекомендации по разработке пользовательского интерфейса БД.

ПРИЛОЖЕНИЕ 1. Модель БД «Склад»



**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**
**Федеральное государственное автономное образовательное учреждение высшего образования**
**«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

# ТЕХНОЛОГИИ ПРОЕКТИРОВАНИЯ И АДМИНИСТРИРОВАНИЯ БАЗ ДАННЫХ
# УЧЕБНОЕ ПОСОБИЕ ДЛЯ ПРОВЕДЕНИЯ ПРАКТИЧЕСКИХ ЗАНЯТИЙ

Направление подготовки 10.05.01 «Компьютерная безопасность»
Направленность (профиль): «Информационно-аналитическая и техническая экспертиза компьютерных систем»
Квалификация выпускника – специалист
Год начала обучения - 2026 г.
Форма обучения - очная

Ставрополь
2026

Печатается по решению
Учебно-методического совета
Северо-Кавказского федерального
университета

Осипов Д.Л., Березницкий А.С. Технологии проектирования и администрирования баз данных**:** учебное пособие. - Ставрополь: Изд-во СКФУ, 2025. – 60 с.

Учебное пособие представляет практикум по дисциплине «Технологии проектирования и администрирования баз данных». Оно составлены в соответствии с программой дисциплины и предназначено для специалистов 10.05.01 «Компьютерная безопасность».

В лабораторном практикуме приведены основные алгоритмы и структуры, применяемые в программировании, а также рассмотрены современные методы и технологии, используемые при разработке баз данных. Структурно каждая из работ, в предлагаемом учебном пособии, состоит из краткого изложения теоретического материала, необходимого для ее выполнения, указаний по порядку выполнения работы, заданий для самостоятельного выполнения и контрольных вопросов.

©ФГАОУ ВО «Северо-Кавказский
федеральный университет», 2026

# Содержание

# 1. Технология ADO

*Вид занятия – практическое занятие*
*Время занятия – 2 часа*
*Программное обеспечение – среда Microsoft Access, C++ Builder (Delphi)*

Технология *объектов данных ActiveX* (*Microsoft® ActiveX® Data Objects*, ADO) появилась на свет в стенах корпорации Microsoft и считается логическим развитием системы доступа к данным OLE DB. У ADO много достоинств, основное из них в том, что технология обеспечивает единый механизм доступа к разнотипным хранилищам данных. Приложение ADO обращается не к конкретной СУБД, а к объекту COM. Многокомпонентная модель объектов (Component Object Model, COM) описывает способ взаимодействия программ любого типа, программа-сервер предоставляет в распоряжение окружающих собственные службы, программа-клиент пользуется услугами доступных служб.

Для доступа к данным в рамках ADO (рис. 11.1) реализована коллекция интерфейсов COM-объектов предназначенных для работы, как с реляционными, так и с не реляционными наборами данных, включая майнфреймы, иерархические базы данных, текстовые, графические, географические данные, XML, e-mail и многое другое. Доступ к данным предоставляют драйверы, называемые *провайдерами*.

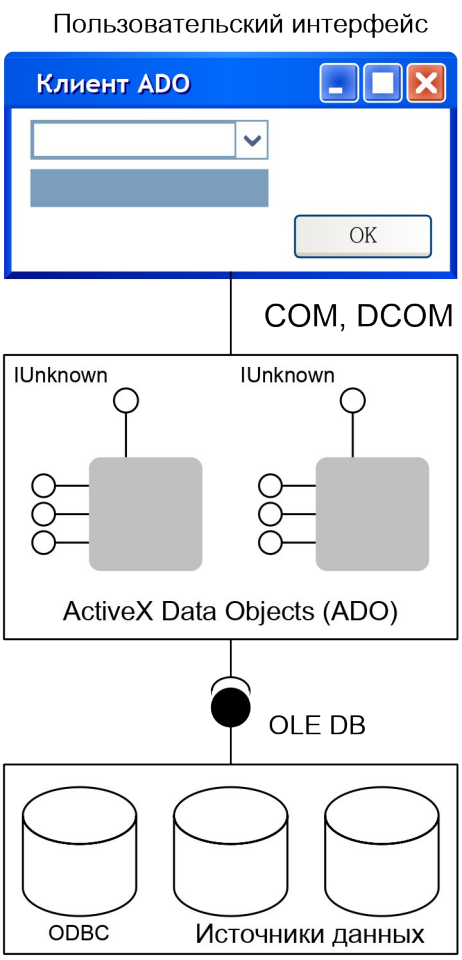



***Рис. 1.1.*** *Архитектура универсального доступа к данным ADO*

Все элементы управления VCL, нацеленные на работу с объектами данных ADO, собраны на странице dbGo палитры компонентов Delphi XE. Основной программный код всех интересующих нас сегодня классов сосредоточен в модуле ADODB, а описания их интерфейсов в `ADOInt`. Семерку компонентов ADO можно условно разделить на три категории (рис. 11.2):

- Компоненты обеспечения централизованного доступа к данным – `TADOConnection` и `TRDSConnection`. Эти компоненты обычно используются при разработке сложных приложений, базирующихся на архитектуре клиент-сервер. Экземпляр класса `TADOConnection` отвечает за установку соединения и взаимодействие проекта с БД, все остальные компоненты dbGo пользуются его посредническими услугами.
- Компоненты обслуживания данных в стиле Microsoft ADO – командный компонент `TADOCommand` и набор данных `TADODataSet`.
- Компоненты, работающие с данными: компонент-таблица `TADOTable`, запрос `TADOQuery` и хранимая процедура `TADOStoredProc`.

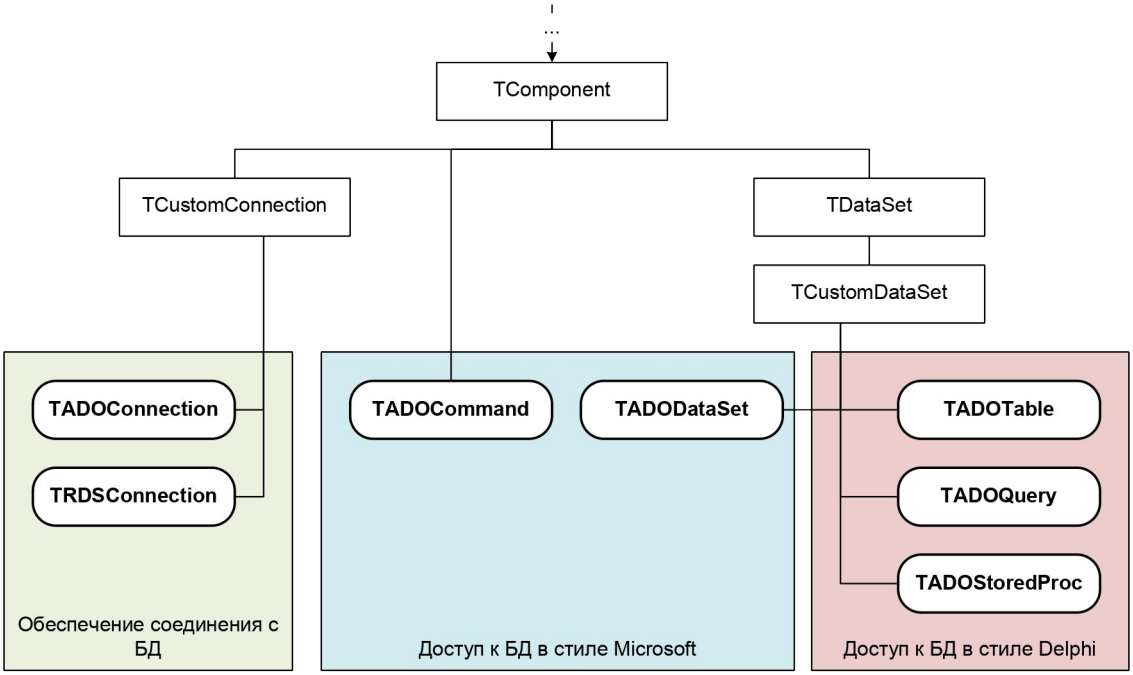



*Рис. 1.2. Иерархия компонентов ADO в библиотеке VCL*

## Строка соединения ADO

Строка соединения ADO представляет собой обычную текстовую строку, содержащую описание параметров соединения с интересующей нас БД. Соединительная строка представляет собой последовательность аргументов и значений, разделяемых точкой с запятой. В качестве символа присвоения применяется знак равенства.

```
Параметр1=значение1;Параметр2=значение2;
```

Например, так выглядит строка доступа к базе данных DEMODB развернутой на Microsoft SQL Server с именем MSSQL_HOST:

```
Provider=SQLOLEDB.1;Persist Security Info=False;User ID=SA;Initial Catalog=DEMODB;Data Source=MSSQL_HOST
```

Перечень наиболее часто встречаемых в соединительной строке аргументов представлен в таблице 1.1.

*Таблица 1.1. Аргументы соединительной строки*

| ***Параметр*** | ***Описание*** |
|---|---|
| `Provider` | Имя поставщика интерфейса OLE DB. Если этот аргумент не определен, то по умолчанию задействуется провайдер Microsoft OLE DB для ODBC – “MSDASQL” |
| `Persist Security Info` | Принимает значение true или false. В состоянии true уведомляет, что в строке соединения имеется конфиденциальная информация о порядке соединения (например, имя пользователя и пароль доступа), состояние false говорит о том, что такая информации отсутствует. |
| `File name` | Название файла, содержащего информацию о соединении |
| `User ID` | Регистрационное имя пользователя по умолчанию |
| `Password` | Пароль пользователя |
| `Data Source` | Имя источника данных, например SQL-сервера |
| `Initial Catalog` | Каталог инициализации – имя базы данных |
| `Remote Provider` | Имя провайдера применяемого на клиентской стороне соединения. Этот параметр необходим только при организации работы со службами удаленных данных RDS |
| `Remote Server` | Имя удаленного сервера. Этот параметр необходим только при работе с RDS |
| `URL` | Абсолютный URL адрес |

Строку соединения допускается хранить в специальном файле связи с данными (UDL-файле). Для создания такого файла достаточно создать пустой текстовый файл и заменить расширение txt на udl. Запуск такого файла автоматически вызовет редактор настройки соединительной строки.

## Соединение с хранилищем данных, компонент TADOConnection

Хотя основной задачей компонента считается установка соединения с базой данных ADO, но, ко всему прочему, `TADOConnection` умеет:

- регистрировать пользователя в СУБД;
- управлять транзакциями;
- отправлять в адрес СУБД различные команды (инструкции SQL);
- взаимодействовать с подчиненными компонентами – наборами данных ADO;
- получать от БД информацию об ее объектах и особенностях соединения.

### Установка соединения

Для описания параметров подключения к БД текст соединительной строки направляется в свойство

```
property ConnectionString: WideString;
```

Если в строке отсутствует информация о поставщике услуг, то его необходимо определить в свойстве:

```
property Provider: WideString;
```

По умолчанию роль провайдера исполняет MSDASQL.

Процесс соединения с указанной в соединительной строке базой данных инициируется посредством обращения к методу Open(). В простейшей ситуации процедура даже не требует параметров.

```
procedure Open; overload;
```

В перегружаемой версии процедуры `Open()` предусмотрена возможность передачи в адрес сервера имени и пароля пользователя. Но это только для случая, когда эта информация не включена в соединительную строку.

```
procedure Open(const UserID: WideString;
               const Password: WideString); overload;
```

Если вы полагаете, что при подключении к базе данных пользователь обязан пройти регистрацию и своими собственными руками ввести имя и пароль, то целесообразно установить в `True` свойство:

```
property LoginPrompt: Boolean;
```

В результате перед соединением с базой вызывается стандартное диалоговое окно регистрации.

Для проверки факта подключения компонента TADOConnection к БД контролируют состояние свойства:

```
property Connected: Boolean;
```

С процессом установки соединения связана строгая последовательность событий (табл. 11.2) позволяющая программисту практически на всех этапах помогать или мешать приложению получать доступ к БД.

*Таблица 1.2. Последовательность событий при соединении TADOConnection с набором данных*

| *Событие* | | *Описание* |
|---|---|---|
| 1. | `property BeforeConnect: TNotifyEvent;` | вызывается перед началом соединения |
| 2. | `property OnWillConnect: TWillConnectEvent;` | Генерируется после отправки запроса на соединение |
| 3. | `property OnLogin: TLoginEvent;` | Регистрация пользователя |
| 4. | `property OnConnectComplete : TConnectErrorEvent;` | Процесс соединения завешается |
| 5. | `property OnInfoMessage: TInfoMessageEvent;` | вызывается после успешного соединения, в тот момент, когда провайдер возвращает дополнительную информацию о соединении |
| 6. | `property AfterConnect: TNotifyEvent;` | вызывается по окончанию всего процесса соединения |

## Пример соединения без регистрации пользователя

Благодаря тому, что параметры соединения ADO содержатся в обычной текстовой строке, программист получает отличную возможность управлять процессом подключения к любому хранилищу данных просто редактируя параметры соединительной строки. Предлагаю вашему вниманию пример, позволяющий пользователю выбирать расположение исходного файл с данными в момент старта приложения настольной БД.

Допустим, что в нашем распоряжении имеется база данных в формате Microsoft Access. В составе форм проекта Delphi пока имеется:

- главная форма проекта `Form1` (модуль Unit1), данная форма пока не содержит элементов управления — вы их добавите позднее;
- форма ввода пути к файлу БД `Form2` (модуль Unit2);
- модуль данных `DataModule1:TDataModule` (модуль Unit3) на котором размещен компонент `ADOConnection1 :TADOConnection`.

Вполне очевидно, что для выбора файла БД в момент старта приложения следует выводить окно, уточняющее путь к файлу. Для воспользуемся формой `Form2` (рис. 11.3) и размещаем на ее поверхности следующий перечень компонентов:

- строка ввода `Edit1:TEdit` для редактирования пути к файлу.
- диалог открытия файла `OpenDialog1`. Для того, чтобы компонент реагировал только на файлы Access следует отредактировать отвечающее за фильтрацию файлов свойство `Filter:= Моя БД Access 2003|demo_db.mdb|Моя БД Access 2007-2016|demo_db.accdb|Все файлы Microsoft Access|*.mdb; *.accdb`. Обратите внимание, на то, что в данном примере предполагается подключение к файлу с именем `demo_db`, в вашем случае следует передать другое имя.
- справа от строки ввода расположите кнопку `Button1:TButton`. Щелчок по этой кнопке будет пробуждать от сна диалог открытия файла `OpenDialog1`.
- Два последних компонента – кнопки класса `TBitBtn`. Первую кнопку называем `btnOK` и ее свойство `Kind` установим в `bkOK`. Свойству `Kind` второй кнопки присвоим `bkCancel`.

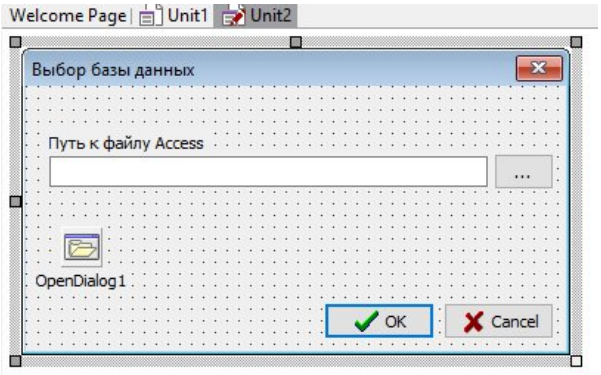



*Рис. 1.3. Форма ввода пути к БД*

Убедитесь, что новый проект сохранен и перейдите к `Form2`. Опишем событие щелчка по кнопке `Button1` — он отвечает за выбор пути к файлу БД (листинг 1.1).

**Листинг 1.1. Щелчок по кнопке — определение пути к файлу БД**

```
procedure TForm2.Button1Click(Sender: TObject);
begin
if OpenDialog1.Execute then
     LabeledEdit1.Text:=OpenDialog1.FileName;
end;
```

В рамках события `OnClick()` мы выводим на экран диалог открытия файла (в котором пользователь выберет имя файла с БД) и передаем имя и путь к файлу в строку `edFilePath`.

Переходим к главной форме проекта `Form1` и обращаемся к ее событию `OnShow()` — оно генерируется в момент вывода формы на экран (листинг 1.2).

**Листинг 1.2. Вывод главной формы проекта на экран**

```
uses Unit2, Unit3;

procedure TForm1.FormShow(Sender: TObject);
begin
if NOT DataModule1.ADOConnection1.Connected then
  if Form2.ShowModal=mrOK then
  begin
     {формируем соединительную строку}
      DataModule1.ADOConnection1.ConnectionString:=
                     'Provider=Microsoft.ACE.OLEDB.12.0;Data Source='+
                      Form2.LabeledEdit1.Text+';Persist Security Info=False';
     DataModule1.ADOConnection1.LoginPrompt:=False;
     DataModule1.ADOConnection1.Connected:=True;
  end;
end;
```

В результате мы получим следующий результат — в момент вывода на экран главной формы проекта над ним будет возникать модальное окно, в котором пользователь должен указать путь к целевому файлу.

## Информирование о БД

В состав методов компонента `TADOConnection` входит процедура способная предоставить нам системную информацию о базе данных

```
procedure OpenSchema(const Schema: TSchemaInfo;
                     const Restrictions: OleVariant;
                     const SchemaID: OleVariant; DataSet: TADODataSet);
```

Первый параметр метода `Schema: TSchemaInfo` определяет какую именно информацию мы рассчитываем получить. Например, передав в параметр константу `siIndexes` – мы приобретем информацию об индексах БД, `siTables` – таблицах, `siViews` – представлениях. Второй параметр `Restrictions` представляет собой массив ограничений на запрашиваемую информацию. Третий по счету параметр `SchemaID` применяется только в ситуации, когда глобальный уникальный идентификатор GUID провайдера данных не определен в стандартной спецификации OLE DB. Как правило, такая необходимость возникает в случаях, когда аргумент `Schema` установлен в `siProviderSpecific`. И, наконец, последний параметр `DataSet` является ссылкой на элемент управления класса `TADODataSet`, в его множество записей `Recordset` будут направлены результаты выполнения процедуры. Полученные данные программист может обрабатывать по своему усмотрению, например, просто отобразить их в сетке `TDBGrid`.

Апробируем приобретенные знания на практике. Разместите на поверхности модуля данных `DataModule1` проекта компоненты:

- соединение ADO – `ADOConnection1`;
- набор данных ADO – `ADODataSet1`;
- источник данных – `DataSource1`.

Перейдите к инспектору объектов. С помощью свойства `Connection` элемента управления `ADODataSet1` свяжите компонент с `ADOConnection1`. Затем выберите источник данных `DataSource1` и, воспользовавшись свойством `DataSet`, соедините его с `ADODataSet1`.

Создайте новую пустую форму `Form4` (Unit4) и расположите на ее поверхности единственный компонент — сетку – `DBGrid1`. Подключите в строку `uses` новой формы ссылку на модуль данных (в нашем примере это Unit3), затем присоедините сетку `DBGrid1` к источнику данных `DataSource1`.

Переходим к программированию (листинг 1.2).

**Листинг 1.3. Получение сведений о всех таблицах БД**

```
uses Unit3, Data.Win.ADODB;
procedure TForm4.FormShow(Sender: TObject);
begin
if DataModule1.ADOConnection1.Connected then
begin
DataModule1.ADODataSet1.Close;
DataModule1.ADOConnection1.OpenSchema(siTables, EmptyParam, EmptyParam,
DataModule1.ADODataSet1);
end;
end;
```

Если все сделано безошибочно, то в результате запуска проекта мы получим данные обо всех отношениях БД (в случае Access будут отображены не только созданные вами таблицы, но и скрытые таблицы и запросы).

На основе процедуры `OpenSchema()` построен ряд справочных методов, способных снабдить нас данными о таблицах, полях таблиц и хранимых процедурах базы данных.

```
procedure GetTableNames(List: TStrings; SystemTables: Boolean = False);
procedure GetFieldNames(const TableName: String; List: TStrings);
procedure GetProcedureNames(List: TStrings);
```

Результаты выполнения процедур передается в список строк `List: TStrings`.

## Задание

1. В среде проектирования Delphi (C++ Builder) на основе компонента `TADOConnection` создайте приложение, соединяющееся с разработанной вами БД MS Access.
2. Предусмотрите в своем проекте возможность регистрации пользователя в БД.
3. Предусмотрите в своем проекте возможность сбора служебной информации о БД (имена доступных таблиц и их полей).

# 2. Общие черты компонентов доступа к данным

*Вид занятия – практическое занятие*
*Время занятия – 2 часа*
*Программное обеспечение – среда Microsoft Access, C++ Builder (Delphi)*

Более десятка визуальных элементов управления среды программирования Delphi предназначены для решения вопросов отображения содержимого отдельных полей, или коллекции строк из набора данных, многие из них обучены редактировать данные, а кое-кто даже способен управлять процессом ввода/удаления данных из таблиц. Благодаря этим компонентам, мы, не написав ни одной строки кода, сможем строить несложные приложения, работающие с базой данных. Для проверки этого утверждения дополните разработанный на лабораторной работе 11 проект новой формой и разместите на главной форме следующий перечень элементов управления:

1. Таблица `ADOTable1`, этот компонент вы обнаружите на страничке `dbGo` палитры компонентов.
2. Источник данных `DataSource1` (страничка Data Access).
3. Навигатор данных `DBNavigator1` (страничка Data Controls).
4. Сетка для отображения данных `DBGrid1` (страничка Data Controls).

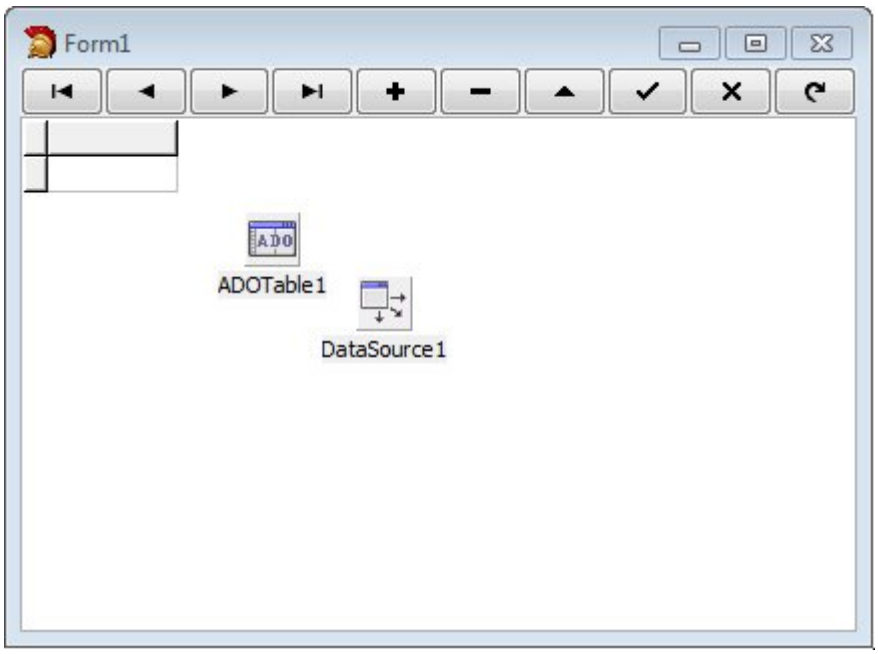



*Рис. 2.1. Внешний вид первого проекта БД*

Разместив все перечисленные компоненты на форме проекта (рис. 12.1), выберите таблицу – компонент `ADOTable1`. Нам предстоит проделать наиболее важную операцию – подключить компонент таблицу к физической таблице, которая в разработанном вами файле БД. Для этого выберите свойство Connection компонента и подключите его к компоненту-соединителю `ADOConnection1`.

Закончив настройку строки соединения, мы получаем возможность подключить компонент `ADOTable1` к файлу с таблицей. Для этого в инспекторе объектов находим свойство `TableName` и вводим имя любой таблицы из вашей БД, например `TableName:='SUPPLIERS'`. Настройка подключения завершена, нам осталось только активировать его, для этого устанавливаем свойство компонента `Active` в состояние `true`.

Настроим оставшиеся три компонента.

1. В свойство `DataSet` компонента `DataSource1` передаем имя компонента `ADOTable1`.
2. В свойство `DataSource` компонента `DBGrid1` передаем имя источника данных `DataSource1`.
3. В свойство `DataSource` компонента `DBNavigator1` передаем имя источника данных `DataSource1`.

Работа над проектом завершена.

## Базовый класс доступа к данным TDataSet

В распоряжение разрабатывающего приложения БД программиста средой Delphi предоставляется более десятка компонентов, позволяющих получить доступ к данным. Однако эта лабораторная работа посвящена изучению ни какого-то отдельного элемента управления, мы рассмотрим предтечу всех компонентов доступа к данным – абстрактный класс `TDataSet`. Класс нацелен на решение следующих основных задач:

1. Открытие и закрытие набора данных.
2. Перемещение по набору данных.
3. Редактирование записей в наборе.
4. Организация доступа к полям таблиц.
5. Фильтрация набора данных.
6. Обеспечение поиска данных.

Из-за того, что `TDataSet` возглавляет иерархию классов, предназначенных для доступа к данным, и служит в своем роде первопроходцем в этом направлении, значительная часть объявленных в нем методов не наполнена содержанием – они являются абстрактными.

### Открытие и закрытие набора данных

Простейшей проверкой на то, в каком состоянии находится набор данных, может стать контроль свойства

```
property Active: Boolean;
```

Если оно вернет значение `true`, то это признак того, что набор данных активен и, как минимум доступен для просмотра. Совместно со свойством работают методы

```
procedure Open; // открытие набора
procedure Close; // закрытие набора
```

Перевод в `true` свойства `Active` последовательно вызовет события

```
property BeforeOpen: TDataSetNotifyEvent; //перед открытием
property AfterOpen: TDataSetNotifyEvent;  //после открытия
```

Вызов метода `Close()` (присвоение значения `false` свойству `Active`) также вызовет два события

```
property BeforeClose: TDataSetNotifyEvent;//перед закрытием
property AfterClose: TDataSetNotifyEvent; //после закрытия
```

### Перемещение по набору данных

Мы уже знаем, что открытый непустой набор данных обеспечивает доступ к своим записям. Текущая запись набора обычно называется активной записью, именно на ней установлен курсор набора. Перемещение курсора приводит к смене активной (текущей) записи. Естественно в классе `TDataSet` реализован целый ряд методов, обеспечивающих перемещение курсора по набору. К наиболее часто используемым процедурам такого рода стоит отнести

```
procedure First; //переход на первую запись
procedure Prior; //возврат к предыдущей записи
procedure Next; //переход на следующую запись
procedure Last; //переход на последнюю запись
```

Для прыжка сразу на несколько записей понадобится помощь метода

```
function MoveBy(Distance: Integer): Integer;
```

В параметр `Distance` передается дистанция, на которую мы хотим переместиться по набору, от текущей записи. Если значение отрицательное, то перемещение осуществляется к началу набора, положительное – к концу. За отправную точку принимается текущая запись набора. Функция возвращает реальное число записей, на которое удалось переместить курсор. Вызов методов перемещения имеет смысл только в том случае, если набор не пуст. Узнать об этом поможет метод

```
function IsEmpty : Boolean;
```

Если в наборе нет ни одной строки, то функция возвратит `true`. Если в наборе есть записи, то общее количество доступных строк нам поведает свойство

```
property RecordCount : Integer;
```

Порядковый номер текущей записи храниться в свойстве

```
property RecNo : Integer;
```

Это свойство доступно только в компоненте `TADOTable`.

Два специальных свойства сигнализируют о местонахождении курсора. Если курсор находится на самой первой записи в наборе, то в `True` установится свойство

```
property Bof : Boolean;//сокр. от англ. "begin of file"
```

Если курсор окажется на самой последней записи, то в состояние `True` установится свойство

```
property Eof : Boolean;//сокр. от англ. "end of file"
```

С помощью перечисленных выше свойств часто реализуют циклы, перебирающие все записи внутри набора

**Листинг 2.1. Просмотр всех строк в таблице**

```
with ADOTable1 do
begin
DisableControls; //отключение визуальных элементов управления от набора данных
       First;
       while EOF<>true do
       begin
              Memo1.Lines.Add(FieldByName('Field1').AsString);
              Next;
       end;
EnableControls; //подключение элементов управления
end;
```

С перемещением курсора по набору данных связано два события

```
property BeforeScroll: TDataSetNotifyEvent;//перед перемещением
property AfterScroll: TDataSetNotifyEvent; //после перемещения
```

## Создание закладок и переход к закладке

В Delphi реализован простой способ запоминания текущего положения курсора (создание закладки) с возможностью последующего перемещения к данной записи. Для того, чтобы запомнить позицию текущей строки набора данных мы вызываем метод

```
function GetBookmark: TBookmark;
```

Функция возвращает данные в переменную-закладку. Для перехода к закладке следует воспользоваться услугами процедуры

```
procedure GotoBookmark(Bookmark: TBookmark);
```

Вместо функций `GetBookmak()` и `GotoBookmark()` допускается использовать свойство

```
property Bookmark: TBookmark;
```

Когда необходимость в услугах закладки отпадает, распределенный ресурс освободит метод

```
procedure FreeBookmark(Bookmark: TBookmark);
```

**Листинг 2.2. Работа с закладками**

```
private
    BM:TBookmark; //переменная для запоминания закладки
//…
procedure TForm1.btnGetBookmarkClick(Sender: TObject);
begin
BM:=ADOTable1.GetBookmark; //помечаем запись закладкой
end;

procedure TForm1.btnGoToBookmarkClick(Sender: TObject);
begin
if ADOTable1.BookmarkValid(BM)=true then //проверка закладки
begin
  ADOTable1.GotoBookmark(BM); //возвращаемся к закладке
  ADOTable1.FreeBookmark(BM); //удаляем закладку
  btnGoToBookmark.Enabled:=False;
end;
```

```
end;
```

## Редактирование записей в наборе

Модификация данных в наборе данных возможна только в ситуации, когда свойство

```
property CanModify: Boolean; //только для чтения
```

возвращает `false` — это признак того, что набор данных возражает против редактирования своих строк.

Получив потенциальное согласие набора данных на внесение изменений, можно добавить в конец набора новую, пустую запись. Для этого используйте метод

```
procedure Append;
```

Если требуется, чтобы новая запись оказалась в месте положения курсора, то применяйте

```
procedure Insert;
```

Вызов методов `Append` и `Insert` подготовит набор данных к приему новой записи. Если, например, средством визуализации избрана сетка (`TDBGrid`), то после вызова функции на экране компьютера пользователь увидит новую пустую строку, готовую к получению данных. Ему останется ввести информацию с клавиатуры. Если вы не доверяете пользователю, то одновременно с операцией добавления (или вставки) новой строки можно заполнить запись данными. Для этого используют методы:

```
procedure AppendRecord(const Values: array of const);
procedure InsertRecord(const Values: array of const);
```

Здесь в качестве параметров передается массив значений, непосредственно вносимых в таблицу. Будьте внимательны – последовательность значений, передаваемых в массиве должна совпадать с физической последовательностью полей в таблице.

```
Table1.InsertRecord(['Петров', 'Петр', 'Петрович', NULL]);
```

С процессом вставки новой записи связаны три последовательно возникающие события.

*Таблица 2.1. События связанные со вставкой новой строки*

| ***Событие*** | ***Описание*** |
|---|---|
| `property BeforeInsert: TDataSetNotifyEvent;` | Вызывается перед вставкой записи |
| `property OnNewRecord: TDataSetNotifyEvent;` | Вызывается в момент вставки новой записи |
| `property AfterInsert: TDataSetNotifyEvent;` | Вызывается по окончанию вставки записи |

Для перевода текущей записи в режим редактирования предназначена процедура

```
procedure Edit;
```

С операцией редактирования взаимодействуют события

```
property BeforeEdit: TDataSetNotifyEvent; //перед редактированием
property AfterEdit: TDataSetNotifyEvent; // после редактирования
```

Для того, чтобы сохранить внесенные изменения вызывают метод

```
procedure Post;
```

**Листинг 2.3. Вставка и сохранение новой записи**

```
with ADOTable1 do
begin
Append;
FieldByName('City').Value:= 'Ставрополь';
Post;
end;
```

Соответствующие процессу сохранения обработчики событий

```
property BeforePost: TDataSetNotifyEvent; //перед сохранением
property AfterPost: TDataSetNotifyEvent;  //после сохранения
```

Метод `Post()` следует вызывать только в случае, когда в результате редактирования в наборе данных возникли изменения. Узнать об этом поможет свойство

```
property Modified: Boolean;//только для чтения
```

Свойство возвратит `True`, если набор нуждается в сохранении. И наоборот значение `False` свидетельствует о том, что набор данных не изменялся.

Для удаления текущей записи используйте метод

```
procedure Delete;
```

Процедуру обслуживают события

```
property BeforeDelete: TDataSetNotifyEvent; //вызывается перед удалением
property AfterDelete: TDataSetNotifyEvent; //после удаления
```

Для отмены последних изменений в тексте используйте метод

```
procedure Cancel;
```

Учтите, что метод `Cancel` сохраняет работоспособность только до вызова метода `Post()`. Соответствующие обработчики событий:

```
property BeforeCancel: TDataSetNotifyEvent; //вызывается перед отменой
property AfterCancel: TDataSetNotifyEvent; //после отмены
```

## Фильтрация набора данных

Фильтрация данных позволяет осуществлять отбор части записей по определенному критерию (критериям) описанным в свойстве:

```
property Filter: string;
```

Правила описания фильтра достаточно просты и похожи на синтаксис, используемый в языке SQL в предложении `WHERE`. Далее приведем несколько примеров по синтаксису строки фильтра.

```
ADOTable1.Filter:= 'Town='+ QuotedStr(MOCKBA);
ADOTable1.Filter:= 'Town<>'+ QuotedStr(MOCKBA);
ADOTable1.Filter:= 'Price>1000 and Price<=2000';
```

Для включения (отключения) фильтра необходимо установить в `true` (`false`) свойство:

```
property Filtered : Boolean;
```

Некоторые особенности в организацию фильтрации данных вносит свойство:

```
property FilterOptions: TFilterOptions;
```

Значение `foCaseInsensitive` определяет чувствительность фильтра к регистру символов. Значение `foNoPartialCompare` разрешает отфильтровывать частичные совпадения.

```
ADOTable1.Filter:= 'Town>M*';
```

С процессом фильтрации связан специализированный обработчик события:

```
type TFilterRecordEvent = procedure(DataSet: TDataSet; var Accept: Boolean) of object;
property OnFilterRecord: TFilterRecordEvent;
```

В рамках указанного события программист получает возможность осуществить более тонкую настройку процесса фильтрации. Параметр `DataSet` возвращает ссылку на фильтруемый набор данных. С помощью переменной `Accept` управляется процесс отбора (фильтрации) записей: `true` – включать запись в результирующий набор, `false` – не включать.

**Листинг 2.4. Событие фильтрации строк набора данных**

```
procedure TForm1.ADOTable1FilterRecord(DataSet: TDataSet; var Accept: Boolean);
begin
{настраиваем фильтр на отбор записей на основе значения в поле Price, в результирующий
набор будут включены записи >500 и меньше 1000}
ACCEPT:=(DataSet['Price']>'500') and (DataSet['Price']<'1000');
```

```
end;
```

Обработчик события будет работать только в случае, если в `true` установлено свойство `Filtered`.

### Организация поиска данных

На уровне класса `TDataSet` реализован самый универсальный метод поиска `Locate`. Универсальность метода заключается в том, что он способен осуществлять поиск даже в неиндексированных полях.

```
function Locate(const KeyFields: String; const KeyValues: Variant;
                Options: TLocateOptions): Boolean;
```

Здесь: `KeyFields` – имя поля (или полей, разделенных точкой с запятой) в которых производится поиск. `KeyValues` – шаблон (или, когда поиск идет в нескольких полях – массив шаблонов) по которым будет осуществлен поиск. Параметр `Options : TLocateOptions` передается в квадратных скобках, внутри скобок через запятую значения:

- `loCaseInsensitive` – поиск нечувствителен к регистру символов.
- `loPartialKey` – допускается поиск по части ключа.

Также допускается оставлять скобки пустыми.

В листинге приведен пример вызова функции `Locate() для` поиска некого гражданина по фамилии “Петров” рожденного в 1980 году. Метод осуществит поиск сразу в двух полях таблицы, не обращая внимания на регистр символов.

**Листинг 2.5. Поиск строки в наборе данных**

```
with ADOTable1 do
  Locate('Name; BDay', VarArrayOf(['Петров', '1980']), [loCaseInsensitive]);
```

В случае обнаружения требуемых значений метод позиционирует курсор на этой записи и возвращает значение `True`.

### Взаимодействие с элементами управления данными

Как правило, потомки класса `TDataSet` (элементы управления `TADOTable`, `TADOQuery`, и т.д.) еще в период разработки приложения связываются с элементами управления, отвечающими за отображение и редактирование данных. Посредником в этих операциях выступает компонент `TDataSource` (источник данных). Для подключения к набору данных источника данных (компонент класса `TDataSource`) предназначено одноименное свойство

```
property DataSource: TDataSource;
```

## Задание

Улучшите начатый на лабораторной работе 11 проект БД. Используя компоненты-таблицы `TADOTable`, `TDataSource и TDBGrid` научите проект БД следующим операциям:

1. Перемещаться по строкам таблицы.
2. Создавать закладки и возвращаться к ним.
3. Добавлять, редактировать и удалять строки из таблицы.
4. Фильтровать строки в таблице (например, строить фильтр по дате).
5. Производить поиск строки в таблице, по значению поля.

**Внимание**! При выполнении задания нельзя использовать компонент `TDBNavigator`.

# 3. Работа с полями набора данных

*Вид занятия – практическое занятие*
*Время занятия – 2 часа*
*Программное обеспечение – среда Microsoft Access, C++ Builder (Delphi)*

При формировании библиотеки VCL программисты Embarcadero реализовали целый арсенал компонентов, предназначенных для организации доступа к данным. К моменту появления Delphi перечень этих элементов управления настолько расширился, что даже простое их перечисление способно внушить благоговение любому программисту. Вместе с этим, не смотря на впечатляющий перечень классов, компоненты достаточно просты в изучении, так как построены по схожим принципам. Как уже отмечалось на предыдущем занятии, все компоненты доступа к данным обладают общим предком – классом `TDataSet`. Материал, сегодняшнего занятия, также является общим для всех компонентов доступа к данным. Он посвящен особому классу `TField` – полю набора данных.

## Поле таблицы – класс TField

Класс `TField` (от англ. *field* поле) инкапсулирует физический объект – поле набора данных. Сам по себе класс `TField` является абстрактным и посему в первозданном виде не используется. А вот богатое потомство класса интегрируется во все построенные на основе класса `TDataSet` компоненты. При открытии набора данных (например, при помощи компонентов `TADOTable` или `TADOQuery`) автоматически рождается ровно столько объектов `TField`, сколько столбцов физически доступно в этом наборе. Объекты-поля созданные таким образом (без явного вмешательства программиста) называют **динамическими**. Основная особенность подобных объектов в том, что они уничтожаются сразу после отключения от набора данных.

Однако, в некоторых ситуациях, и на долю программиста выпадает обязанность немного поработать. Особенно если идет речь о реализации:

- полей синхронного просмотра (*lookup*);
- вычисляемых полей (*calculated*);
- агрегатных полей (*aggregate*).

Для этого в среде Delphi реализован специальный редактор “`Fields Editor`”, вызываемый по двойному щелчку кнопкой мышки по компоненту-набору данных. Все объекты-поля, созданные руками программиста, именуют **статическими**. В отличие от динамического поля, жизненный цикл его статического собрата начинается практически в момент старта приложения, продолжается до завершения его работы и ни коим образом не зависит от того, открыт или закрыт набор данных.

> Один и тот же набор данных не в состоянии одновременно обладать, как динамическими, так и статическими полями. И если в наборе определено хотя бы одно статическое поле, то компонент, утратит возможность самостоятельно создавать его динамических коллег.

Теперь немного об основных обязанностях класса `TField`, потомки класса умеют:

1. Читать данные из текущей записи конкретного столбца таблицы.
2. Редактировать значения поля, текущей записи с синхронной проверкой допустимости вводимых данных.
3. Указывать компонентам отображения данных, каким образом следует представлять содержимое поля на экране.
4. Обрабатывать вычисляемые поля.
5. Искать значения поля синхронного просмотра в другом наборе данных

## Классификация полей по функциональному назначению

Для того, чтобы выяснить назначение поля сразу стоит обратиться к его свойству:

```
property FieldKind: TFieldKind;
```

В подавляющем числе случаев обращение к этому это свойству несет справочный характер, и редактировать хранящееся в нем значение не только излишне, но и даже опасно. Список возможных вариантов ответов не велик и представлен в таблице 3.1.

*Таблица 3.1. Значения TFieldKind*

| ***Значение*** | ***Описание*** |
|---|---|

| | |
|---|---|
| fkData | Объект представляет собой поле данных. |
| fkCalculated | Это вычисляемое поле, содержимое которого определяется в обработчике события OnCalcFields() содержащего его компонента. |
| fkLookup | Поле синхронного просмотра. |
| fkInternalCalc | Внутренне вычисляемое поле. |
| fkAggregate | Агрегатное поле. |

Самый распространенный тип поля – fkData. На физическом уровне это чаще всего поле обычной таблицы, доступ к которому мы получили с помощью компонентов TTable, TQuery или их коллег.

### Классификация полей по типу обслуживаемых данных

Ключевой характеристикой поля служит тип данных, который может в нем содержаться. Для выяснения этого прочитайте значение свойства

```
property DataType: TFieldType;
```

Например, поле предназначенное для работы с 32-х разрядными целыми числами имеет тип данных ftInteger, вещественное поле – ftFloat, строковое – ftString.

### Обращение к отдельному объекту-полю

Для того чтобы выяснить, какому набору данных принадлежит поле стоит обратиться к свойству

```
property DataSet: TDataSet;
```

Свойство вернет ссылку на родительский компонент.

Каждое поле владеет информацией об имени связанного с ним столбца таблицы

```
property FieldName : string;
```

В наборе данных (потомке TDataSet) поля хранятся в свойстве Fields. В этом списке, каждое поле описывается уникальным индексом.

```
property Index: Integer;
```

Изменяя индекс, мы получаем возможность корректировать последовательность полей в этом наборе. Порядковый номер поля в наборе данных редактировать нельзя. Это свойство доступно только для чтения

```
property FieldNo: Integer;//только для чтения;
```

Оно отображает физическую последовательность столбцов в наборе.

### Доступ к данным поля

Для того, чтобы проверить содержит ли поле значение обращайтесь к свойству:

```
property IsNull: Boolean;
```

Если значение поля неопределенное, то свойство вернет true. Если значение поля определено, то вы имеете полное право выяснить, что там находится. Для чтения и записи данных в поле предназначается свойство

```
property Value: Variant;
```

```
if ADOTable1.Fields[0].IsNull=False then
Result:= Table1.Fields[0].Value;
```

Для полной очистки содержимого поля стоит применять процедуру

```
procedure Clear;
```

После вызова метода значение поля будет соответствовать NULL.

Поле можно перевести в режим “только для чтения” применив свойство

```
property ReadOnly: Boolean;
```

Это свойство позволяет программисту программным образом запрещать или разрешать редактировать данные. Но перед этим стоит проверить доступно ли для правки поле на физическом уровне, ведь оно, например, может оказаться вычисляемым, т.е. не редактируемым в принципе. Для этого требуется проанализировать состояние свойства:

```
property CanModify: Boolean;//только для чтения
```

Свойство информирует нас о том, разрешено (`true`) или запрещено (`false`) модифицировать данные поля.

При вводе новой или редактировании существующей записи важную роль играет свойство:

```
property Required: Boolean;
```

Значение `true` свидетельствует о том, что это так называемое "**обязательное**" поле. Другими словами, оно должно быть заполнено, иначе (если при сохранении записи поле окажется пустым) – мы получим сообщение об ошибке.

## Приведение типов

Один и тот же объект-поле способен представлять содержащееся в нем значение в виде некоторых стандартных типов данных. Для этого в классе `TField` инкапсулировано несколько свойств (табл. 3.2).

*Таблица 3.2. Приведение типов данных в поле TField*

| ***Свойство*** | ***Описание*** |
|---|---|
| `property AsBCD: TBcd;` | Содержит значение поля в формате двоично-кодированного десятичного числа. Значения такого типа иногда применяются вместо денежных (currency) типов данных |
| `property AsBoolean: Boolean;` | Содержит значение в логическом формате |
| `property AsCurrency: Currency;` | Денежный формат |
| `property AsDateTime: TDateTime;` | Формат дата/время |
| `property AsFloat: Double;` | Вещественное число |
| `property AsInteger: Integer;` | Целое число |
| `property AsSQLTimeStamp: TSQLTimeStamp;` | Значение поля представлено в формате `TSQLTimeStamp`. Это один из способов хранения даты/времени, в виде записи, применяемый в серверах SQL |
| `property AsString: String;` | Строка символов |
| `property AsVariant: Variant;` | Универсальный формат |

Следующая строка кода демонстрирует способ передачи данных из поля в компонент-метку.

```
Lable1.Caption:=Query1.FieldByName('CityName ').AsString;
```

Или обратный пример, показывающий каким образом можно забрать значение из компонента-календаря:

```
Table1.Fields[1].AsDateTime:= DateTimePicker1.Date;
```

### Размер поля

При обращении к значению поля иногда требуется знать физический размер, занимаемый этим значением. для таких целей предназначено свойство:

```
property DataSize: Integer;
```

Свойство возвратит размер в байтах. Иногда часто это свойство применяют для того, чтобы зарезервировать область памяти для хранения содержимого поля.

Еще одно свойство:

```
property Size: Integer;
```

для поля строкового типа указывает максимальное число символов, допустимых для ввода в это поле. А для остальных типов полей – размерность в единицах, зависящих от типа данных поля.

### Значение по умолчанию

Значение по умолчанию, вносимое в поле при вводе новой записи определяется в свойстве.

```
property DefaultExpression: String;
```

Как правило, в этом свойстве указываются наиболее вероятные данные, что освобождает пользователя от повинности многократного ввода одного и того же значения.

```
FieldCurrentTime.DefaultExpression:='12:00:00';
```

Для наибольшей универсализации свойство сделано строковым. Это позволяет задавать значения по умолчанию для большинства типов полей.

### Ограничения на ввод данных

Объекты поля способны осуществлять строгий контроль за корректностью вводимых данных. Набор свойств и методов поля позволяет, как импортировать доменные ограничения, заложенные в ядре СУБД, так и определять ограничения на уровне клиентского приложения. Благодаря этому строится гибкая и многоуровневая система защиты от ввода некорректных данных.

Ограничения на диапазон допустимых значений определяются в свойстве:

```
property CustomConstraint: String;
```

Это ни что иное, как обычная строка, в которой описывается выражение на языке SQL. Допустим, что физическое имя поля “field1”, тогда примерно так будет выглядеть выражение, определяющее допустимые пределы вводимого значения:

```
CustomConstraint:='field1>=10 and field1<100';
```

При попытке пользователя ввести недопустимое значение набор данных, генерирует исключительную ситуацию. Для того, чтобы сообщение об ошибке было понятно для пользователя целесообразно описать, вероятную причину ошибки в свойстве:

```
property ConstraintErrorMessage: string;
```

Если ограничения определены на уровне СУБД, то вы можете лишь ознакомится с ними, посмотрев содержимое свойства:

```
property ImportedConstraint: string;
```

Попытка внести изменения в импортируемые ограничения обречена на провал.

Еще одно справочное свойство предназначено для проверки – есть ли какие-нибудь ограничения на вводимые значения

```
property HasConstraints: Boolean; //только для чтения
```

### Маска ввода

Маска ввода, предназначена для определения формата вводимых пользователем значений и для повышения наглядности представляемых данных.

```
property EditMask: TEditMask;
```

### Индексные поля

На основе поля или группы полей в таблице может быть построен индекс. Если поле входит в состав установленного в настоящий момент индекса набора данных свойство:

```
property IsIndexField: Boolean;
```

возвращает значение `true`.

### Отображение данных

Класс `TField` и его потомки не вооружены методами, позволяющими визуализировать свое содержимое. Однако компоненты линейки **Data Controls** палитры компонентов, отвечающие за отображение информации вполне способны узнать у класса `TField` каким образом он хотел бы представлять свои данные на экране.

Данные, предназначенные для вывода на экран, хранятся в свойстве

```
property DisplayText: String; //только для чтения
```

Все компоненты (`DBText`, `DBEdit`, `DBGrid`, и т.д.) при отображении данных обратятся именно к этому свойству поля.

Если поле находится в режиме редактирования, то все изменения (до момента их сохранения) аккумулируются в свойстве.

```
property Text: String;
```

В период редактирования содержимое свойств `DisplayText` и `Text` различается.

При назначении имен заголовков колонок сетка `TDBGrid` обязательно проверит содержимое свойства объекта `Field`. В этом свойстве по умолчанию хранится физическое имя поля, сетка `DBGrid` “подхватит” это имя и с его помощью определит заголовок колонки. Альтернативный способ назначения подписей полей обеспечивает свойство

```
property DisplayLabel : String;
```

В этом свойстве по умолчанию находится имя физическое имя поля, однако программист имеет право вместо невразумительного “ColumnCity” написать “Город”. И тогда имена колонок сразу получат осмысленные названия.

Еще одно свойство, связанное с отображением данных, называется:

```
property DisplayName: String;        //только для чтения
```

Содержимое этого свойства используется при выводе сообщения об ошибке, возникшей при редактировании данных в поле.

## Задание

1. Разработайте функцию, на вход которой поступает имя любой таблицы вашей БД. В результате выполнения функция возвратит все строки из заданной таблицы в многострочный редактор `TMemo`.
2. Разработайте функцию, на вход которой поступает имя любой таблицы вашей БД. В результате выполнения функции вы должны получить сведения о составе полей этой таблицы, включая: имя поля, тип поля, размер данных поля.

# 4. Вычисляемые поля и поля подстановки

*Вид занятия – практическое занятие*
*Время занятия – 2 часа*
*Программное обеспечение – среда Microsoft Access, C++ Builder (Delphi)*

Опытные программисты знают, что без особой необходимости в таблицах не следует хранить информацию, дублирующую уже существующие данные. Также следует отказываться от накопления информации, которая может быть получена путем проведения простейших операций над существующими данными. Поясним это утверждение на примере. Допустим, что перед нами поставлена задача – разработать базу данных содержащую сведения о зарплате сотрудников предприятия и размере подоходного налога. В такой ситуации заводить отдельное поле для внесения в него суммы налога избыточное действие, ведь зная сумму жалования значение налога можно получать при помощи простейшей математики.

## Поля подстановки

Поля подстановки (lookup fields) активно применяются программистами для построения связи между главной и справочной таблицей. Изучите схему данных, представленную на рисунке 14.1. В нашем распоряжении две таблицы. В справочную таблицу DEPARTMENT занесены названия отделов и служб предприятия. Во второй таблице SALARY вы найдете текстовое поле FIO, предназначенное для хранения фамилии и инициалов и одноименное с именем таблицы поле SALARY специализирующееся на запоминании должностного денежного оклада работника предприятия.

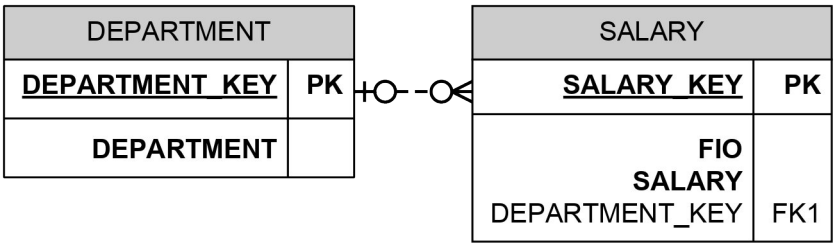



*Рис. 4.1. Модель данных для примера работы с полем подстановки*

Связь между таблицами осуществляется по ключевому полю DEPARTMENT_KEY. При вводе новой записи в таблицу SALARY в поле внешнего ключа DEPARTMENT_KEY передается значение, соответствующее первичному ключу в справочной таблице DEPARTMENT. Однако такое представление информации абсолютно неприемлемо для пользователя, вместо непонятных цифр в таблице ему хотелось бы видеть реальное название отдела. К счастью это не сложно, механизм, осуществляющий поиск данных в справочной таблице, с последующим показом найденных данных в исходной таблице осуществляется при посредничестве поля подстановки.

Создадим базу данных в Microsoft Access. Для этого проведем следующие операции:

1. Создаете новый проект базы данных в среде Access и сохраните ее в отдельном каталоге, например под именем demo.mdb.
2. Создайте таблицу DEPARTMENT включающую поле первичного ключа DEPARTMENT_KEY автоинкрементного типа; текстовое поле длиной 30 символов DEPARTMENT для хранения названия отдела предприятия.
3. Создайте таблицу SALARY (рис. 4.2) состоящую из четырех полей: поле первичного ключа SALARY_KEY автоинкрементного типа; текстовое поле длиной 30 символов FIO для хранения фамилии сотрудника; поле денежного типа SALARY с данными о ежемесячном жаловании и внешнего ключа DEPARTMENT_KEY целочисленного типа.
4. У поля SALARY установите значение по умолчанию равное 0.

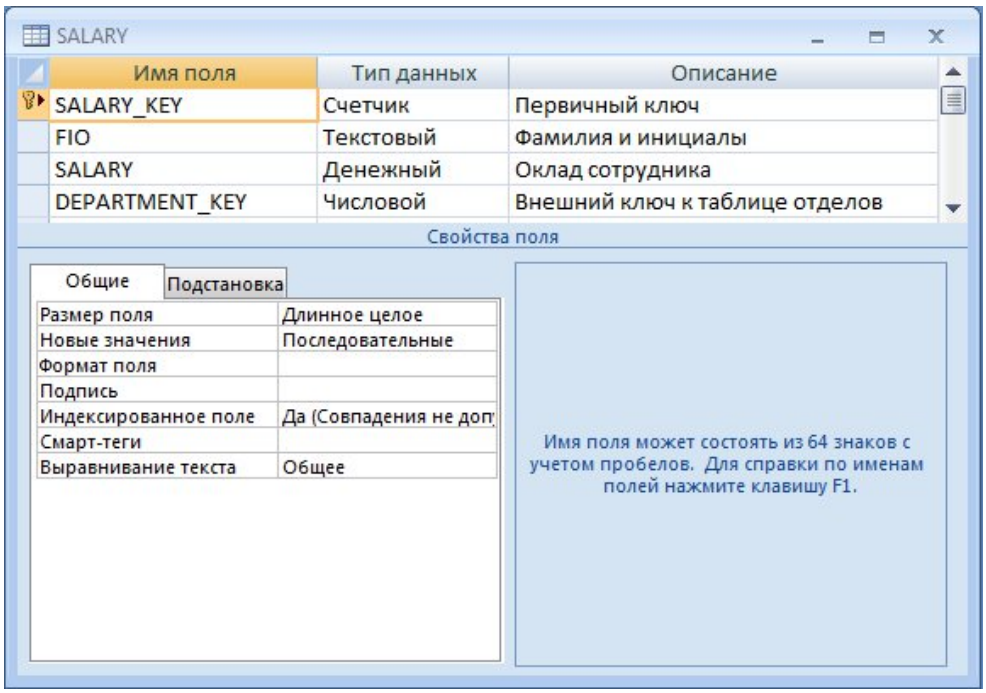



*Рис. 4.2. Таблица SALARY в режиме конструктора*

Введите в таблицу DEPARTMENT несколько записей и закрывайте программу Access, в ближайшее время она нам не понадобится. Теперь приступим к проектированию приложения Delphi. Начните новый проект. Переименуйте главную форму проекта `Name:=frmMain` и расположите на ее поверхности следующие компоненты (рис. 4.3):

- Две таблицы (компоненты класса `TADOTable`). Первый компонент предназначен для взаимодействия с таблицей DEPARTMENT, поэтому переименуем его в `tblDEPARTMENT`. Второй компонент станет обслуживать таблицу SALARY, назовем его `tblSALARY`.
- Источник данных `DataSource1`. Компонент предназначен для взаимодействия с таблицей SALARY, переименуйте его в `dsSALARY`. С помощью свойства `DataSet` подключите источник данных к компоненту `tblSALARY`.

Сетку для отображения данных `DBGrid1` и компонент для управления данными `DBNavigator1`. С помощью свойства `DataSource` присоедините оба компонента к `dsSALARY`.

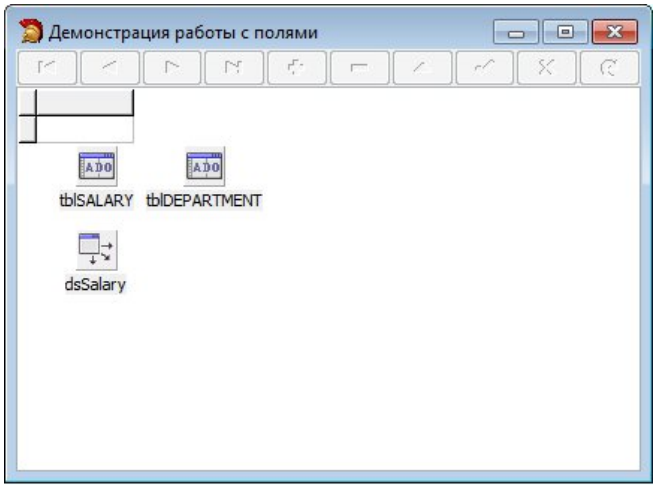



*Рис. 4.3. Вид главного окна приложения*

Сохраните проект под именем fielddemo.dpr в том же каталоге, где находится созданная ранее база данных demo.mdb. выберите компонент `tblDEPARTMENT` и, воспользовавшись свойством `ConnectionString`, подключите его к нашей базе данных. Скопируйте эту же строку в свойство `ConnectionString` второго компонента– таблицы `tblSALARY`, ведь ему нужен доступ к этой же самой БД.

Закончив работать с соединительной строкой, подключите компоненты к соответствующим таблицам:

- компонент `tblDEPARTMENT` к таблице отделов и служб `TableName:= 'department.db';`

- компонент `tblSALARY` к таблице сотрудников `TableName:='salary.db'`.

Проверьте корректность подключения, активировав свойства `Active` обеих таблиц.

Дважды щелкните левой кнопкой мышки по компоненту `tblSALARY` для вызова редактора полей этой таблицы. Во всплывающем меню редактора полей выберите пункт “Add all fields” (Добавить все поля). После этого в редакторе появится все поля таблицы. В результате форма заполнится списком полей, входящими в состав нашей таблицы SALARY. Осуществленную нами операцию называют определением статическим полей. В результате для каждого из полей таблицы SALARY среда Delphi поставила в соответствие новый объект – потомок класса `TField`. Класс поля определяется типом хранящихся в нем данных, например, для текстового поля создается объект класса `TStringField`, для денежного – `TFMTBCDField`; для поля счетчика – `TAutoIncField`, и т.д. Каждый объект-поле получает имя путем сложения имен компонента-таблицы и физического имени поля. В нашем случае поле, хранящее данные о фамилии сотрудника будет названо Table1+FIO=tblSALARYFIO.

Редактор “New Field” не предназначен для физического добавления поля в структуру таблицы. Он лишь способен создавать экземпляры поля (потомки класса `TField`) на программном уровне. В качестве основных клиентов редактора выступают вычисляемые (*calculated*) поля и поля подстановки (*lookup*).

Вновь вызовите контекстное меню редактора полей и найдите пункт меню “New field…” (Новое поле). Это действие вызовет окно настройки (рис. 4.4) в котором следует описать характеристики поля подстановки:

- в строке “Name” (имя поля) укажем имя создаваемого поля – `luDEPARTMENT`;
- в списке “Type” (тип данных) определим тип данных этого поля – `String`;
- в группе переключателей “Field type” (тип поля) отметим переключатель Lookup.

Заполняем четыре комбинированных списка в группе “Lookup definition”. Ввод данных производим слева направо, сверху вниз. Приступим. выпадающий список “Key fields” определяет поле таблицы сотрудников, являющееся внешним ключом, то есть предназначенным для взаимодействия с таблицей отделов и служб. Внешним ключом в таблице сотрудников является поле DEPARTMENT_KEY, поэтому останавливаем свой выбор на нем. В выпадающем списке “Dataset” указываем имя справочного набора данных. В нашем случае это таблица tblDEPARTMENT. Во втором ряду, в списке “Lookup Keys” требуется ввести имя первичного ключа таблицы-справочника отделов и служб – поле DEPARTMENT_KEY. И, наконец, в последнем списке “Result Field” необходимо выбрать имя поля, которое мы хотим видеть в результирующем наборе данных – это название отдела – DEPARTMENT. Закончив описание нового поля, нажимаем кнопку OK.

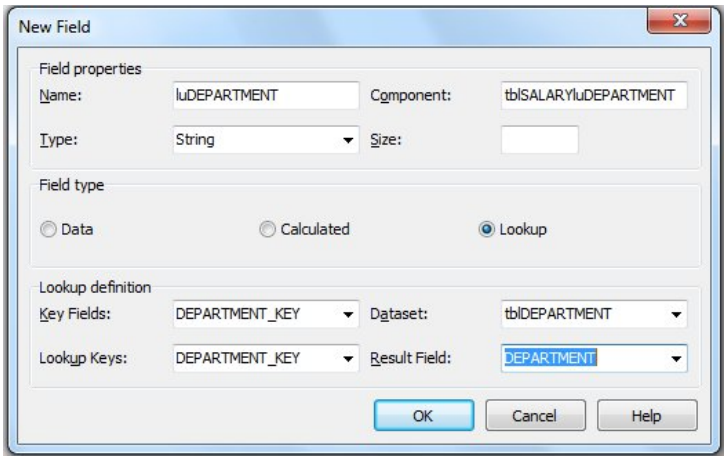



*Рис. 4.4. Создание подстановочного поля*

С этого момента наш проект приобретает работоспособность, но, к сожалению, нельзя сказать, что он лишен недостатков. Один из них – избыточность информации отображаемой в сетке `DBGrid1`. Этот компонент выводит на экран столбцы которые совсем необязательно видеть пользователю, это служебные поля первичного (SALARY_KEY) и вторичного (DEPARTMENT_KEY) ключей таблицы `tblSALARY`. Рекомендую вам найти свойства `visible` названных полей и присвоить им значения `false`, что сделает поля невидимыми. Кроме того имеет смысл настороить свойство `DisplayLabel` всех оставшихся видимыми полей так, чтобы сетка `DBGrid1` выводила осмысленные названия подсказывающие пользователю назначение того или иного столбца (рис. 4.5).

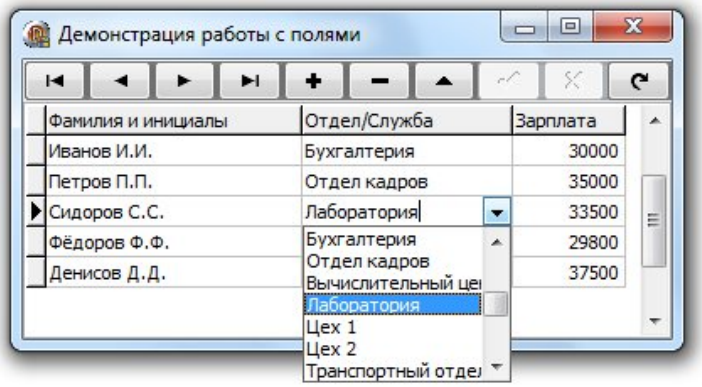



*Рис. 4.5. Поле подстановки в действии*

## Вычисляемые поля

Вновь возвращаемся к примеру. Еще раз обратимся к всплывающему меню редактора полей `tblSALARY` и выберем пункт “New Field”. На экран будет вызван редактор, позволяющий создавать новый экземпляр поля.

В нашем случае мы конструируем вычисляемое поле. Для этого в строке `Name` указываем имя нового поля – `CALC_TAX`, в строке `Type` – определяем тип поля `Currency`. Убедитесь, чтобы в группе переключателей “Field type” была выделена кнопка “Calculated” (рисунок 4.6) и нажмите на кнопку OK. После того, как окно редактора закроется, загляните в инспектор объектов. В нем появился новый объект – поле `tblSALARYCALC_TAX`, а его свойство `FieldKind` установилось в состояние `fkCalculated`.

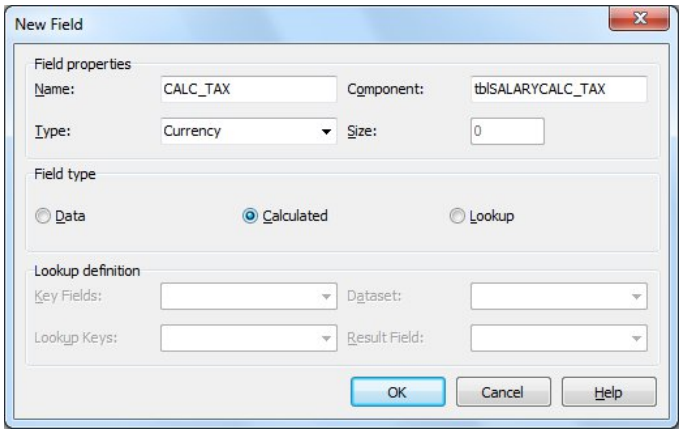



*Рис. 4.6. Создание вычисляемого поля*

Наступил черед поломать голову над листингом программы. У компонента `tblSALARY` находим обработчик события `OnCalcFields()` и внутри его напишем всего одну строку.

**Листинг 4.1. Пример расчета значения для вычисляемого поля**

```
procedure TfrmMain.tblSALARYCalcFields(DataSet: TDataSet);
begin
tblSALARYCALC_TAX.Value:=tblSALARYSALARY.AsCurrency*13/100;
end;
```

Проверьте состояние свойства `AutoCalcFields` таблицы, оно должно быть установлено в `True`. Опять “включаем” таблицу (`Active:=True`) и запускаем проект на выполнение. Теперь отображающая данные сетка `DBGrid1` дополнилась новой колонкой, содержащей рассчитанное значение подоходного налога.

## Организация отношения главная-подчиненная таблица

Один из полезных приемов, которым владеют компоненты таблицы, и в том числе `TADOTable` связан с построением отношения главная-подчиненная (master-detail) таблица. Подобное отношение имеется и в нашей демонстрационной БД книжного магазина (рис. 4.7). Таблице поставщиков SUPPLIERS подчиняется таблица контрактов CONTRACTS. Отношение подчиненности реализуется через хранящее ключ поставщика ключевое поле SUPPLIER_KEY.

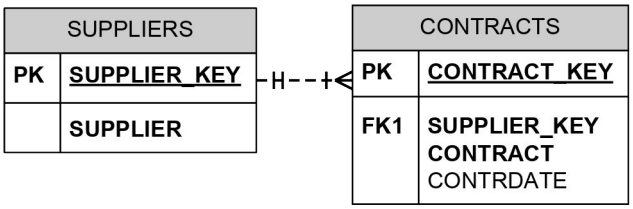



*Рис. 4.7. Отношение "главный-подчиненный" между таблицами*

Функциональные возможности компонента `TADOTable` позволяют формировать отношение главный-подчиненный при желании даже без единой строки кода. Для претворения такого плана в жизнь достаточно помнить о существовании свойств: `MasterSource` и `MasterFields`. Все перечисленные свойства имеют прямое отношение к подчиненной таблице и настраиваются в строгой последовательности. В первую очередь в свойстве:

```
property MasterSource: TDataSource;
```

указывается на подключенный к главной таблице источник данных (компонент `TDataSource`). Затем программист определяет перечень полей, участвующих в организации взаимодействия между таблицами. Для этого в свойстве:

```
property MasterFields: String;
```

указывают имена столбцов, входящих в первичный ключ главной таблицы.

Для практического претворения в жизнь нашего замысла можете взять за основу один из разработанных в прошлой главе проектов. Благо он уже обучен подключаться к хранилищу данных. Добавьте к предыдущей разработке новую форму и следующий перечень компонентов (рис. 4.8):

- две таблицы `TADOTable`. Переименуйте первую таблицу в `tblSUPPLIERS`, этот компонент станет обслуживать таблицу поставщиков. Вторую таблицу назовем `tblCONTRACTS`, мы позднее подключим этот компонент к таблице контрактов.
- два источника данных `TDataSource`. Первый источник переименуем в `dsSUPPLIERS` и в инспекторе объектов подключаем к таблице поставщиков (свойство `DataSet:= tblSUPPLIERS`). Второй источник называем `dsCONTRACTS` и присваиваем свойству `DataSet:= tblCONTRACTS`.
- две сетки `TDBGrid`. Сетки позволят нам увидеть содержимое таблиц. Первый компонент называем `dbgSUPPLIERS` и располагаем в левой части формы. Подключаем сетку к источнику данных `dsSUPPLIERS` (свойство `DataSource:= dsSUPPLIERS`). Оставшаяся сетка получит имя `dbgCONTRACTS`, свойство `DataSource:=dsCONTRACTS`.

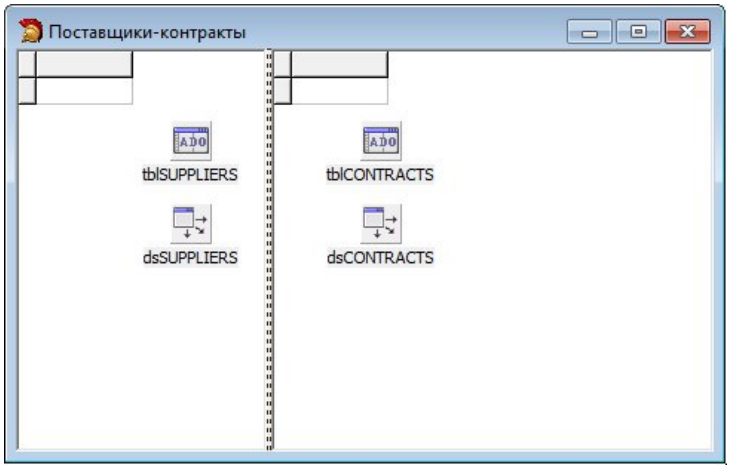



*Рис. 4.8. Форма, демонстрирующая отношение главный-подчиненный*

Переименуйте форму в `frmSuppliersContracts` и найдите событие `OnShow()` вызываемое в момент вывода формы на экран и внесите в него следующие строки кода.

**Листинг 4.2. Настройка параметров компонентов**

```
procedure TfrmSuppliersContracts.FormShow(Sender: TObject);
begin
with tblSUPPLIERS do
begin
```

```
  Connection:=frmDM.ADOConnection1;
  TableName:='SUPPLIERS';
  Open;
end;

with tblCONTRACTS do
begin
  Connection:=frmDM.ADOConnection1;
  TableName:='CONTRACTS';
  MasterSource:=dsSUPPLIERS;
  MasterFields:='SUPPLIER_KEY';
  Open;
end;
end;
```

Программа готова. Запустите приложение на выполнение внимательно проследите за его работой. выбор в сетке `dbgSUPPLIERS` того или иного поставщика приводит к автоматическому отображению в сетке `dbgCONTRACTS` сведений о заключенных с ним контрактов.

## Задание

Дополните ваш проект БД примерами:

1. полей подстановки.
2. вычисляемых полей.
3. отношением главная таблица — подчиненная таблица.

При разработке приложения рекомендуется воспользоваться следующими компонентами: `TADOTable`, `TDataSource`, `TDBNavigator` и `TStringGrid`.

# 5. Управление индексами, поля BLOB

*Вид занятия – практическое занятие*
*Время занятия – 2 часа*
*Программное обеспечение – среда Microsoft Access, C++ Builder (Delphi)*

Индекс представляет собой часть БД, в нем содержится информация об организации данных в таблицах БД. Индекс аналогичен предметному указателю, обычно приводимому в конце книги. Благодаря нему, при поиске нужной информации исключается необходимость полистного просмотра книги.

Индекс отвечает за ускорение процесса сортировки и поиска записей в наборе данных.

Различают первичные и вторичные индексы. **Первичный индекс** это ничто иное, как первичный ключ таблицы – он гарантированно содержит уникальные значения для всех записей в таблице. Если поле (поля) индексирования не является первичным ключом, то оно называется **вторичным индексом**.

## Особенности набора данных при работе с вторичными индексами

Одна и та же таблица вправе обзавестись некоторым перечнем вторичных индексов. Но в один и тот же момент времени таблица имеет право пользоваться услугами одного единственного индекса.

Для выяснения списка имен доступных индексов вызывается процедура:

```
procedure GetIndexNames(List: TStrings);
```

Перечень имеющихся в наличии индексов процедура возвратит в набор строк `List`.

Подключение индексов по его имени или по названиям входящих в индекс полей осуществляется с помощью свойств:

```
property IndexName: WideString;    //имя индекса
property IndexFieldNames: string; //поля индекса
```

Ниже предложен листинг, демонстрирующий способ выбора текущего индекса пользователем во время работы программы.

**Листинг 5.1. Подключение индекса**

```
procedure TForm1.ADOTable1AfterOpen(DataSet: TDataSet);
begin
ADOTable1.GetIndexNames(ComboBox1.Items); //сбор индексов
end;
procedure TForm1.ComboBox1Change(Sender: TObject);
begin
if ComboBox1.ItemIndex<>-1 then
ADOTable1.IndexName:=ComboBox1.Items.Strings[ComboBox1.ItemIndex];
end;
```

Сразу после соединения с таблицей (в момент возникновения события `AfterOpen`) производится сбор имен индексов в комбинированный список `ComboBox1`. Теперь пользователь приобрел право определить нужный ему режим сортировки данных, для этого он просто выбирает требуемый индекс из списка `ComboBox1`.

## Поля BLOB

Класс `TBlobField` реализован для обеспечения работы с двоичными большими объектами (*binary large object*, BLOB) переменной длины. Поля этого типа и его производные представляют собой универсальное хранилище большого объема разнотипной информации. Поля BLOB широко представлены практически во всех СУБД. Например: в таблицах Paradox к таковым относятся большие текстовые поля (Memo), поля форматированного текста (`FormattedMemo`), графическое поле (`Graphic`).

Если рассуждать о специализации BLOB полей, то стоит обратить внимание на свойство:

```
property BlobType: TBlobType;
```

где

```
type TBlobType = ftBlob..ftOraClob;
```

Тип данных `TBlobType` это всего-навсего подмножество `TFieldType`. Значения, которые может принимать это тип данных, вы можете просмотреть в справочной системе. Хотя свойство `BlobType`

доступно как для чтения, так и для записи, но если возникла необходимость, поменять тип поля целесообразнее вызвать метод:

Почти все методы BLOB поля нацелены на решение вопросов чтения или записи данных.

Для загрузки данных в поле предназначены два метода:

```
procedure LoadFromFile(const FileName: string);
procedure LoadFromStream(Stream: TStream);
```

В первом случае требуется передать имя графического файла `FileName`. Реализация второго метода несколько сложнее, в нем работа осуществляется с областью памяти – потоком `Stream`.

Для выгрузки данных из поля реализованы два симметричных метода с аналогичными параметрами:

```
procedure SaveToFile(const FileName: string);
procedure SaveToStream(Stream: TStream);
```

Для копирования данных из другого BLOB-поля определена процедура.

```
procedure Assign(Source: TPersistent);
```

Здесь в качестве параметра передается источник данных – `Source`. В роли источника данных могут выступать: другие BLOB поля, набор строк `TStrings`, графические объекты, например `TBitmap`.

Факт изменения данных внутри поля можно проверить, просмотрев свойство:

```
property Modified: Boolean;
```

Значение `True` является признаком, что поле изменялось.

Разработаем небольшой пример, демонстрирующий работу с BLOB полем. Задача заключается в следующем: нам требуется научить приложение сохранять в таблице растровую картинку и имя файла, соответствующего этому изображению. Начнем с проектирования структуры таблицы. Для решения поставленной задачи нам достаточно определить три поля: первичный ключ PICTURE_KEY, поле PICTUREBLOB для хранения графики и текстовое поле для хранения имени файла PICTURENAME. Воспользуемся услугами Microsoft Access и создадим в ней файл БД с именем blobdemo.accdb. База данных будет содержать единственную таблицу PICTURETABLE (рис 5.1) с описанными выше характеристиками.

PICTURETABLE

| Имя поля | Тип данных | |
|---|---|---|
| PICTURE_KEY | Счетчик | Ключевое поле |
| PICTURENAME | Текстовый | Название файла с рисунком |
| PICTUREBLOB | Поле объекта OLE | Объект BLOB |

*Рис. 5.1. Структура таблицы для работы с полем BLOB*

Создайте новый проект и разместите на главной форме следующий перечень компонентов: таблица – `ADOTable1`, диалог доступа к файлам – `FileOpenDialog1`, объект `Image1` – для отображения рисунков, метку для отображения названия файла – `StaticText1`, и семь кнопок класса `TSpeedButton`. Первые четыре кнопки (внешний вид формы на рисунке 5.2) предназначены для перемещения по записям внутри таблицы. Назовем их следующим образом: `btnFirst` – для перемещения на самую первую запись, `btnPrior` – предыдущая запись, `btnNext` – следующая запись, `btnLast` – последняя запись. Кроме имен кнопок потребуется изменить значения их свойств `tag`. Для `btnFirst` – 0, `btnPrior` – 1, `btnNaxt` – 2, `btnLast` – 3. И осталось три кнопки, отвечающие непосредственно за редактирование данных в таблице: `btnAppend` – добавляет новую запись в конец набора, `btnEdit` – редактирует текущую запись и `btnDelete` – отвечает за удаление.

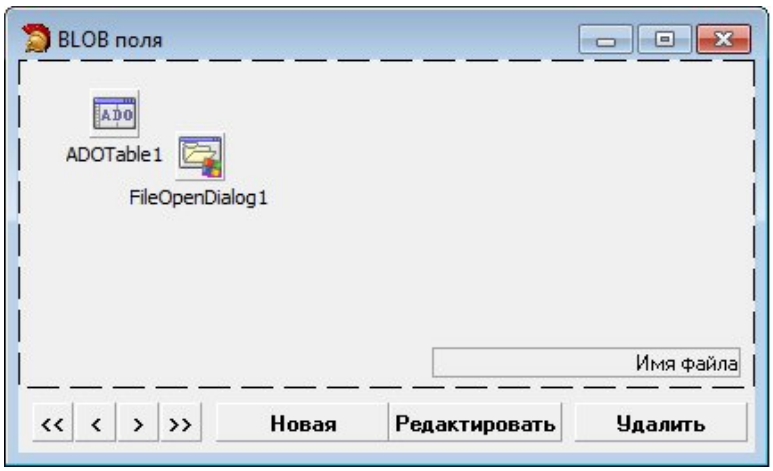



*Рис. 5.2. Внешний вид формы для работы с BLOB-полем*

Сохраните проект в том же каталоге где и расположен файл blobdemo.accdb. Главную форму проекта по традиции назовем `frmMain.dfm`, соответствующий ей модуль – `main.pas`, а весь проект – `blobdemo.dpr`.

Настройте строку соединения для компонента `ADOTable1`.

В секции частных объявлений опишем заголовки двух процедур `SetData()` и `GetData()`. Процедура `SetData()` предназначена для организации загрузки файла рисунка в BLOB поле. Для этого вызывается диалог открытия изображения. Если пользователь выберет файл и при этом таблица находится в режиме вставки (`dsInsert`) или редактирования (`dsEdit`) оба поля записи будут заполнены соответствующими данными.

**Листинг 5.2. Отправка данных в поле BLOB**

```
procedure TfrmMain.SetData;
begin
if (FileOpenDialog1.Execute) and
        ((ADOTable1.State=dsInsert) or (ADOTable1.State=dsEdit)) then
    begin
   (ADOTable1.FieldByName('PICTUREBLOB') as
    TBlobField).LoadFromFile(FileOpenDialog1.FileName);
    ADOTable1.FieldByName('PICTURENAME').AsString:=
                            ExtractFileName(FileOpenDialog1.FileName);
   end;
end;
```

Процедура `GetData()` предназначена для извлечения данных из текстового и BLOB поля. Название файла загружается в `Caption` компонента `StaticText1`. Если поле `Image` не пусто, то мы создаем поток `M:TMemoryStream`, загружаем в него данные из поля, и передаем их в компонент `Image1`. Обращаю внимание, на то что, перед передачей данных из потока его следует установить на нулевую позицию. В заключении освобождаем ресурсы, занимаемые потоком, вызвав метод `Free()`.

**Листинг 5.3. Получение данных из поля BLOB**

```
procedure TfrmMain.GetData;
var M : TMemoryStream;
begin
with ADOTable1 do
     begin
      StaticText1.Caption:=FieldByName('PICTURENAME').AsString;
      if (FieldByName('PICTUREBLOB') as TBlobField).IsNull=False then
       begin
         M:=TMemoryStream.Create;
         (FieldByName('PICTUREBLOB') as TBlobField).SaveToStream(M);
         M.Position:=0;
         Image1.Picture.Bitmap.LoadFromStream(M);
         M.Free;
       end else Image1.Picture.Bitmap.FreeImage;
    end;
end;
```

После того, как были созданы ключевые процедуры чтения и записи данных, переходим к описанию основных событий. Первое из них – событие `OnCreate()` главной формы проекта. В нем мы осуществим подключение компонента `ADOTable1` к базе данных, определим параметры диалога открытия изображений.

**Листинг 5.4. Создание формы**

```
procedure TfrmMain.FormCreate(Sender: TObject);
var s:string;
    FT:TFileTypeItem;
    const db='\blobdemo.accdb';
begin
{подключаемся к базе данных}
s:=ExtractFilePath(Application.ExeName)+db;
if FileAge(s)>0 then
  begin
   ADOTable1.ConnectionString:='Provider=Microsoft.ACE.OLEDB.12.0;
                               Data Source='+s+';Persist Security Info=False';
   ADOTable1.TableName:='PICTURETABLE';
   ADOTable1.Open;
   GetData;
   FT:=FileOpenDialog1.FileTypes.Add;
   FT.FileMask:='*.bmp';
   FT.DisplayName:='Растровый файл';
  end
  else
  begin
   Application.MessageBox(pChar('Не найден файл'+#13+S),'Ошибка',MB_OK);
   Application.Terminate;
  end;
end;
```

выберите кнопку `btnFirst`, отвечающую за перевод курсора на самую первую запись в таблице и опишите ее обработчик события `OnClick()`. Особенность события в том, что оно будет применяться для всех четырех кнопок перемещения. Если помните, кнопки перемещения по записям таблицы различаются свойством `tag`, `btnFirst` – 0, `btnPrior` – 1, `btnNext` – 2, `btnLast` – 3. Анализируя свойство `tag` в селекторе `Case` мы вызываем соответствующий метод таблицы.

**Листинг 5.5. Щелчок по кнопкам перемещения по строкам таблицы**

```
procedure TfrmMain.btnFirstClick(Sender: TObject);
begin
with ADOTable1 do
begin
{перемещаем курсор в соответствии со значением свойства tag кнопок}
     case (sender as TComponent).Tag of
        0: First;   //на первую запись
        1: Prior;   //на предыдущую запись
        2: Next;    //на следующую запись
        3: Last;    //на последнюю запись набора данных
     end;
     GetData;//получаем данные из таблицы
{с помощью методов BOF и EOF проверяем местоположение текущей записи}
     btnFirst.Enabled:=NOT(BOF);{кнопка отключается, если курсор на первой записи}
     btnPrior.Enabled:=NOT(BOF);
     btnNext.Enabled:=NOT(EOF);{кнопка отключается, если курсор на последней записи}
     btnLast.Enabled:=NOT(EOF);
end;
end;
```

Нам осталось описать действия добавления, редактирования и удаления записи.

**Листинг 5.6. Щелчки по кнопкам редактирования строк таблицы**

```
procedure TfrmMain.btnAppendClick(Sender: TObject);//добавление
begin
```

```
 ADOTable1.Append; {добавление новой записи в конец набора}
 SetData; {вызов выбора изображения}
 GetData; {передаем данные в компонент Image1 и StaticText1}
end;

procedure TfrmMain.btnEditClick(Sender: TObject);//редактирование
begin
 ADOTable1.Edit;{перевод таблицы в режим редактирования}
 SetData;
 GetData;
end;

procedure TfrmMain.btnDeleteClick(Sender: TObject);//удаление
begin
if MessageBox(frmMain.Handle,
              pWideChar('Удалить запись?'),
              pWideChar('Удаление'),
              MB_YESNO + MB_ICONQUESTION)=idYes then
    begin
      ADOTable1.Delete;                {удаление записи}
      Image1.Picture.Bitmap.FreeImage; {освобождаем ресурс картинки}
      StaticText1.Caption:='';         {очищаем заголовок метки}
      GetData;
    end;
end;
```

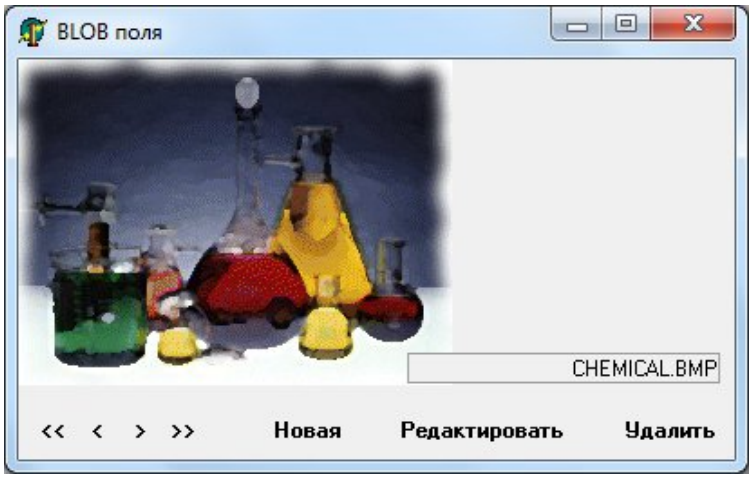



*Рис. 5.3. Внешний вид приложения*

## Задание

Внесите в свой проект БД следующие доработки:

1. Возможность сортировать данные по индексному полю.
2. Добавьте к одной из таблиц поле предназначенное для хранения фотографии, и разработайте форму для просмотра, ввода, редактирования и удаления данных из этого поля.

# 6. Разработка интерфейса клиентского приложения

*Вид занятия – практическое занятие*
*Время занятия – 2 часа*
*Программное обеспечение – среда Microsoft Access, C++ Builder (Delphi)*

## Источник данных – компонент TDataSource

Невизуальный элемент управления `TDataSource` минималист, у него всего 4 опубликованных свойства, пара методов и плюс три обработчика событий. Но, не смотря такую скромность, источник данных самым выдающимся образом решает следующие задачи:

Во-первых, источник данных незаменим при организации взаимодействия между компонентом доступа к набору данных (элементы `TADOTable`, `TSQLQuery`, `TIBStoredProc`, и т.п.) и элементами управления данными (`TDBNavigator`, `TDBEdit`, `TDBGrid`). Источник берет на себя ответственность за передачу информации в компоненты отображения данных и возвращает назад осуществленные пользователем изменения (рис. 6.1).

Во-вторых, компонент `TDataSource` способен организовать связь между главным и подчиненным наборами данных. Этот вопрос мы уже рассматривали в лабораторных работах 13 и 14.

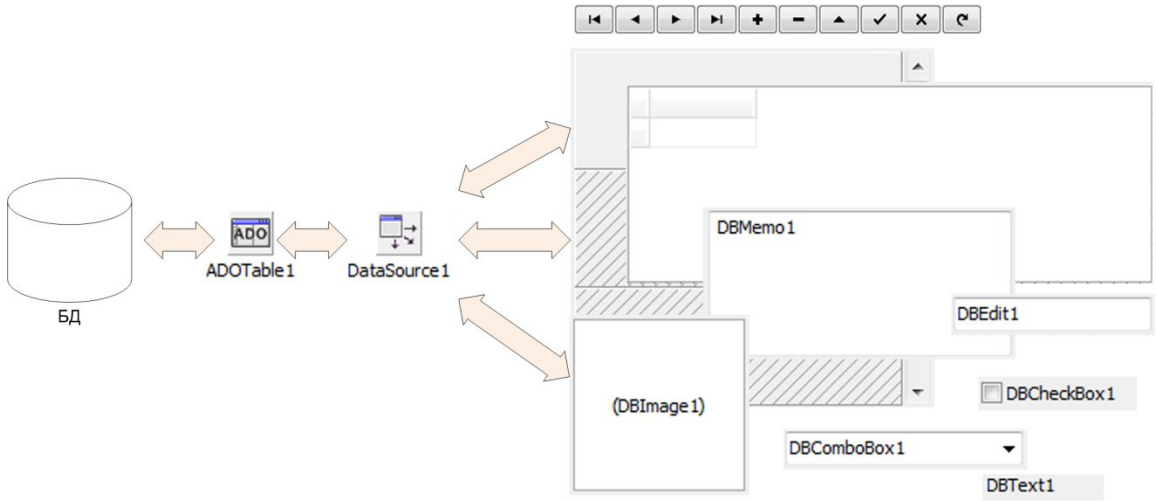



*Рис. 6.1. Место компонента TDataSource в приложении БД*

Ключевым свойством компонента по праву считается

```
property DataSet: TDataSet;
```

Благодаря этому свойству и поддерживается связь с набором данных. Свойство доступно как в период разработки приложения, так и во время выполнения. Во время разработки инспектор объектов предоставит программисту список из всех доступных в приложении потомков `TDataSet` (таблиц, запросов, хранимых процедур), нам лишь останется выбрать нужный из них.

Для проверки, связан ли в текущий момент источник данных с набором `DataSet`, предназначена функция

```
function IsLinkedTo(DataSet: TDataSet): Boolean;
```

Контроль состояния набора данных, к которому в данный момент присоединен `DataSource`, осуществляет с помощью свойства:

```
property State: TDataSetState; //только для чтения
```

Оставшиеся два свойства имеют прямое отношение к элементам отображения данных. Первое из них приспособлено для их централизованного включения/отключения:

```
property Enabled: Boolean;
```

Второе свойство (в состоянии `true`) обеспечивает автоматический перевод в режим редактирования связанного с `DataSource` компонента – набора данных

```
property AutoEdit: Boolean;
```

Такой переход осуществляется при внесении пользователем изменений в элементах отображения данных. Еще один способ перехода в редактирование сводится к вызову метода:

```
procedure Edit;
```

При любом изменении в данных (например: при переходе от одной записи к другой или в момент модификации записи) вызывается обработчик события:

```
property OnDataChange: TDataChangeEvent;
type TDataChangeEvent = procedure(Sender: TObject; Field: TField) of object;
```

здесь параметр `Field` указывает, в каком из полей произошли изменения. Но если одновременно модифицировалось два (и более) поля параметр окажется пустым – `nil`.

При каждом изменении состояния связанного набора данных происходит событие:

```
property OnStateChange: TNotifyEvent;
```

Перед передачей в базу сделанных пользователем модификаций вызывается:

```
property OnUpdateData: TNotifyEvent;
```

Данный обработчик события может использоваться для проверки правильности данных перед их сохранением.

## Общие черты компонентов отображения данных

Практически все компоненты отображения данных построены на базе стандартных элементов управления и являются их логическим развитием. Соединение компонента с набором данных реализуется при прямом посредничестве элемента управления `TDataSource`. Поэтому, первой и наиболее важной объединяющей чертой всех элементов управления, предназначенных для работы с данными, является не что иное, как источник данных. Для подключения к источнику потребуется передать имя необходимого компонента `TDataSource` в свойство

```
property DataSource: TDataSource;
```

Компоненты отображения данных, специализирующиеся на предоставлении доступа к отдельному полю набора данных (`TDBText`, `TDBEdit`, `TDBMemo`, и т.д.) для подключения к столбцу потребуют передать его имя в свойство

```
property DataField: String;
```

Если соединение осуществилось без проблем, то в свойстве

```
property Field: TField; //только для чтения
```

мы обнаружим ссылку на соответствующий обслуживаемому столбцу таблицы экземпляр объекта-поля `TField`.

Все компоненты, способные не просто отображать, но еще и редактировать данные, переводятся в режим только для чтения свойством

```
property ReadOnly: Boolean;
```

## Сетка базы данных – компонент TDBGrid

Компонент `TDBGrid` один из наиболее часто используемых элементов управления для отображения и редактирования данных. Его основное преимущество над всеми своими коллегами заключается в том, что сетка способна вывести на экран сразу несколько строк и столбцов набора данных, все остальные компоненты (за исключением `TDBCtrlGrid`) ориентированы на работу только с текущей записью и одним единственным полем.

Каждая подключенная к активному набору данных сетка знает, содержимое каких именно полей предстоит ей обслуживать. Этой информацией сетка способна поделиться и с нами. Свойство

```
property Fields[Index: Integer]: TField; //только для чтения
```

предоставляет доступ ко всем объектам `TField` набора данных, как к элементам массива. Общее количество элементов в массиве хранится в свойстве

```
property FieldCount: Integer; //только для чтения
```

Объект-поле, соответствующий выделенной в сетке ячейке

```
property SelectedField: TField;
```

Индекс выбранной колонки находится в свойстве

```
property SelectedIndex: Integer;
```

Свойства `SelectedField` и `SelectedIndex` не только информационные. С их помощью вы получаете возможность выбирать требуемую колонку в коде программы.

Некоторые особенности поведения сетки определяются свойством

```
property Options: TDBGridOptions;
TDBGridOptions = set of TDBGridOption;
```

*Таблица 6.1. Описание опций сетки*

| Значение | Описание |
|---|---|
| `dgEditing` | Сетка допускает редактирование своего содержимого, но при условии, что в состав опций не входит значение `dgRowSelect`. |
| `dgAlwaysShowEditor` | Ячейки сетки постоянно готовы к редактированию своих данных, пользователю даже нет необходимости нажимать клавиши Enter или F2. |
| `dgTitles` | Самая верхняя строка сетки отводится для показа заголовков колонок. |
| `dgIndicator` | Напротив текущей записи появляется указатель. |
| `dgColumnResize` | Разрешается перестановка местами и изменение ширины колонок. |
| `dgColLines` | Колонки сетки разделяются линией. |
| `dgRowLines` | Строки сетки разделяются линией. |
| `dgTabs` | Нажатие клавиш `Tab` и `Shift+Tab` позволяет перемещаться между ячейками сетки. |
| `dgRowSelect` | Пользователь теряет возможность выбрать отдельную ячейку сетки. Вместо этого в сетке выделяется вся строка целиком. |
| `dgAlwaysShowSelection` | Сетка продолжает выделять цветом выбранные пользователем ячейки, даже если она сама потеряла фокус ввода. |
| `dgConfirmDelete` | Перед удалением строки из сетки вызывается диалоговое окно предлагающее подтвердить эту операцию. |
| `dgCancelOnExit` | Автоматически отменяет несохраненные изменения в данных при потере сеткой фокуса ввода. |
| `dgMultiSelect` | Допускается одновременный выбор более одной строки сетки. |

В таблице 6.2 представлены шесть событий компонента `TDBGrid` обеспечивающих взаимодействие пользователя и элемента управления.

*Таблица 6.2. События сетки TDBGrid*

| Событие | Описание |
|---|---|
| `property OnColEnter: TNotifyEvent;` | Событие генерируется в момент получения фокуса ввода одной из колонок сетки. |
| `property OnColExit: TNotifyEvent;` | Одна из колонок сетки теряет фокус ввода. |
| `type TDBGridClickEvent = procedure (Column: TColumn) of object;`<br>`property OnCellClick: TDBGridClickEvent;` | Событие вызывается при щелчке кнопкой мышки в ячейке сетки. Здесь `Column` – колонка, которой принадлежит данная ячейка. |
| `type TDBGridClickEvent = procedure (Column: TColumn) of object;`<br>`property OnTitleClick: TDBGridClickEvent;` | Щелчок мышкой по заголовку колонки. |
| `type TMovedEvent = procedure (Sender: TObject; FromIndex, ToIndex: Longint) of object;`<br>`property OnColumnMoved: TMovedEvent;` | Событие вызывается после перемещения колонки из позиции `FromIndex` в позицию `ToIndex`. |
| `property OnEditButtonClick: TNotifyEvent;` | Щелчок по кнопке внутри ячейки сетки. |

## Статический текст – компонент TDBText

Элемент управления TDBText предназначен только для отображения текущего значения поля и не имеет никаких навыков по редактированию связанных с ним данных. Для подключения компонента к набору данных достаточно выполнить стандартные операции определения источника данных (свойство DataSource) и указать какое именно поле нас интересует (свойство DataField).

Для обеспечения автоподстройки размеров компонента при выводе текста с различным количеством символов установите в true свойство:

```
property AutoSize: Boolean;
```

Если текстовые данные настолько необъятны, что ни каким образом не желают помещаться в одну строку, следует перевести в true свойство:

```
property WordWrap: Boolean;//по умолчанию false
```

В этом случае элемент управления вспомнит про свои навыки вывода информации в несколько строк.

## Строка ввода БД – компонент TDBEdit

Компонент класса TDBEdit представляет собой строку ввода способную не только отображать но и редактировать текущее значение поля.

## Многострочный текстовый редактор БД – TDBMemo

Элемент управления TDBMemo спроектирован для работы с текстовыми полями больших размеров или с BLOB полями, специализирующихся на хранении текстовых данных.

## Изображение БД – компонент TDBImage

Элемент TDBImage специализируется на обработке графических данных, содержащихся в BLOB полях. Ключевое свойство компонента:

```
property Picture: TPicture;
```

Оно инкапсулирует весь функционал для работы с растровой графикой и метафайлами. Для ускорения работы с BLOB полями в элементе управления объявлено свойство:

```
property AutoDisplay: Boolean;
```

Задача свойства – включение/отключение автозагрузки данных из поля. Если свойство установлено в false, то для инициализации процесса загрузки изображения в компонент вызывают метод
`procedure LoadPicture;`

## Список БД – TDBListBox

Список TDBListBox отображает текущее значение поля (при условии совпадения его с одним из элементов списка) и позволяет пользователю передавать в поле таблицы одно из заранее подготовленных текстовых значений. Перечень допустимых значений храниться в унаследованном свойстве:

```
property Items : TStrings;
```

Заполнение списка строк может осуществляться как в период разработки проекта во встроенном редакторе, так и во время работы приложения.

## Комбинированный список БД – TDBComboBox

Комбинированный список выбора TDBComboBox призван решать задачи по выбору значения из списка. Текущее значение связанного с компонентом поля отображается в строке редактирования элемента управления. При желании пользователь может передать в базу данных одно из значений, хранящихся в списке компонента. Список значений содержится в свойстве Items, индекс текущего значения – ItemIndex.

## Флажок БД – TDBCheckBox

Элемент позволяет пользователю согласиться или отказываться от условия. Хотя в первую очередь флажок нацелен на взаимодействие с полями данных типа Boolean, но его также можно научить передавать в базу текстовые значения. Эта возможность осуществляется с помощью двух свойств:

```
property ValueChecked: String;      //значение на включение флажка
property ValueUnchecked: String;    //значение на отключение флажка
```

по умолчанию содержащими значения True и False соответственно. Теперь, если в присоединенном к компоненту текстовом поле базы данных обнаруживается значение True, то этот факт найдет свое

отражение – в виде “галочки” на поверхности элемента. Значение `False` наоборот – прогонит всех галок. Компонент способен отслеживать одновременно несколько значений, но в этом случае значения должны быть разделены точкой с запятой

```
DBCheckBox1.ValueChecked := 'Да;Вкл';
DBCheckBox1. ValueUnchecked := 'Нет;выкл';
```

Для управления компонентом из программы потребуются свойства:

```
property Checked: Boolean;
```

и

```
type TCheckBoxState = (cbUnchecked, cbChecked, cbGrayed);
property State: TCheckBoxState;
```

## Радиогруппа БД – TDBRadioGroup

Компонент реализует группу кнопок выбора, состояние которых определяется значением текущего поля присоединенного набора данных. Одновременно может быть отмечена только одна из кнопок группы. При выборе пользователем кнопки, связанное с ним значение передается в базу данных.

Список заголовков кнопок определяется содержимым свойства `Items`:

```
property Items: TStrings;
```

Перечень возможных значений в свою очередь заполняется в свойстве:

```
property Values: TStrings;
```

Оба свойства доступны во время разработки и во время выполнения программы. Отдельно подчеркну, что количество строк в обоих списках должно быть одинаковым.

Текущее значение связанного поля мы можем выяснить из свойства:

```
property Value: String;
```

Индекс выбранной кнопки находится в свойстве:

```
property ItemIndex: Integer;
```

## Компонент – TDBCtrlGrid

Компонент `TDBCtrlGrid` представляет собой набор динамически создаваемых панелей. После открытия, связанного с ним набора данных компонент самостоятельно создает несколько панелей. Каждая панель ассоциируется с отдельной записью из набора данных. Главная черта `TDBCtrlGrid` в том, что этот элемент управления может являться владельцем других компонентов, предназначенных для работы с БД (рис. 6.2).

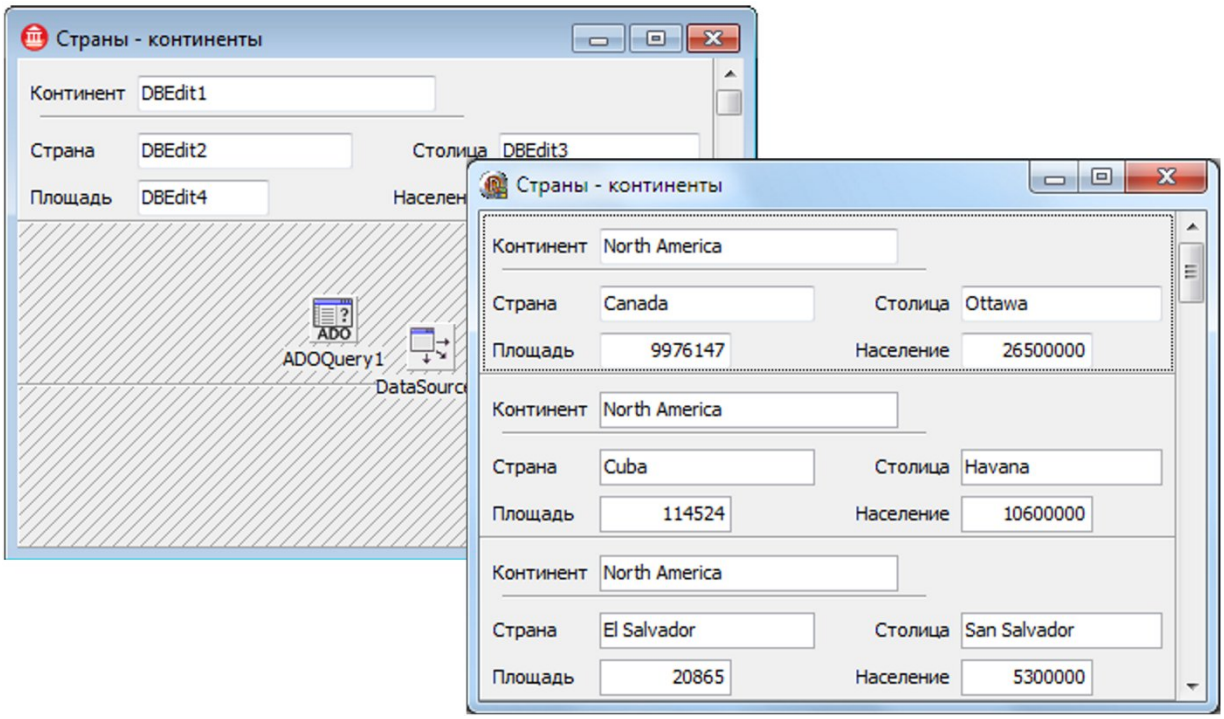



*Рис. 6.2. Работа с компонентом `TDBCtrlGrid`*

Порядок следования записей в компоненте `TDBCtrlGrid` определен свойством

```
property Orientation: TDBCtrlGridOrientation;
type TDBCtrlGridOrientation = (goVertical, goHorizontal);
```

По умолчанию свойство принимает значение `goVertical`, что соответствует последовательности сверху-вниз.

По умолчанию панели компонента размещаются в одном столбце. Для изменения этой традиции требуется настроить свойство

```
property ColCount: Integer;
```

Количество рядов видимых в клиентской области элемента управления определяется свойством:

```
property RowCount: Integer;
```

За геометрические размеры панелей отвечают свойства:

```
property PanelHeight: Integer;
property PanelWidth: Integer;
```

Индекс панели текущей записи хранится в свойстве:

```
property PanelIndex: Integer;
```

Это же свойство можно применять при необходимости переместиться к панели с определенным номером.

## Навигатор – TDBNavigator

Навигатор по набору данных визуально представляет собой группу кнопок (рисунок 6.3). Компонент инкапсулирует в себе золотой минимум методов достаточных для создания элементарного приложения работающего с таблицей базы данных. Сразу после подключения к источнику данных при помощи свойства `DataSource` элемент управления сможет обеспечить перемещение по записям в наборе, вставку, редактирование и удаление записи – для этого достаточно щелкнуть по соответствующей кнопке. По умолчанию компонент содержит 10 кнопок, однако по желанию программиста вполне можно отказаться от показа ненужных. Для этого надо настроить свойство

```
property VisibleButtons: TButtonSet;
type TButtonSet = set of TNavigateBtn;
type TNavigateBtn = (nbFirst, nbPrior, nbNext, nbLast, nbInsert, nbDelete, nbEdit, nbPost,
nbCancel, nbRefresh);
```

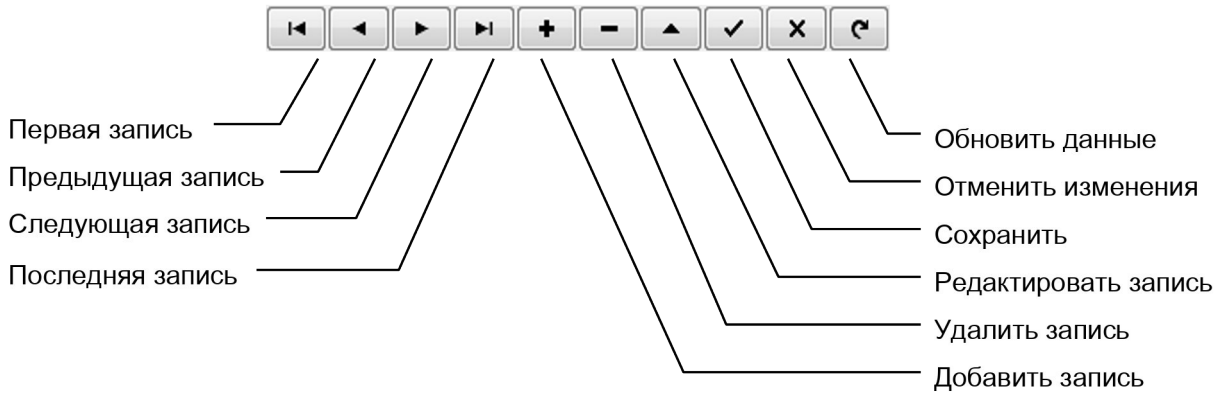



*Рис. 6.3. Элемент управления `TDBNavigator`*

Как и практически все визуальные элементы управления VCL навигатор обладает элементарным обработчиком события `OnClick()`. С той только разницей, что он способен различать по какой из кнопок был произведен щелчок.

```
property OnClick: ENavClick;
type ENavClick = procedure (Sender: TObject; Button: TNavigateBtn) of object;
```

для этого в событии объявлен дополнительный параметр – `Button`.

## Задание

Используя полученные на занятии знания, разработайте интерфейс для вашего приложения БД. В приложении должны быть обязательно задействованы компоненты `TDataSource`, `TDBGrid`, `TDBCtrlGrid`, `TDBEdit`, `TDBListBox` (или `TDBComboBox`), `TDBCheckBox`, `TDBRadioGroup` и другие компоненты на ваш выбор.

# 7. Создание БД, таблиц и представлений в Microsoft SQL Server

*Вид занятия – практическое занятие*
*Время занятия – 2 часа*
*Программное обеспечение – Microsoft SQL Server*

К настоящему времени насчитывается не боле полутора десятков современных систем управления базами данных. Среди них наибольшей популярностью пользуются серверы баз данных Oracle, Informix, Interbase и Microsoft SQL Server.

## Создание базы данных

В терминологии Microsoft SQL Server (MS SQL) база данных это логический объект, в которые MS SQL помещает таблицы, индексы и другие объекты. Физически база данных располагается в одном (или нескольких файлах).

Для создания базы данных можно воспользоваться программой SQL Server Enterprise Manager:

1. Запустите SQL Server Enterprise Manager. выберите сервер, с которым вы станете работать и выделите папку “Databases”.
2. выберите команду “Действие – Создать”.
3. В окне “Database properties” определите параметры новой БД (как минимум название БД, допустим “Demo”).
4. Щелкните по кнопке OK.

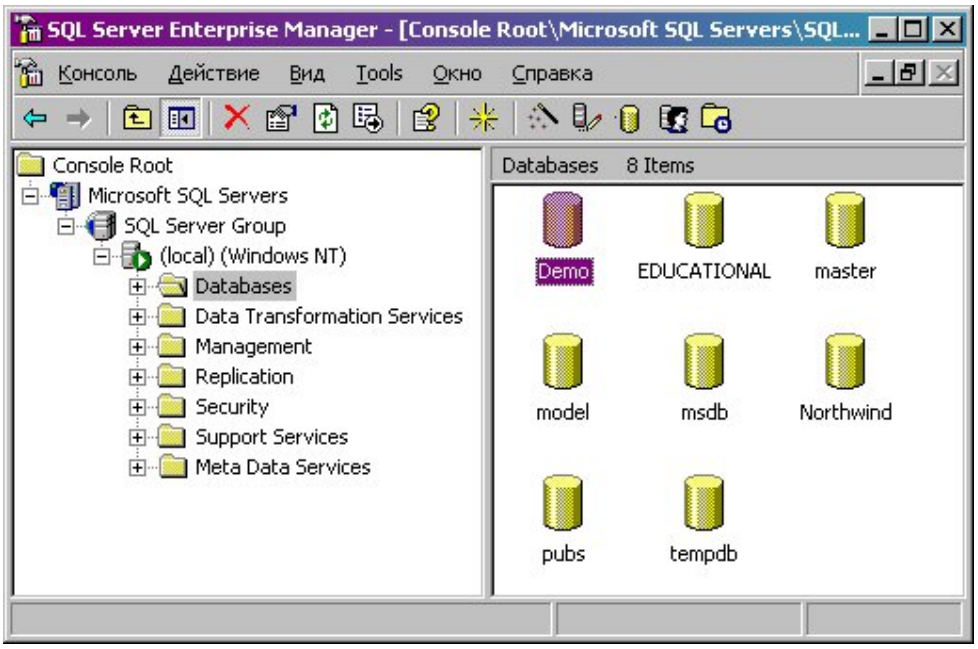



*Рис. 7.1. Окно SQL Server Enterprise Manager с созданной БД “Demo”*

В результате MS SQL создаст, пустую БД (рис. 17.1). Такого же результата можно добиться, воспользовавшись командой SQL “Create database”, рассмотренной на 8-м занятии.

Для внесения изменений в параметры вновь созданной БД достаточно выделить пиктограмму БД и в контекстном меню воспользоваться командой Action – Properties (Действие – Свойства).

## Удаление базы данных

При удалении БД вместе с ней удаляются все принадлежащие ее объекты, и освобождается пространство, занимаемое БД на диске. Для удаления БД выберите команду “Действие – Удалить”.

Альтернативным способом “борьбы” с ненужной БД является использование команды SQL Drop Database.

## Создание таблиц

Для создания таблиц в MS SQL используется программа SQL Server Enterprise Manager или команда Create Database. Рассмотрим процесс создания таблиц на примере схемы данных, представленной на рисунке 17.2.

1

| Таблица континентов - Continent | | | | |
|---|---|---|---|---|
| *Название* | *Тип данных* | *Ключ* | *NOT NULL* | *Уникальность* |
| Continent_ID | INT | PK | ☑ | ☑ |
| Continent | VARCHAR(15) | | ☑ | ☑ |

M

| Таблица стран - Country | | | | |
|---|---|---|---|---|
| *Название* | *Тип данных* | *Ключ* | *NOT NULL* | *Уникальность* |
| Country_ID | INT | PK | ☑ | ☑ |
| Continent_ID | INT | FK | ☑ | |
| Country | VARCHAR(20) | | ☑ | ☑ |
| City | VARCHAR(20) | | ☑ | ☑ |
| Population | INT | | | |
| Area | INT | | | |

*Рис. 7.2. Схема данных БД "Demo"*

Для создания новой таблицы в SQL Server Enterprise Manager выберите нужный сервер, а затем откройте папку Databases и БД "Demo" (которой будет принадлежать таблица). Затем выполните команду: "Действие – Создать – Таблица". В результате на экран компьютера будет выведено окно редактора таблицы (рис. 7.3).

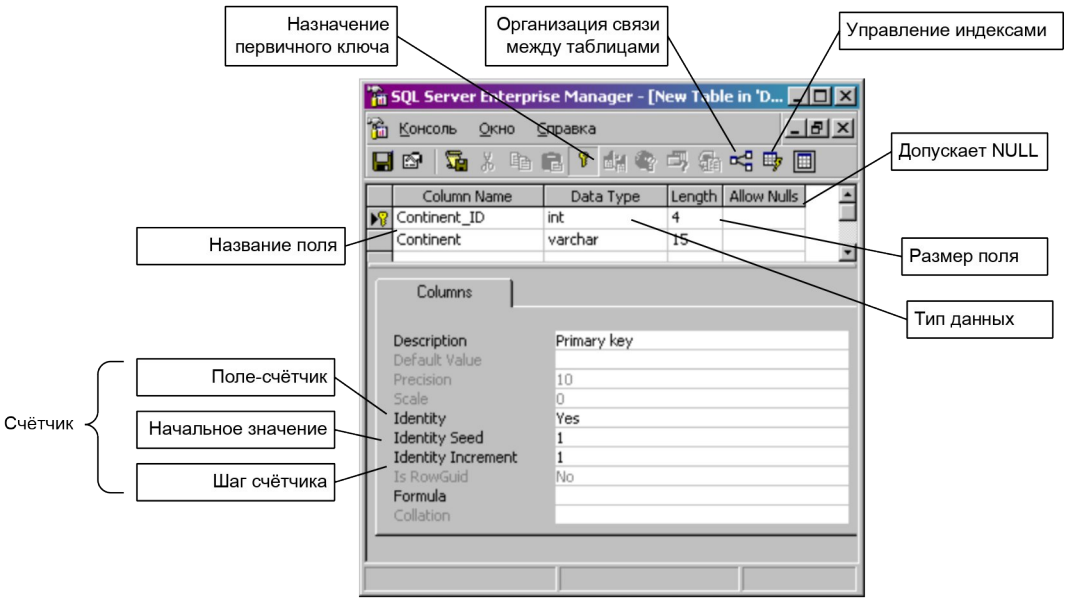



*Рис. 7.3. Окно создания таблицы*

Завершив описание будущей таблицы, сохраните изменения. Для этого нажмите на кнопку с изображением дискеты. Вам останется ответить на вопрос об имени создаваемой таблицы.

При создании таблицы Country необходимо позаботиться об организации связи один ко многим между таблицей стран и континентов. Для этого в состав полей таблицы Country введено поле Continent_ID, которое станет содержать внешний ключ (Foreign Key). Для определения связей между главной и подчиненной таблицами на панели инструментов требуется нажать кнопку "Manage Relationships…". На рисунке 7.4. показано окно редактора свойств таблицы с активной страницей "Relationships".

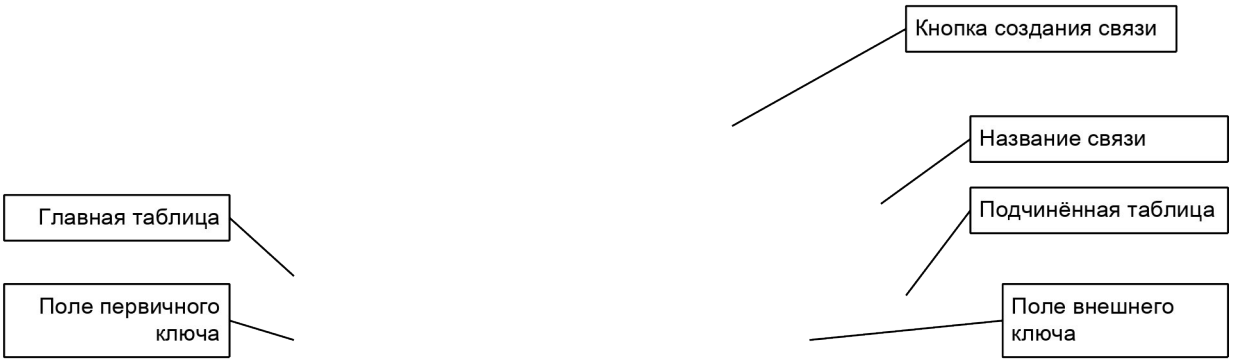



*Рис. 7.4. Страница определения связи между таблицами*

Ограничения Foreign Key и Primary Key гарантируют, что соответствующие строки связанных таблиц всегда останутся согласованными. Помимо ограничения внешнего ключа MS SQL сервер позволяет накладывать на поля таблицы дополнительные ограничения:

1. Ограничение поддержки уникальности (Unique). Применяется к любому столбцу таблицы с целью предотвращения ввода в него повторяющихся значений.
2. Ограничение проверки данных (Check). Налагает дополнительные условия на значения, которые можно ввести в один или несколько столбцов таблицы. Одним из примеров ограничения может служить диапазон вводимых значений.
3. Значение по умолчанию (Default). Применяется для того, чтобы автоматически вводить в столбец некоторое значение.

## Пример создания таблиц средствами Transact SQL

Все проделанные нами операции по созданию таблиц можно представить в виде нескольких строк команд на языке Transact SQL (диалект SQL, используемый в MS SQL).

**Листинг 7.1. Код TSQL для создания таблиц для СУБД MS SQL**

```
/*Создание таблицы континентов*/
 CREATE TABLE Continent
    (Continent_ID INTEGER IDENTITY (1, 1) PRIMARY KEY,
    Continent VARCHAR(15) NOT NULL UNIQUE)
/*Создание таблицы стран*/
CREATE TABLE Country
 (Country_ID INTEGER IDENTITY (1, 1) PRIMARY KEY,
 Continent_ID INTEGER,
 Country VARCHAR(20) NOT NULL UNIQUE,
 City VARCHAR(20) NOT NULL UNIQUE,
 Population INT,
 Area INT,
 CHECK (Population>0), CHECK (Area>0),
FOREIGN KEY (Continent_ID) REFERENCES Continent (Continent_ID));
```

## Создание представлений

Одним из способов показа информации в MS SQL является представление.

> Представление (view) это хранимое определение оператора SELECT.

Чтобы создать новое представление необходимо сделать следующие шаги:

1. Запустите SQL Server Enterprise Manager.
2. Находясь в текущей базе данных, выберите команду “Создать – View…”.
3. В окне “New View” выберите таблицы и поля, включаемые в представление (рис. 17.5).
4. Закройте окно.
5. Назначьте имя нового представления, например “vn_Continent_Country”

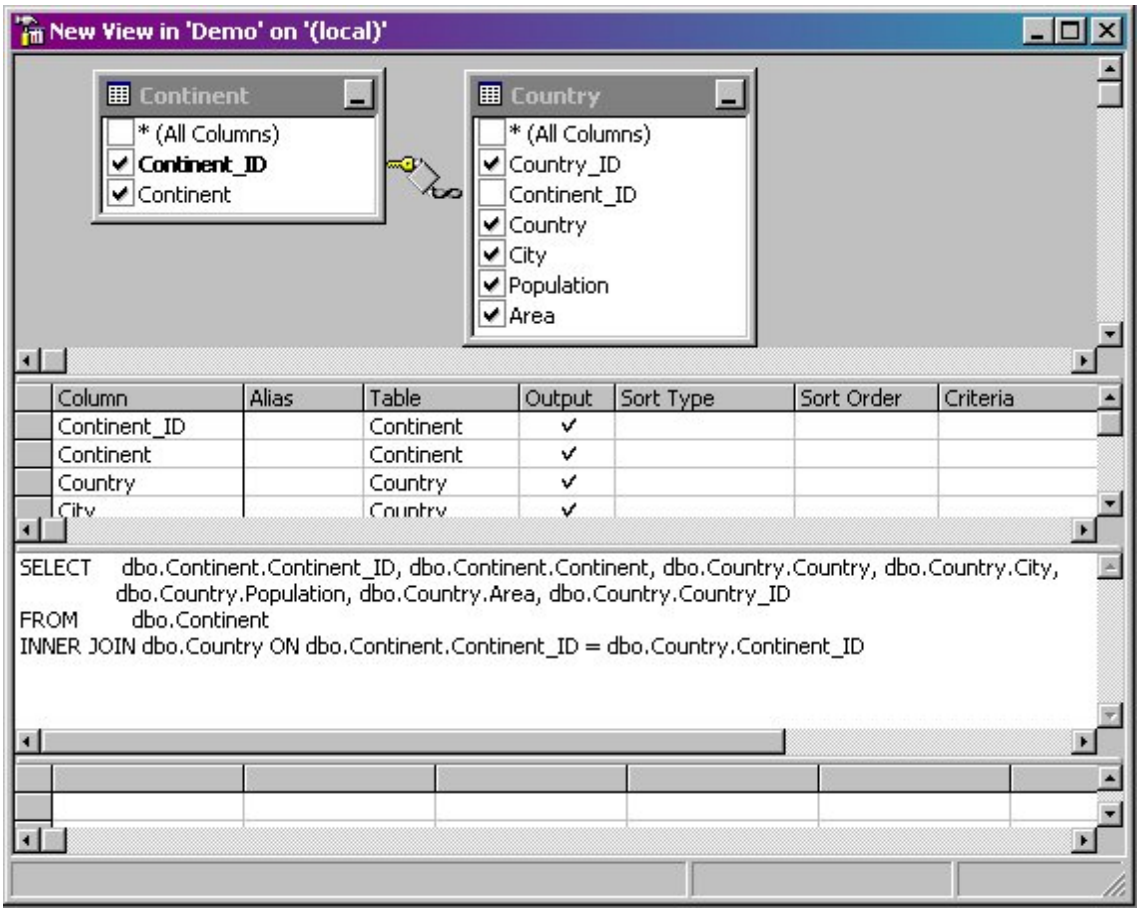



*Рис. 7.5. Создание представления с помощью мастера*

Этого же результата можно добиться, выполнив следующую команду на языке SQL

**Листинг 7.2. Инструкция TSQL для создания представления для СУБД MS SQL**

```
CREATE VIEW vn_Continent_Country
AS
SELECT Continent.Continent_ID, Continent.Continent, Country.Country,
 Country.City, Country.Population, Country.Area, Country.Country_ID
FROM Continent
INNER JOIN Country ON Continent.Continent_ID = Country.Continent_ID
```

## Задание

В соответствии со схемой данных разработанной на 4-й лабораторной работе создайте свой проект БД средствами Microsoft SQL Server.

# 8. Введение в Microsoft Transact SQL

*Вид занятия – практическое занятие*
*Время занятия – 2 часа*
*Программное обеспечение – Microsoft SQL Server*

Каждая из современных реляционных СУБД внесла свои дополнения в стандартизированный еще в прошлом веке язык структурированных запросов SQL-92. Это занятие посвящено некоторым особенностями одного из самых известных диалектов SQL – языка Transact-SQL.

В рамках Transact-SQL практически полностью поддержан язык SQL-92, кроме того, Transact-SQL обладает несколькими сотнями новых функций и хранимых процедур.

## Определение и использование переменных

Переменные в Transact-SQL могут быть локальными или глобальными. Локальные переменные определяются с помощью оператора `DECLARE`. Перед именем переменной всегда ставится символ "`@`", а перед именем глобальной два таких символа.

```
DECLARE @имя_переменной тип_данных
```

Например

```
DECLARE @S CHAR(20), @I INTEGER
```

Для вывода переменной на экран можно использовать команду `PRINT`.

Для преобразования типа данных применяются функции `CONVERT` или `CAST`.

Для присвоения переменной значения применяется оператор `SET` или `SELECT`.

В MS SQL объявлен ряд глобальных переменных, которые вы сможете применять в своей работе. Некоторые из них представлены в таблице 8.1.

*Таблица 8.1. Основные глобальные переменные в TSQL*

| Название | Описание |
|---|---|
| `@@CONNECTIONS` | Общее количество соединений с сервером или попыток входа в систему |
| `@@ERROR` | Номер последней системной ошибки; в случае отсутствия ошибок это значение равно 0 |
| `@@FETCH_STATUS` | Состояние последнего выполненного оператора FETCH |
| `@@IDENTITY` | Значение последней вставленной в столбец IDENTITY величины |
| `@SIDLE` | Суммарное время простоя центрального процессора сервера. |
| `@@IO_BUSY` | Суммарное время, затраченное сервером на выполнение операций ввода/вывода. |
| `@@LANGUAGE` | Название используемого в данный момент языка |
| `@@LOCK_TIMEOUT` | Величина тайм-аута для операций блокировки (в миллисекундах) |
| `@@MAX_CONNECTIONS` | Максимальное число одновременных подключений. |
| `@@MAX_PRECISION` | Максимальный уровень точности для числовых и десятичных типов данных |
| `@@NESTLEVEL` | Номер текущего уровня вложения при вызове подпрограмм |
| `@@OPTIONS` | Возвращает установки текущих опций, заданных с помощью оператора SET |
| `@@PROCID` | Идентификатор текущей хранимой процедуры |
| `@@REMSERVER` | Имя удаленного сервера |
| `@@ROWCOUNT` | Количество строк, на которые повлиял последний запрос |
| `@@TRANCOUNT` | Общее число активных транзакции для текущего пользователя |

## Операторы управления Transact-SQL

В составе Transact-SQL имеется несколько операторов, которые применяются для изменения порядка выполнения команд внутри запроса SQL.

Ключевые слова IF…ELSE используются для реализации аналога условного оператора, знакомого Вам из традиционных языков программирования:

```
IF выражение
        Операторы
[ELSE]
        [IF выражение]
                Операторы
```

Ключевые слова BEGIN и END используются для обозначения набора операторов SQL, выполняемых как единое целое

```
BEGIN
        операторы
END
```

Ключевое слово WHILE используется для определения условия, на основании которого выполняется один или несколько операторов SQL, пока это условие истинно (равно TRUE):

```
WHILE <булево выражение>
<операторы SQL>
```

Внутри конструкции WHILE могут применяться операторы BREAK и CONTINUE, которые соответственно предназначены для прерывания работы конструкции или для продолжения ее выполнения.

**Листинг 8.1. Организация цикла WHILE на языке TSQL**

```
DECLARE @X INT, @Y TINYINT
SET @X=1 SET @Y=1
WHILE @X<5
BEGIN
        PRINT 'X<5'
        SET @X=@X+1
        SET @Y=@Y+1
IF @Y=2
        BEGIN
                PRINT 'Y=2'
                BREAK
        END
END
PRINT 'ЦИКЛ ЗАВЕРШЕН'
```

## Базовые функции Transact-SQL

В таблицах 8.1 - 8.4 представлены наиболее часто применяемые функции transact-SQL.

*Таблица 8.2. Основные математические функции в TSQL*

| Функция | Описание | |
|---|---|---|
| ACOS | Арккосинус числа | Значение угла выражено в радианах |
| ASIN | Арксинус числа | |
| ATAN | Арктангенс числа | |
| ATAN2 | Арктангенс числа | |
| COS | Тригонометрический косинус угла | |
| COT | Тригонометрический котангенс угла | |
| SIN | Тригонометрический синус угла | |
| TAN | Тригонометрический тангенс угла | |
| DEGREES | Преобразование радиан в градусы | |

| | |
|---|---|
| RADIANS | Преобразование градусов в радианы |
| CEILING | Наименьшее целое значение, которое больше или равно значению выражения |
| FLOOR | Наибольшее целое значение, которое меньше или равно значению выражения |
| EXP | Экспонента значения выражения |
| LOG | Натуральный логарифм значения выражения |
| LOG 10 | Десятичный логарифм значения выражения |
| PI() | Значение числа ПИ |
| POWER | Значение выражения в степени y |
| ABS | Абсолютное значение выражения |
| RAND | Случайное действительное число, находящееся в промежутке от нуля до единицы либо до значения, указанного в необязательном аргументе integer expression |
| ROUND | Округление значения с точностью до integer |
| SIGN | Возвращает значение 1, 0 или -1, в зависимости от знака выражения. Имеет тот же тип, что и исходное выражение |
| SQRT | Квадратный корень из значения выражения |

*Таблица 8.3. Основные функции для работы со строками в TSQL*

| **Функция** | **Описание** |
|---|---|
| CHAR | Преобразует ASCII код в символ |
| LOWER | Преобразует прописные буквы в строчные |
| UPPER | Преобразует строчные буквы в прописные |
| LTRIM | Удаляет из строки начальные пробелы |
| RTRIM | Удаляет из строки конечные пробелы |
| CHARINDEX | Осуществляет поиск подстроки в строке |
| STUFF | Вставляет подстроку в строку |
| SUBSTRING | Извлекает из строки подстроку |
| (+) | Сложение строк |

*Таблица 8.4. Основные функции для работы с датой в TSQL*

| **Функция** | **Описание** |
|---|---|
| DATENAME | Возвращает часть даты в виде символьной строки |
| GETFATE | Возвращает текущее значение даты/времени |
| DATEADD | Добавляет к дате значение |
| DATEDIFF | Возвращает разницу между двумя датами |

*Таблица 8.5. Основные системные функции в TSQL*

| **Функция** | **Описание** |
|---|---|
| HOST_NAME() | Имя компьютера-сервера |
| HOST_ID() | Идентификационный номер компьютера-сервера |
| SUSER_ID | Идентификатор пользователя, назначаемый при входе в систему |
| SUSER_NAME | Имя пользователя, используемое для входа в систему |
| USER_ID | Идентификационный номер пользователя, назначаемый при подключении к базе данных |
| USER_NAME | Имя пользователя, используемое для подключения к базе данных |

| | |
|---|---|
| DB_NAME | Имя базы данных |
| DB_ID | Идентификационный номер базы данных |
| OBJECT_ID | Идентификационный номер объекта базы данных |
| OBJECT_NAME | Имя объекта базы данных |
| INDEX_COL | Имя индексного столбца |
| COL_LENGTH | Длина столбца |
| COL_NAME | Имя столбца |
| DATALENGTH | Реальная длина выражения для некоторого типа данных |
| COALESCE | Возвращает первое ненулевое выражение (не равное NULL) |
| ISNULL | Заменяет нулевое значение выражения (NULL) заданным значением |
| NULLIF | Возвращает элемент NULL, если оба выражения идентичны |

### Функции преобразования типов данных

Для приведения типов данных предназначены функции CONVERT и CAST. Синтаксис этих функций следующий

```
CONVERT (тип данных [ ( длина) ] , выражение [ , стиль ] )
CAST (выражение AS тип данных)
```

Пример работы с функциями предложен в листинге 8.2.

**Листинг 8.2. Преобразование типов данных**

```
DECLARE @X decimal (5, 2)
SET @X = 193.57
SELECT CAST(CAST(@X AS varbinary(20)) AS decimal(10,5))
SELECT CONVERT(decimal(10,5), CONVERT(varbinary(20), @X))
```

## Хранимые процедуры

Хранимые процедуры представляют собой подпрограммы, выполняемые на сервере. Эти подпрограммы могут запускаться из клиентского приложения, из других процедур и из триггеров.

В хранимую процедуру можно передавать значения, а они, в свою очередь, способны возвращать результаты своей работы.

К основным преимуществам хранимых процедур стоит отнести следующее:

- Производительность.
- Преимущества при разработке систем клиент-сервер.
- Безопасность.
- Поддержание дополнительных бизнес-правил на ввод данных.

Для создания хранимой процедуры применяется команда CREATE PROCEDURE. Вашему вниманию предложен код хранимой процедуры позволяющей добавлять в таблицу континентов новый материк.

**Листинг 8.3. Пример хранимой процедуры TSQL**

```
CREATE PROCEDURE Continent_Insert (@Continent CHAR(15), @Continent_ID INT OUTPUT)
AS
IF NOT EXISTS (Select * FROM Continent WHERE Continent=@Continent)
BEGIN
INSERT INTO Continent (Continent) VALUES (@Continent)
SET @CONTINENT_ID=@@IDENTITY
END ELSE @CONTINENT_ID=-1
```

Процедура с именем “Continent_Insert” обладает двумя параметрами: @Continent (в него передается название материка) и выходной параметр @Continent_ID (он возвращает значение первичного ключа вновь вставленной записи). Перед добавлением новой строки процедура проверяет нет ли в таблице континентов материка с названием @Continent, и только после этого добавляет новую запись.

## Триггеры

Сервер БД способен осуществлять контроль за логикой работы приложения и прохождении данных по таблицам. Один из мощнейших инструментов такого рода называется триггером.

> **Триггер** – специальный тип процедуры, которая выполняется при вставке, модификации или удалении данных в некоторой таблице. Триггеры запускаются в самую последнюю очередь, и если работа триггера завершается аварийно, то информация в БД не обновляется.

Синтаксическая конструкция описания триггера выглядит следующим образом:

```
CREATE TRIGGER имя_триггера
ON имя_таблицы
FOR [INSERT, UPDATE, DELETE]
AS Операторы
```

По сути, триггер – это событие, возникающее после вставки (*insert*), редактирования (*update*) или удаления записей (*delete*) из таблицы.

В листинге 8.4 предложен пример триггера обеспечивающего каскадное удаление данных из подчиненной таблицы, по факту удаления записи из главной таблицы.

**Листинг 8.4. Пример триггера TSQL**

```
CREATE TRIGER Continent_TR_DELETE
ON Continent
FOR DELETE
AS
DECLARE @Continent _ID INT
Select @ Continent _ID = Continent_ID  From INSERTED
DELETE FROM Country WHERE Continent_ID=@Continent_ID
```

## Задание

На основе БД разработанной на 17 лабораторной работе. Разработайте хранимые процедуры предназначенные для:

- вставки, редактирования и удаления данных из таблиц;
- просмотра данных в отдельных таблицах БД;
- просмотра сводных данных (объединение всех таблиц БД);
- поиска данных;
- фильтрации данных.

# 9. Разработка приложений по технологии клиент-сервер

*Вид занятия – практическое занятие*
*Время занятия – 2 часа*
*Программное обеспечение – Microsoft SQL Server, Delphi, C++ Builder*

Занятие посвящено изучению ряда компонентов Delphi предназначенных для просмотра, вставки, редактирования и удаления данных с применением команд на языке структурированных запросов SQL. Рассматриваемые на занятии компоненты обычно применяются для построения клиентских приложений БД.

## Запрос TADOQuery

Компонент `TADOQuery` предназначен для взаимодействия с базой данных на основе структурированного языка запросов SQL. Для этих целей он инкапсулирует команду на языке SQL и отправляет ее в адрес сервера БД по команде клиентского приложения.

Ключевым свойством компонента является

```
property SQL: TStrings;
```

Это набор строк, в котором программист описывает команды на языке SQL. Для заполнения набора строк можно использовать все функциональные возможности инкапсулированные в классе `TStrings`. А как минимум надо помнить о существовании пары методов: `Clear()` – очистка содержимого набора строк и `Add()` – добавить новую строку в конец набора.

**Листинг 9.1. Пример работы с компонентом `TADOQuery`**

```
with ADOQuery1 do
begin
SQL.Clear;
SQL.ADD('Select * from writer');
Open;
end;
```

Весь текст запроса можно проконтролировать, обратившись к свойству:

```
property Text: String; //только для чтения
```

> Перед началом работы с компонентом `TADOQuery` надо обязательно подключить компонент к БД. Для этого следует воспользоваться свойством `ConnectionString` или `Connection`

## Выполнение запроса SQL

После того, как программист опишет текст SQL он получает право отправить команду на выполнение. Для этого компонент `TADOQuery` вооружен двумя методами, отправляющими команду на выполнение запроса SQL:

```
procedure Open;
procedure ExecSQL;
```

Метод `Open()` унаследован от универсального набора данных `TDataSet`. Он применяется только тогда, когда при помощи компонента `TADOQuery` программист планирует получить некоторый массив строк. Метод `ExecSQL()` просто приказывает БД отработать поступившую команду SQL.

> Если текст запроса SQL начинается с команды SELECT (другими словами мы планируем получить от БД информацию) — используйте метод `Open()`. Если же мы редактируем данные (команды INSERT, UPDATE, DELETE), создаем объекты БД (команды CREATE ….), уничтожаем объекты (DROP …), то в этом случае наш верный союзник метод `ExecSQL()`.

## Параметры запроса

Для нас уже не секрет, что по сравнению с `TADOTable` компонент `TADOQuery` предлагает программисту весьма гибкий способ получения набора данных. Дело за малым – на языке SQL описываем текст запроса и наш новый знакомый позволит объединять данные из нескольких таблиц, отбирать только требуемые поля, наложить ограничения на получаемые данные, сортировать их в указанной последовательности. Взглянем на строки, в которых предложен способ получения перечня стран с населением более 100 000 человек.

**Листинг 9.2. Выборка данных с помощью TADOQuery**

```
ADOQuery1.SQL.Clear;
ADOQuery1.SQL.Add('SELECT * FROM Country WHERE Population>=100000');
```

Беда в одном, при необходимости поменять условия отбора данных приходится очищать предыдущие SQL-инструкции и писать все заново, а это весьма неудобно. Хотя, проявив немного изобретательности, мы вместо статичного запроса сможем создать более эффективную конструкцию (листинг 9.1)

**Листинг 9.3. Динамическое изменение текста запроса**

```
procedure TForm1.Edit1Change(Sender: TObject);
var X : integer;
begin
X:= StrToIntDef(Edit1.Text,0);
  with ADOQuery1 do
  begin
    If Active Then Close;
    SQL.Clear;
    SQL.Add('SELECT * FROM Country');
    SQL.Add('WHERE Population>='+IntToStr(X));
    Open;
  end;
end;
```

Пользователь вводит число жителей в строку ввода `Edit1`. Мы динамически обновляем условие отбора данных в момент события `OnChange()`.

Параметры окажут неоценимую помощь в запросах, в которых ограничения на отбор данных могут меняться, или в командах модифицирующих данные в таблицах. Продолжим эксперименты с таблицей Country, хранящей данные о названии страны (поле Name) имя столицы (поле Capital) и количество жителей в стране (поле Population). Пример универсальной команды SQL, позволяющий использовать ее неоднократно для вставки данных в таблицу предложен ниже:

**Листинг 9.4. Подготовка к TADOQuery вставке строки**

```
with ADOQuery1 do
begin
SQL.Clear;
SQL.Add('INSERT INTO Country (Name, Capital, Population)');
SQL.Add('VALUES (:Name, :Capital, :Population)');
end;
```

Данная процедура вызывается только один раз, например, при старте приложения. Обратите внимание на строку

```
SQL.Add('VALUES (:Name, :Capital, :Population)');
```

Символ двоеточия признак того, что следующее за ним слово является именем параметра. Теперь, для вставки в таблицу новой записи достаточно передать в текст запроса соответствующие значения. Для хранения параметров в запрос инкапсулировано специальное свойство

```
property Parameters: TParameters;
```

Для обращения к параметру надо указать его индекс от 0 до `ParamCount-1`.

О количестве параметров нас информирует свойство:

```
property ParamCount: Word; //только для чтение
```

**Листинг 9.5. Процедура вставки новой строки**

```
Procedure InsertTown(Name, Capital :String; Population :Cardinal);
begin
ADOQuery1.Parameters [0].AsString  := Name;
ADOQuery1.Parameters [1].AsString  := Capital;
ADOQuery1.Parameters[2].AsInteger := Population;
ADOQuery1.ExecSQL;
end;
```

При передаче значений параметров главное не перепутать их индекс. Если же вы не знаете очередности параметров, то вместо порядкового номера параметра целесообразнее применять метод

```
function ParamByName(const Value: String): TParam;
```

Функция возвращает параметр с именем `Value`

```
ADOQuery1.Parameters.ParamByName('Population').AsInteger:= Population;
```

## Хранимая процедура TADOStoredProc

В отличие от своих коллег (таблицы `TADOTable` и запроса `TADOQuery`), компонент `TADOStoredProc` не предназначен для прямого обращения к набору данных (таблице или объединению таблиц). `TADOStoredProc` предназначен для вызова описанной на SQL сервере хранимой процедуры (stored procedure). В виду этой особенности `TADOStoredProc` не найдет применения в приложениях, работающих с локальными данными (например такими как Access), так как драйверы этих баз данных не поддерживают такой вид объекта. Но, в клиентских приложениях, обслуживаемых SQL-сервером, компонент чувствует себя превосходно.

### Соединение с хранимой процедурой

Для подключения `TStoredProc` к размещенной на сервере процедуре программист должен описать два свойства. В первую очередь в с помощью свойства `Connection` следует подсоединиться к компоненту `TADOConnection`. Затем, в свойстве:

```
property StoredProcName: String;
```

указывается имя хранимой процедуры, которую станет вызывать компонент. Кроме того, если хранимая процедура обладает параметрами, то необходимо проверить список параметров в свойстве:

```
property Parameters: TParameters;
```

Как правило, список параметров заполняется автоматически после подключения компонента к расположенной на сервере процедуре.

### Выполнение хранимой процедуры

Для выполнения хранимой процедуры `TADOStoredProc` может воспользоваться двумя методами:

```
procedure Open;
procedure ExecProc;
```

Процедура `Open()` используется только в тех случаях, когда хранимая процедура возвращает набор данных (процедура построена на основе команды SELECT). Во всех остальных случаях нам нужен метод `ExecProc`.

**Листинг 9.6. Обращение к хранимой процедуре**

```
ADOStoredProc1.Parameters.Params[0].AsString := Edit1.Text; //передача значения
ADOStoredProc1.ExecProc;                      //выполнение процедуры
Result:= ADOStoredProc1.Params[1].Value;      //получение результатов
```

При работе с SQL-серверами Sybase и Microsoft SQL пригодится метод:

```
procedure GetResults;
```

Этот метод возвращает выходные параметры хранимой процедуры в клиентское приложение.

## Транзакции и их изоляция

При разработке БД нацеленных на обслуживание более чем одного пользователя одновременно необходимо досконально разбираться в процессах выполнения транзакций и блокировок.

Стоит выделить 4-ре основные принципа транзакций:

1. **Целостность**. Транзакция может быть завершенной или нет. Транзакция не может быть выполнена частично.
2. **Согласованность**. После выполнения (или после отката) транзакции система должна находиться в некотором известном состоянии.
3. **Изолированность**. Транзакции должны быть автономными и не должны воздействовать на другие транзакции, а также зависеть от них.
4. **Устойчивость**. После завершения транзакции ее цель считается достигнутой, и отменить ее уже нельзя.

Существуют ограничения, которые нельзя выполнить с помощью транзакций. Это операции которые нельзя отменить без существенного влияния на компоненты системы. В частности внутри транзакции нельзя использовать команды на создание (CREATE), удаление (DROP) и реконфигурацию (ALTER) объектов БД.

При построении многопользовательских БД разработчик программы должен предусмотреть поведение программного обеспечения в случае одновременного доступа к одним и тем же данным нескольких пользователей. Ведь нельзя исключить ситуацию, когда пара пользователей попытается отредактировать одну и ту же строку в таблице. Что окажется в ней после их работы? Способ защиты информации многопользовательских БД называется **уровнем изоляции транзакций**. В MS SQL существует 4-ре уровня изоляции:

1. Уровень **Read Uncommitted**. Транзакции разрешено читать данные, в настоящий момент обрабатываемые другими транзакциями. Говоря о таком процессе, часто применяют термин **грязное чтение**.
2. Уровень **Read Committed**. Транзакция может читать только завершенные изменения других транзакций.
3. Уровень **Repeatable Read** (повторяющееся чтение). Уровень изоляции гарантирует, что на просматриваемые пользователем данные не повлияют транзакции, выполняемые другими транзакциями. Недостаток уровня в том, что Ваша транзакция захватывает данные в монопольное пользование и не разрешает обращение к ним из других транзакциям, а это полный запрет на параллельную обработку данных.
4. Уровень **Serializable**. Рекомендуемый уровень изоляции. Он поддерживает все ограничения уровня Repeatable Read, но в тоже время предотвращает чтение фиктивных данных.

## Управление транзакциями и компонент TADOConnection

Для запуска новой транзакции предназначен метод

```
function BeginTrans: Integer;
```

Метод возвращает целочисленное значение, соответствующее уровню вложения транзакции. Если запуск новой транзакции осуществился без заминок, то это приведет к установке в `true` свойства

```
property InTransaction: Boolean;
```

Значение `true` свидетельствует о том, что `TADOConnection` инициировал транзакцию и эта транзакция находится в стадии выполнения. С окончанием транзакции свойство вернется в состояние `false`. Сразу после успешного запуска новой транзакции происходит событие

```
property OnBeginTransComplete: TBeginTransCompleteEvent;
TBeginTransCompleteEvent = procedure(Connection: TADOConnection;
                           TransactionLevel: Integer; const Error: Error;
                           var EventStatus: TEventStatus) of object;
```

Параметр `EventStatus` информирует нас, насколько успешно прошел процесс запуска транзакции. Если операция выполнилась корректно, то в параметре окажется значение `esOK`.

Для сохранения результатов выполнения транзакции программист вызывает процедуру

```
procedure CommitTrans;
```

Сразу после сохранения транзакции возникает событие

```
property OnCommitTransComplete: TConnectErrorEvent;
TConnectErrorEvent = procedure(Connection: TADOConnection; Error: Error;
                                var EventStatus: TEventStatus) of object;
```

Если в процесс выполнения транзакции вкралась ошибка, то для ее отмены воспользуйтесь методом

```
procedure RollbackTrans;
```

Откат транзакции сопровождается событием

```
property OnRollbackTransComplete: TConnectErrorEvent;
```

Если разрабатываемая база данных подлежит эксплуатации в многопользовательском режиме, то особое внимание стоит уделить вопросам обеспечения параллельной обработки данных. Накал борьбы клиентских приложений за записи таблиц можно снизить, установив соответствующий уровень изоляции транзакций

```
property IsolationLevel: TIsolationLevel;
```

Уровень разграничения транзакций определяет возможность одновременного совместного доступа пользователей к одним и тем же данным. В таблице 19.1 приведены допустимые значения уровней изоляции транзакций в ADO. Блокировки упорядочены по степени возрастания пессимистических настроений. Другими словами каждая последующая строка все более и более затрудняет параллельную работу нескольких пользователей с одними и теми же данными, но зато снижает вероятность появления конфликтов.

*Таблица 9.1. Возможные уровни изоляции транзакции ADO – TIsolationLevel*

| ***Уровень изоляции*** | ***Описание*** |
|---|---|
| `ilUnspecified` | Сервер может остаться в любом режиме изоляции транзакций, так как приложение не предъявляет к СУБД каких либо особенных требований |
| `ilChaos` | Изменения, сделанные транзакцией с более высоким уровнем изоляции, не могут быть переписаны текущей транзакцией. |
| `ilReadUncommitted` или `ilBrowse` | **Грязное чтение.** Допустимо чтение данных незавершенных транзакций. |
| `ilCursorStability` или `ilReadCommitted` | **Исключение грязного чтения.** Данные доступны только после завершения транзакции. |
| `ilRepeatableRead` | **Повторяющееся чтение.** Изменения, произведенные другими транзакциями к предварительно прочитанным данным не видны, пока не будет осуществлено повторное чтение данных. |
| `ilSerializable` или `ilIsolated` | **Полная изоляция.** Максимальная степень изоляции транзакций. |

## Задание

На основе изученных компонентов разработайте процедуры и функции, позволяющие получать, добавлять, редактировать и удалять данные из таблиц вашей БД. Операции, связанные с модификацией данных должны быть заключены в транзакции.

# 10. Слабо структурированные данные и язык XML

*Вид занятия – практическое занятие*
*Время занятия – 2 часа*
*Программное обеспечение – среда Microsoft Visual Studio, C++ Builder (Delphi)*

Расширяемый язык разметки (eXtensible Markup Language, XML) был разработан специально для создания структурированных документов – он как никто другой позволяет представлять и манипулировать элементами данных. Любой документ XML может рассматриваться как символьная строка, состоящая из символов разметки и данных. Благодаря тому, что двоичные данные в XML отсутствуют, документ на основе расширяемого языка разметки может быть открыт и отредактирован в любом текстовом редакторе, от элементарного блокнота до текстового процессора класса Microsoft Word.

## Построение простейшего документа XML

Правила создания рядового документа XML столь просты, что для написания демонстрационного проекта в формате XML уйдет не более минуты.

**Листинг 10.1. Документ XML**

```
<?xml version="1.0" encoding="WINDOWS-1251"?>
<!--Это комментарий-->
<example>
	<line1>Простой пример работы</line1>
	<line2>с языком XML</line2>
</example>
```

Каждому открывающему тэгу должен сопоставляться закрывающий тэг (за исключением первой строки документа). Признаком того, что тэг является закрывающим служит наклонная черта “/”.

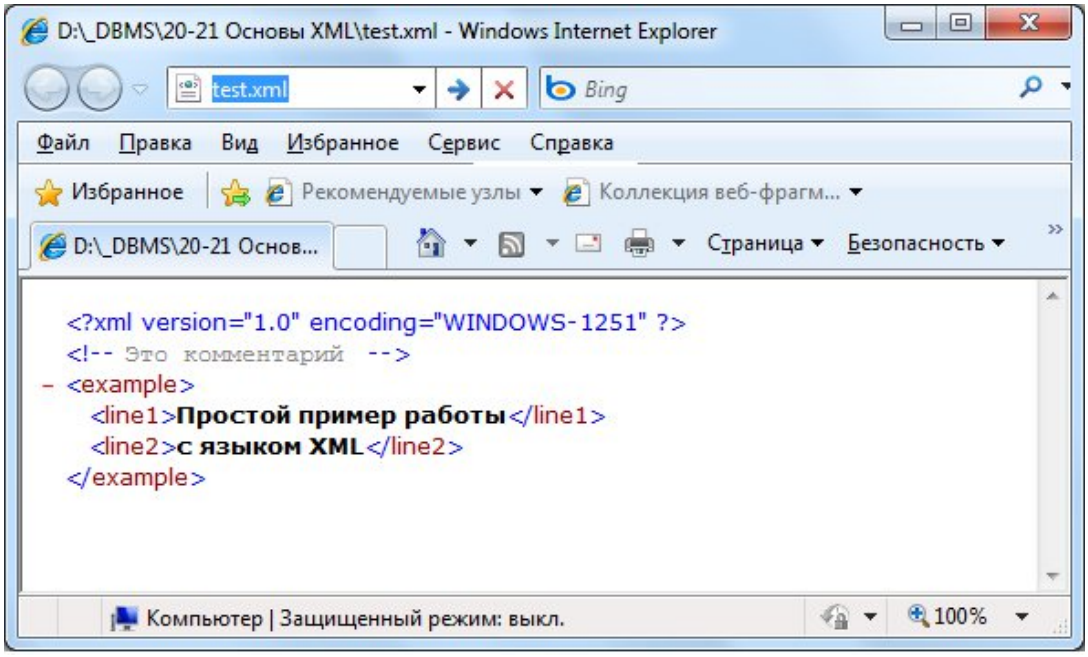



***Рис. 10.1.*** *Внешний вид XML-документа в браузере Windows Internet Explorer*

Совокупность открывающего, закрывающего тэга и данных внутри них

```
<line3>данные</line3>
```

называется *элементом* документа.

Базовые правила определения элементов XML:

1. Каждому открывающему тэгу должен соответствовать закрывающий тэг.
2. Документ XML позволяет использовать вложенные друг в друга тэги, при условии, что тэги не должны перекрываться.
3. В документе XML должен быть всего один корневой элемент.

4. Имена тэгов должны начинаться с буквы. После первого символа разрешается применять цифровые и буквенные символы. Имя не должно содержать символы пробела и двоеточия “:”. В именах запрещено применять аббревиатуры “XML”, “xml”, “Xml”, и т.п.
5. Имена чувствительны к регистру.

## Атрибуты

Предположим, что мы через Интернет получаем прайс-лист книжного магазина. В нем содержатся данные систематизированные по жанру произведения, а внутри элементов жанра <GENRE> .. </GENRE> хранятся элементы с описанием книг <BOOK>..</BOOK>.

**Листинг 10.2. Документ XML с атрибутами**

```
<?xml version="1.0" encoding="WINDOWS-1251"?>
<pricelist>
<GENRE>Роман
   <BOOK WRITER="Толстой Л.Н." PUBLISHER="Мир" PRICE="100.00">
    Война и мир</BOOK>
   <BOOK WRITER="Толстой Л.Н." PUBLISHER="Детская литература"   PRICE="57.00">
    Анна Каренина</BOOK>
   <BOOK WRITER="Достоевский Ф.М." PUBLISHER="Кавказ" PRICE="210.00">
    Братья Карамазовы</BOOK>
  </GENRE>
  <GENRE>Поэма
   <BOOK WRITER="Гете И.В." PUBLISHER="Худ. литература" PRICE="45.5">
    Фауст</BOOK>
  </GENRE>
</pricelist>
```

Самое главное в новом примере то, что открывающий тэг описания книг содержит атрибуты. Атрибут состоит из названия и значения. Порядок именования атрибутов ничем не отличается от правил задания имен для элементов документа. Значения атрибутов заключаются в двойные или одинарные кавычки, в нашем примере в них содержатся дополнительная информация, описывающая ту или иную книгу.

## Определение документа DTD

Нельзя исключить вероятность того, что заложенная разработчиком смысловая нагрузка в элементы документа окажется не столь очевидной как для пользователя, так и для анализатора XML. Примеров неоднозначностей в толковании разметки документа достаточно много. Например, создавая элемент `<PHONENUM>` мы рассчитываем, что в нем будет храниться 11-значный номер телефона, но пользователь предположит, что достаточно ограничиться сокращенным 6-значным номером абонента. Для решения подобных проблем в базах данных разработан весьма эффективный аппарат поддержки целостности и непротиворечивости данных, а как обстоят дела в XML?

В ситуации неоднозначности толкования разметки стоит воспользоваться еще одним козырем XML – способностью подключения к документу дополнительного описания называемого *определением типа документа* (*Document Type Definition, DTD*). DTD конкретизирует, что именно должно содержаться в элементах и атрибутах документа.

Описание структуры документа XML с помощью DTD начинается сразу после обязательного заголовка документа и включает слово DOCTYPE. Признаком завершения секции описания выступает закрывающая квадратная скобка и закрывающий тэг ]>. Таким образом простейшее описание будет выглядеть следующим образом:

```
<!DOCTYPE имя_элемента [ ]>
```

Внутри секции содержится некоторый набор тэгов `<!` и `>` в которые заключаются имена элементов структуры или списки атрибутов. Восклицательный знак указывает на то, что это *элемент объявления*. Сразу за восклицательным знаком (без пробела) следует слово `DOCTYPE` и имя элемента структуры.

В простейшем случае описание DTD заносится непосредственно в документ XML, сразу после заголовка с версией. В представленном ниже примере мы создаем блок определения типа документа, информирующего нас о том, что:

1. Каждый информационный блок документа `pricelist` состоит из двух элементов (`GENRE` и `BOOK`).
2. Оба элемента представляют собой разобранные символьные данные (Parsed Character Data, PCDATA) или, проще говоря, элементы хранят обычный текст.

3. Элемент BOOK содержит список атрибутов (тэг <!ATTLIST>). Из них два атрибута обязательны для заполнения (#REQUIRED) и один необязательный (#IMPLIED).

**Листинг 10.3. Документ XML с описанием DTD**

```
<?xml version="1.0" encoding="WINDOWS-1251" standalone="yes"?>
<!DOCTYPE pricelist [
      <!ELEMENT GENRE (#PCDATA)>
      <!ELEMENT BOOK (#PCDATA)>
        <!ATTLIST BOOK
              WRITER CDATA    #REQUIRED
               PUBLISHER CDATA #IMPLIED
               PRICE CDATA     #REQUIRED>]>
<pricelist>
  <GENRE>Роман
      <BOOK
        WRITER="Толстой Л.Н."
        PUBLISHER="Мир"
        PRICE="100.00">
        Война и мир</BOOK>
  </GENRE>
</pricelist>
```

В процессе чтения блоков данных XML, анализатор проверяет их на соответствие требованиям DTD. Если выявляются отсутствующие (или наоборот лишние) элементы и атрибуты, или элемент данных находится не в ожидаемом месте, или содержимое элементов и атрибутов не соответствует объявлению, то анализатор делает вывод о том, что документ некорректен.

В нашем примере элемент BOOK содержит объявления атрибутов. Список атрибутов начинается с тэга <!ATTLIST, далее следует имя элемента, которому принадлежит список, затем описание атрибутов и закрывающая угловая скобка. Ключевое достоинство атрибута в том, что тип хранимых в нем данных не ограничивается простым CDATA (табл. 10.1).

*Таблица 10.1. Типы атрибутов DTD*

| *Тип* | *Описание* |
|---|---|
| CDATA | В качестве значения атрибута могут использоваться символьные данные |
| ID | Признак, что значение атрибута должно быть уникальным. Весьма полезное качество при работе с базами данных, так как позволяет однозначно идентифицировать элемент, которому принадлежит уникальный атрибут |
| IDREF | В атрибуте хранится ссылка на идентификатор (ID) элемента |
| IDREFS | В атрибуте находится разделенная пробелами последовательность значений IDREF |
| ENTITY | Ссылка на внешний объект (например, файл мультимедиа). При проверке корректности документа подобный объект пропускается анализатором |
| ENTITIES | Разделенные пробелами ссылки на внешние объекты |
| NMTOKEN | Строка символьных данных, которая будет рассматриваться анализатором как корректное значение |
| NMTOKENS | Строка, содержащая несколько разделенных пробелами NMTOKEN |
| Перечислимый атрибут | Атрибут, содержащий список допустимых значений. Например:<br><!ATTLIST video format (MPEG1 \| MPEG2 \| MPEG4) "MPEG2"><br>В скобках мы перечислили все допустимые для атрибута значения видеоформата. По умолчанию предлагается применять формат видеозаписи MPEG2 |

Описание DTD позволяет формировать достаточно сложные модели XML-данных. Допустим нам необходимо построить иерархическую структуру состава системного блока персонального компьютера (листинг).

**Листинг 10.4. Описание DTD для системного блока**

```
<!DOCTYPE COMPUTERS [
<!ELEMENT COMPUTER (CPU, HDD+, VIDEO, RAM, OTHER*)>
<!ELEMENT CPU (#PCDATA)>
<!ELEMENT HDD (#PCDATA)>
```

```
<!ELEMENT VIDEO (#PCDATA)>
<!ELEMENT RAM (#PCDATA)>
<!ELEMENT OTHER (#PCDATA)>
]>
```

указывают на то, что элемент “COMPUTER” должен содержать информацию о процессоре “CPU”, жестком диске “HDD”, видеокарте “VIDEO”, памяти “RAM” и другом оборудовании “OTHER”. С этой целью названия вложенных элементов перечисляются в круглых скобках после слова “COMPUTER”. Подобное перечисление требует, чтобы при вводе XML данных строго соблюдался порядок следования элементов, на первом месте процессор, на втором – винчестер, и т.д. Если обозначенный порядок, по каким-то причинам нарушится, то анализатор документа немедленно известит нас об ошибке в документе.

Обратите внимание еще на одну особенность описания DTD для примера с компьютером. Во второй строке листинга после названий элементов “HDD” и “OTHER” появились ранее не встречавшиеся нам символы “+” и “*”. Благодаря им мы приобретаем дополнительные возможности (табл. 10.2) по введению ограничений на количество элементов данных и установке признака обязательности и необязательности элемента.

*Таблица 10.2. Ограничение количества элементов*

| ***Определитель*** | ***Описание*** |
|---|---|
| ? | Признак необязательного элемента. Он может вообще отсутствовать в XML данных, либо появляться один раз. |
| + | Элемент может появиться один или несколько раз |
| * | Элемент может не появляться или появляться несколько раз |
| *Пустой определитель* | Значение по умолчанию. Элемент обязателен и может появляться только один раз |

С помощью символов “+” и “*” мы сумели создать определение DTD для XML-документа с переменным числом элементов. Теперь, при описании состава компьютера нам позволено подключать один или несколько жестких дисков  и неограниченное число дополнительного оборудования “OTHER”, при этом документ XML остается корректным.

**Листинг 10.5. Пример XML файла с данными системного блока**

```
<?xml version="1.0" encoding="WINDOWS-1251" standalone="no"?>
<!DOCTYPE COMPUTERS SYSTEM "computers.dtd">
<COMPUTERS>
	<COMPUTER>
		<CPU>INTEL CoreDuo2400</CPU>
		<HDD>320GB</HDD>
		<VIDEO>RODEON8800</VIDEO>
		<RAM>2GB</RAM>
	</COMPUTER>
	<COMPUTER>
		<CPU>AMD 3000</CPU>
		<HDD>500GB</HDD>
		   <HDD>500GB</HDD>
		<VIDEO>NVidia</VIDEO>
		<RAM>4GB</RAM>
		<OTHER>ТВ-тюнер Aver</OTHER>
		<OTHER>Принтер HP LJ P1005</OTHER>
	</COMPUTER>
</COMPUTERS>
```

выше предложен фрагмент документа XML ссылающийся на файл с описанием computers.dtd.

## Задание

Разработайте DTD-описание и описание XML Schemas документа для экспорта всех данных (за исключением ключевых полей) из вашей БД. В соответствии с полученным описанием разработайте и заполните XML-документ данными (соответствующим 10-15 строкам таблиц БД).

# 11. Создание БД в InterBase

*Вид занятия – практическое занятие*
*Время занятия – 2 часа*
*Программное обеспечение – СУБД InterBase (FireBird)*

Основным инструментом администратора СУБД InterBase выступает консоль **IBConsole**, доступ к которой вы сможете получить через меню. По умолчанию консоль администрирования IBConsole.exe окажется в каталоге

C:\Program Files (x86)\Embarcadero\Studio*n.n*\InterBaseXE7\bin

Воспользуйтесь консолью и подключитесь к экземпляру сервера. На запрос регистрационных данных введите имя *SYSDBA* с паролем *masterkey*. Для создания новой БД обратитесь к пункту контекстного меню **Create Database** щелкнув правой кнопкой по узлу Database (рис. 11.1).

---

По умолчанию для получения административного доступа к СУБД используется имя “SYSDBA” с паролем “masterkey”. В период проектирования разработчик БД может не менять этот логин и использовать его в своей работе, однако при развертывании СУБД у конечного пользователя следует обязательно сменить общеизвестный пароль!

---

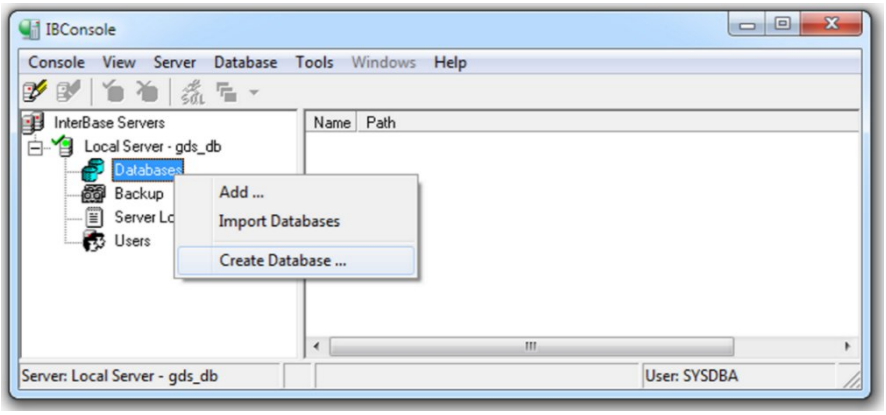



*Рис. 11.1. Создание новой БД средствами IBConsole*

В простейшем случае для создания нового экземпляра БД в достаточно лишь указать имя и путь к файлу с будущей БД (рис. 11.2). Если требуется более тонкая настройка платформы БД, то изучите группу опций **Options**.

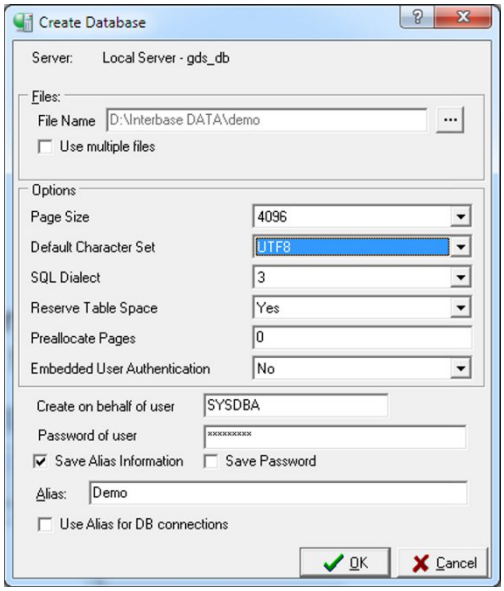



*Рис. 11.2. Настройка параметров создаваемой БД*

## Определение домена в консоли администрирования

Если вы хотите пойти по наиболее простому пути создания домена, то сразу обращайтесь к консоли администрирования IBConsole. Для этого в консоли следует развернуть дерево целевой БД и выбрать в нем узел **Domains**. Щелчок правой кнопкой мышки вызовет контекстное меню, в котором нам останется выбрать пункт **Create** (рис. 11.3).

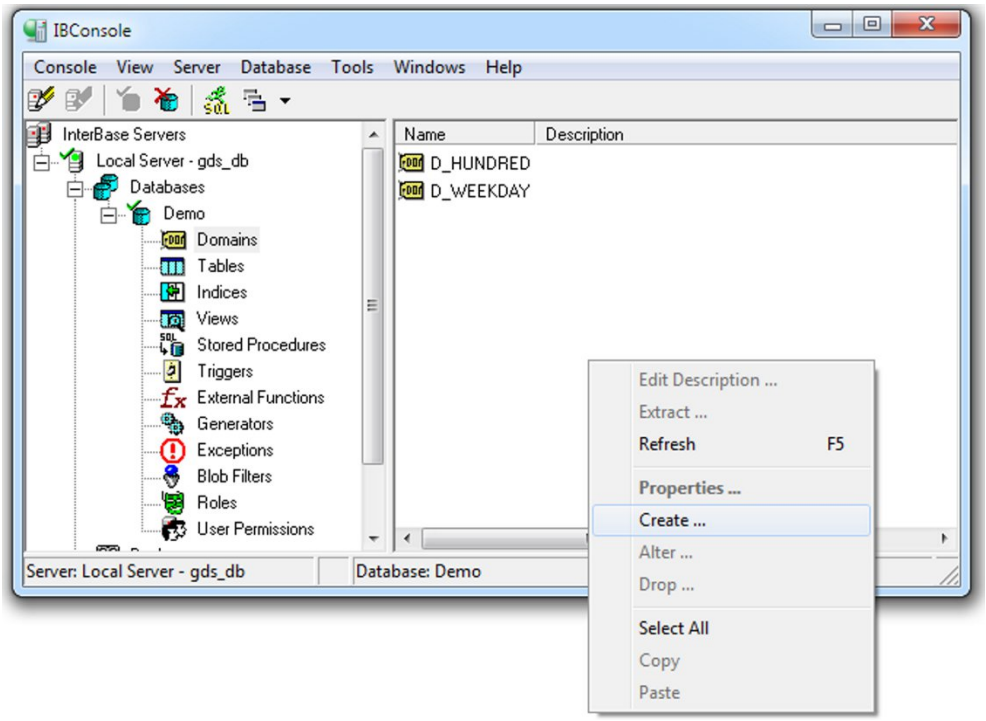



*Рис. 11.3. Создание домена средствами IBConsole*

В ответ на это действие перед программистом раскроется окно редактора домена (рис. 21.4) в котором, кроме собственно уникального имени домена **Domain Name**, следует:

- определить опорный тип данных **Data Type**;
- назначить значение по умолчанию **Default**;
- подтвердить запрет на вставку неопределенного значения **Not Null**;
- описать выражение-ограничение на перечень допустимых значений **Check**.

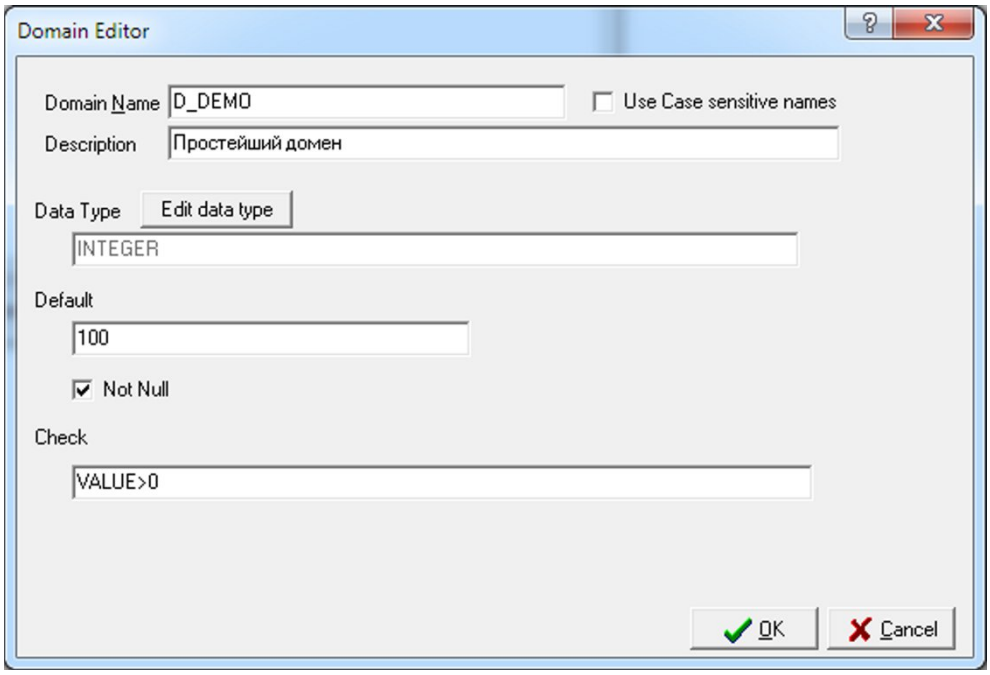



*Рис. 11.4. Создание домена средствами IBConsole*

## Таблицы и консоль администрирования

Сторонникам работы с таблицами с помощью консоли администрирования сервера безусловно понравится удобный и одновременно простой интерфейс утилиты. Для создания новой таблицы достаточно выбрать узел **Tables** в дереве объектов проектируемой БД и затем щелкнуть правой кнопкой мышки в области списка таблиц и вызвать пункт меню **Create** (рис. 11.5).

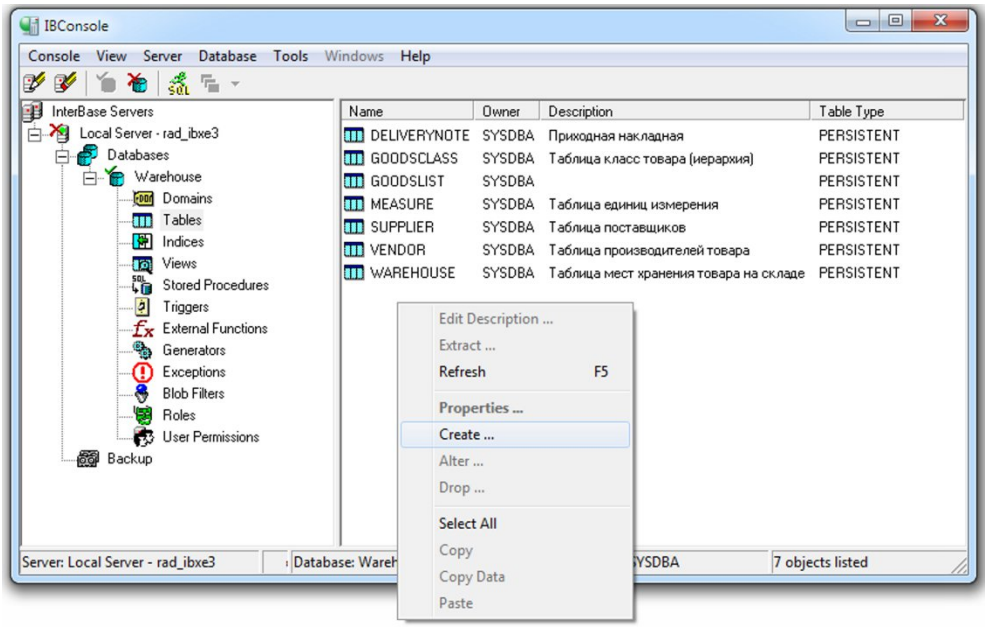



*Рис. 11.5. Команда на создание новой таблицы в IBConsole*

В ответ на это действие перед разработчиком появится новое окно редактора таблицы (**Table Editor**) в котором следует определить (рис. 11.6):

- имя создаваемой таблицы (в строке ввода **Table Name**);
- перечень полей таблицы (кнопка **Add Field**);
- первичный ключ (вкладка **Primary Key**);
- поля с уникальными значениями (вкладка **Unique Keys**);
- внешние ключи (вкладка **Foreign Keys**) для связи с другими таблицами;

- дополнительные ограничения накладываемые на поля таблицы (вкладка **Check Constraints**).

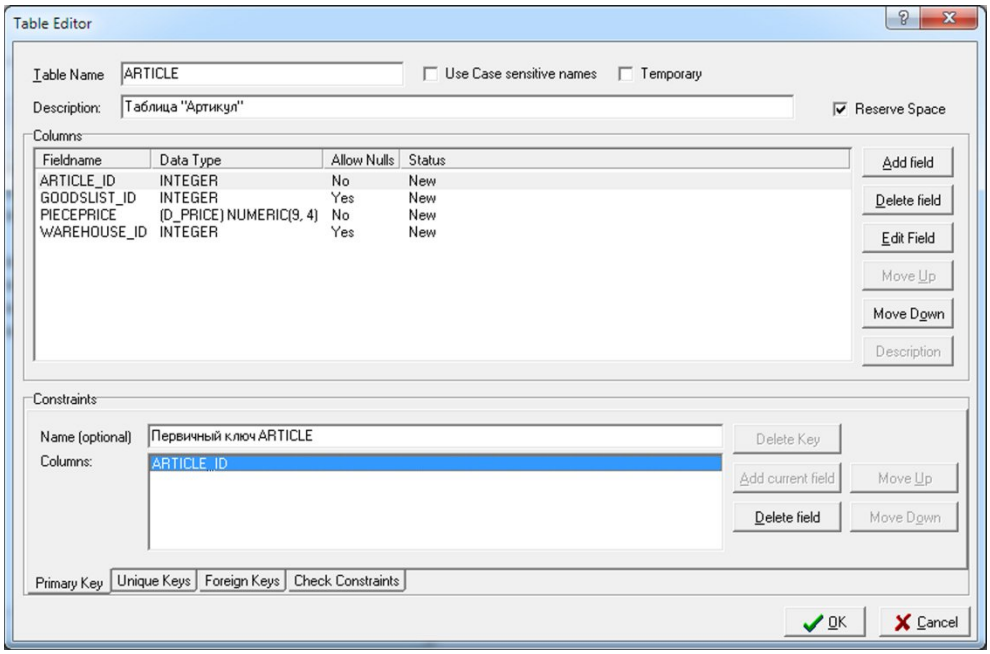



*Рис. 11.6. Создание новой таблицы в консоли IBConsole*

Помимо создания таблиц, консоль администрирования позволяет просматривать свойства, редактировать и удалять таблицы.

## Создание значений ключа с помощью генератора

Для автоматического ввода значений первичных ключей разработчики СУБД InterBase придумали оригинальное решение. Оно заключается в создании отдельного объекта–генератора, представляющего собой счетчик автоинкрементного типа. Для того, чтобы вдохнуть жизнь в генератор следует воспользоваться инструкцией из листинга 11.1.

**Листинг 11.1. Создание и инициализация генератора с помощью SQL**

```
CREATE GENERATOR G_WAREHOUSE_ID;
SET GENERATOR G_WAREHOUSE_ID TO 0;
```

В этом примере в первой строчке в базе данных создается генератор с именем G_WAREHOUSE_ID, а затем этому генератору присваивается стартовое значение 0.

Для обращения к генератору предусмотрена функция:

```
GEN_ID(имя_генератора, шаг_приращения);
```

Названная функция обладает двумя параметрами, в которые передаются имя генератора и шаг приращения значения. Для того, чтобы подготовленное генератором число попало в столбец первичного ключа таблицы функция `GEN_ID` вызывается в триггере (листинг 11.2) или хранимой процедуре.

**Листинг 11.2. Генерация значения ключа в триггере**

```
CREATE TRIGGER T_WAREHOUSE_BI_GEN
FOR WAREHOUSE
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
IF (NEW.WAREHOUSE_ID IS NULL) THEN
    NEW.WAREHOUSE_ID=GEN_ID(G_WAREHOUSE_ID, 1);
END;
```

Несмотря на некоторые сложности работы с генераторами у них есть неоспоримое достоинство – они предоставляют программисту дополнительную степень свободы при управлении значениями первичных ключей.

Если при отладке БД вы осуществляли ввод новых записей в таблицу не применяя генератор (например, отключив триггер и формируя значение первичного ключа вручную), то затем не забудьте перестроить текущее значение генератора, установив в него значение не ниже текущего значения первичного ключа таблицы. Для этого вам потребуется команда SET GENERATOR.

## Представления и консоль администрирования

Для быстрого создания представления достаточно обратиться к консоли администрирования **IBConsole**. Найдя узел **Views** в иерархии объектов базы данных щелкните правой кнопкой мышки в области списка представления и выберите элемент меню **Create**. В ответ на эти действия перед вами появится редактор представления **View Editor** (рис. 11.7).

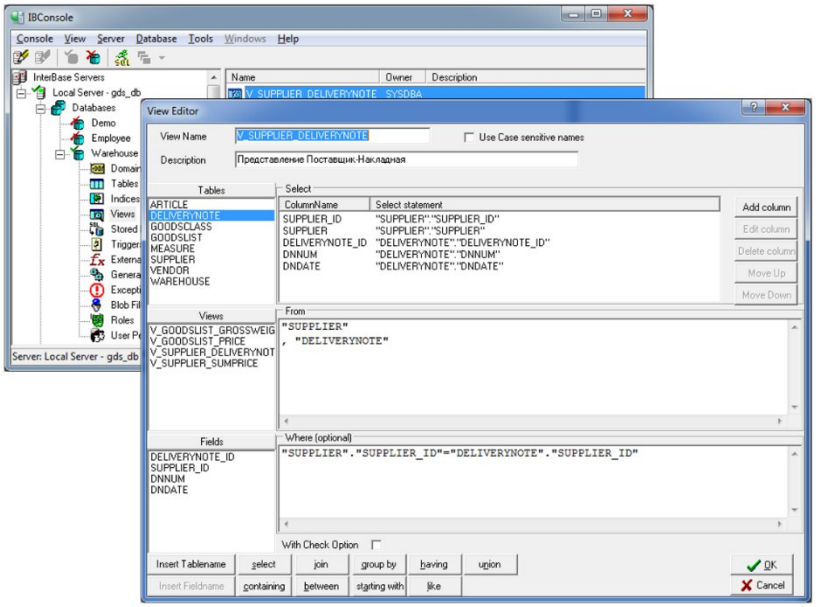



*Рис. 11.7. Создание представления с помощью консоли администрирования IBConsole*

Для описания нового представления следует определиться со следующими атрибутами:

- имя представления (строка ввода **View Name**);
- состав столбцов входящих в представление (кнопка **Add column**). При описании отдельного столбца редактор попросит указать его псевдоним, он станет отображаться в колонке **ColumnName** списка **Select**;
- в многострочное поле ввода **From** следует поместить перечень таблиц и уже существующих представлений на основе которых будет создано новое представление;
- если представление формируется на базе двух или более исходных отношений, то в многострочное поле ввода **Where (optional)** записывается условие объединения.

Закончив описание нажимаем на кнопку OK — InterBase проверит корректность описания представления и создаст новый объект.

## Задание

В соответствии с утвержденной ER–моделью базы данных сделайте следующее:

1. Создайте БД средствами консоли администрирования InterBase (или Firebird).
2. Создайте не менее 3 доменных ограничений, расширяющих возможности различных типов данных.
3. Создайте все таблицы БД с применением доменных ограничений и генераторов для первичных ключей.
4. Убедитесь, что все таблицы обладают механизмом генерации первичного ключа.
5. Создайте не менее 3 представлений, одно из них должно объединить все таблицы БД.

Воспользовались пунктом меню **View Metadata** получите метаданные разработанной БД.

# 12. Процедурный SQL, хранимые процедуры

*Вид занятия – практическое занятие*
*Время занятия – 2 часа*
*Программное обеспечение – среда InterBase (FireBird)*

Хранимая процедура представляет собой подпрограмму на языке SQL, сохраняемую в системном каталоге СУБД. Благодаря тому, что процедура выполняется на сервере, она не требует ресурсов от клиентских станций, что в значительной степени повышает производительность.

В код хранимой процедуры выносятся как наиболее часто вызываемые, так и максимально “тяжеловесные” и трудоемкие команды. В качестве таких распространенных операций стоит упомянуть процедуры вставки, модификации, удаления и выборки данных из таблиц. Благодаря тому, что процедура может содержать блоки команд на языке SQL, переменные, условные операторы и циклы, она позволяет строить весьма сложные логические конструкции, позволяющие решать нетривиальные задачи.

Обобщенная синтаксическая конструкция создания хранимой процедуры выгладит следующим образом

```
CREATE PROCEDURE имя_процедуры [(входной_параметр1 тип, …)]
                               [RETURNS (выходной_параметр1 тип, …)]
AS
[блок объявления переменных]
BEGIN
тело_процедуры
END
```

Процедуры можно создавать как с помощью консоли администрирования (рис 12.1), так и прямым вызовом инструкции `CREATE PROCEDURE`.

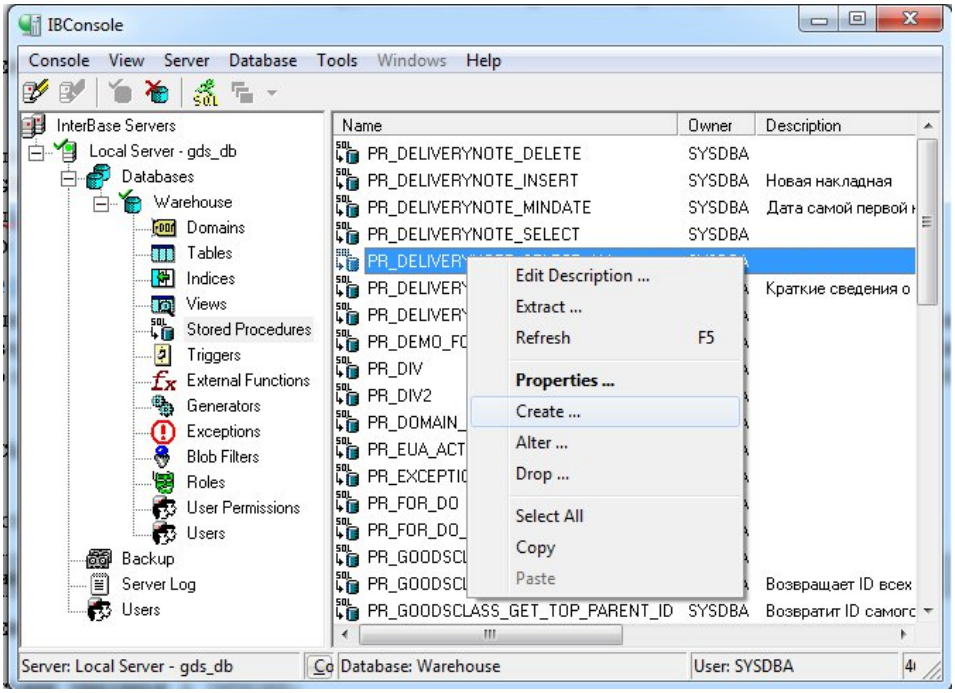



Рис. 12.1. вызов редактора хранимой процедуры из окна IBConsole

## Переменные

Если вы вспомните свои первые шаги при изучении любого высокоуровневого языка программирования, то наверняка отправной точкой стала работа с переменными. Мы не станем изменять этой традиции, тем более, что процедуры и триггеры обладают правом объявлять локальные переменные. Синтаксис объявления таков

```
DECLARE VARIABLE имя_переменной тип_данных;
```

Листинг 12.1 демонстрирует порядок работы с переменными в рамках хранимой процедуры

**Листинг 12.1. Объявление переменных в хранимой процедуре**

```
CREATE PROCEDURE PR_VARDEMO
AS
DECLARE VARIABLE A INTEGER;
DECLARE VARIABLE B INTEGER;
DECLARE VARIABLE C FLOAT;
BEGIN
    A=5;
    B=2;
    C=A/B;
   /*...*/
END
```

В InterBase существует следующее соглашение по работе с переменными. При обращении к переменным внутри процедур и триггеров достаточно просто указать идентификатор переменной, однако если переменная используется в конструкции SQL (например, совместно с оператором SELECT), то имени переменной должен предшествовать *маркер двоеточия* ":".

Листинг 22.1 раскрывает еще одну важную синтаксическую конструкцию — *составной оператор* `BEGIN...END`. Все операторы, входящие в составной оператор рассматриваются компилятором как единый блок кода. Внутри `BEGIN...END` могут находиться не только операторы присваивания, но и условные операторы; структуры циклов; операторы вызова исключительных ситуаций; инструкции SQL; и т п.

## Выборка данных с помощью SELECT ... INTO

Обычно запросы на выборку основанные на инструкции SELECT предназначены для возвращения результирующего набора, включающего несколько строк с данными. Однако это не догма, случаются ситуации, когда создатель запроса планирует получить всего одну строку, например содержащую результат выполнения одной из функций агрегирования. В таких случаях (когда в результате выполнения запроса всегда будет возвращена лишь одна строка с данными) можно воспользоваться услугами оператора INTO, позволяющего явным образом передать значения из столбцов таблицы в заранее подготовленные переменные. Синтаксис подобного запроса выглядит следующим образом

```
SELECT <столбец1> [, < столбец2> ...]
FROM имя_таблицы WHERE <условия отбора>
INTO :переменная1 [, :переменная2 ...];
```

Сохраненные в переменных значения могут быть задействованы в условном операторе, в циклах или подвергнуты какой-то иной обработке. Листинг 12.2 демонстрирует порядок применения инструкции `SELECT…INTO` в хранимой процедуре

**Листинг 12.2. Получение названия класса товара по его первичному ключу**

```
CREATE PROCEDURE PR_GOODSCLASS_SELECTNODE
(AGOODSCLASS_ID INTEGER)
RETURNS
(AGOODSCLASS VARCHAR(255))
AS
begin
  SELECT GOODSCLASS  FROM GOODSCLASS
  WHERE GOODSCLASS_ID=:AGOODSCLASS_ID
  INTO  :AGOODSCLASS;
end
```

## Условный оператор IF...THEN...ELSE

Организуя вычислительный процесс, программист должен обладать возможностью направлять ход вычислений в то или иное русло в зависимости от полученных в ходе выполнения программы промежуточных результатов или введенных пользователем исходных данных. Для этих целей в языке SQL имеется условный оператор IF…THEN.

*Условный оператор*, (или как его иногда называют *оператор ветвления*) задается с помощью инструкции

```
IF (условие) THEN  <оператор> [ELSE <оператор>];
```

Поведение условного оператора определяется описываемым сразу за ключевым словом IF условием. В качестве условия может использоваться любой правильный предикат. Если условие истинно, то

выполнению подлежит оператор (или блок операторов) следующий после THEN, иначе (если условие ложно) — обрабатываются альтернативные операторы, принадлежащие необязательной секции ELSE.
При необходимости условие может быть составным, в этом случае элементы условия объединяются с помощью логических операторов AND, OR и NOT. Предположим, что в после какого-то действия с БД мы обращаемся к паре переменных ASUPPLIER и CITYCODE. Дальнейшие действия зависят от состояния этих переменных (листинг 22.3).

**Листинг 12.3. Пример условного оператора**

```
IF (((CITYCODE >=100) AND (CITYCODE <=200))
     OR
    (ASUPPLIER IS NULL)) THEN
BEGIN
  /*...*/
END
```

Как видите, составной оператор `BEGIN…END` подлежит выполнению только при условии, что значение переменной `CITYCODE` принадлежит диапазону чисел от 100 до 200 или если в переменной `ASUPPLIER` находится неопределенность `NULL`.

> Для проверки значения на неопределенность следует использовать выражение `IS NULL` или `IS NOT NULL`. Все другие варианты неприемлемы!

## Цикл WHILE...DO

Циклы предназначены для многократного выполнения заданной последовательности операторов тех пор, пока соблюдается условие выполнения цикла. В InterBase предусмотрена следующая синтаксическая конструкция цикла

```
WHILE (условие) DO
BEGIN
<операторы цикла>
END
```

Это так называемый цикл с предусловием, выполнение которого начинается с проверки истинности условия. Условие должно быть определено внутри круглых скобок, при необходимости условие может быть сложным и содержать несколько предикатов, объединенных логическими операторами.

Пример цикла `WHILE…DO` предложен в листинге 22.4. В данном случае мы разработали процедуру `PR_LOOPDEMO` осуществляющую последовательное приращение значения локальной переменной.

**Листинг 12.4. Цикл WHILE…DO**

```
CREATE PROCEDURE PR_LOOPDEMO
RETURNS(Y INTEGER)
AS
DECLARE VARIABLE X INTEGER;
BEGIN
 X=0;
 WHILE (X<10) DO
 BEGIN
   X=X+1;
 END
 Y=X;
END
```

Запустив процедуру на выполнение с помощью команды

```
EXECUTE PROCEDURE PR_LOOPDEMO;
```

мы получим результат: Y=10.

## Цикл выборки данных FOR SELECT...DO

Некий симбиоз цикла и операции выборки данных представляет синтаксическая конструкция

```
FOR <предложение_SELECT>
DO <операторы_цикла>
```

Предложенное выражение позволяет построчно перебирать все строки отношения, возвращенного запросом SELECT. В запросе SELECT должно быть задействовано ключевое слово INTO, благодаря ему мы на каждой итерации цикла сможем передавать в переменную результаты запроса, а затем, полученный результат станет обрабатываться в рамках тела цикла.

Простой, но вместе с тем показательный пример цикла выборки предложен в листинге 22.5.

**Листинг 12.5. Хранимая процедура выборки всех строк из таблицы поставщиков**

```
CREATE PROCEDURE PR_SUPPLIER_SELECT_ALL
RETURNS
( ASUPPLIER VARCHAR(100),
  ID INTEGER,
  NUM INTEGER)
AS
BEGIN
  NUM=0;
  FOR
  SELECT SUPPLIER, SUPPLIER_ID FROM SUPPLIER
            ORDER BY SUPPLIER INTO :ASUPPLIER,:ID
  DO
  BEGIN
    NUM=:NUM+1;
    SUSPEND;
  END
END
```

В данном листинге описывается хранимая процедура выборки всех записей из таблицы поставщиков с упорядочиванием их по алфавиту. На каждой итерации цикла выборки мы осуществляем приращение значения выходного параметра NUM на единицу. Что получится в результате? Для ответа на этот вопрос напишем обычный запрос SELECT в котором после ключевого слова FROM следует имя только что разработанной хранимой процедуры выборки (листинг 22.6).

**Листинг 12.6. Обращение к хранимой процедуре из запроса SELECT**

```
SELECT NUM, ID, ASUPPLIER FROM PR_SUPPLIER_SELECT_ALL
```

Результат выполнения запроса представлен на экранном снимке (рис. 2.1) — в нашем распоряжении появился виртуальный столбец с порядковым номером записи.

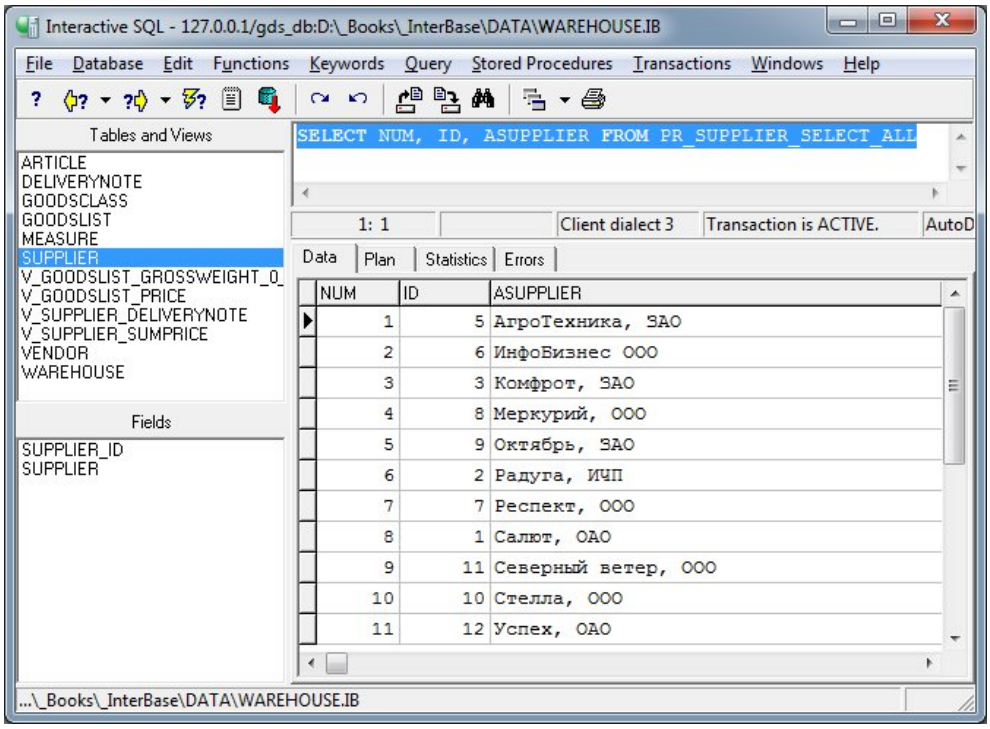



*Рис. 12.2. Запрос к хранимой процедуре выборки FOR SELECT … DO*

## Оператор SUSPEND

Оператор `SUSPEND` предназначен для небольшой приостановки выполнения программного кода и обычно применяется совместно с запросами выборки `SELECT`, в особенности если осуществляется построчная обработка строк отношения (курсоры и `FOR SELECT … DO`).

Приостановка выполнения кода нужна с единственной целью — дать возможность системе заполнить выходной буфер только-что полученными значениями строки. Сразу после заполнения буфера выполнение кода продолжится автоматически.

> Оператор `SUSPEND` следует применять исключительно в хранимых процедурах выборки, в остальных случаях он не нужен, т.к. вызов `SUSPEND` приводит к переходу к завершающему `END` процедуры.

## Оператор EXIT

Как и в большинстве процедурных языков программирования для повышения процедурной гибкости в состав InterBase SQL введен оператор прерывания выполнения программного кода `EXIT`. В случае вызова `EXIT` все следующие за ним строки кода пропускаются и управление передается последнему оператору `END` процедуры.

## Задание

Разработайте следующий перечень хранимых процедур:

1. Процедуры вставки, редактирования и удаления записей в одну из таблиц БД.
2. Процедура выборки всех строк из 2-х объединенных таблиц на основе выражения `FOR SELECT..DO`.
3. Произвольная процедура демонстрирующая работу условного оператора.
4. Произвольная процедура демонстрирующая работу с циклом.

В отчет о выполненной лабораторной работе необходимо включить исходный код разработанных процедур.

# 13. Процедурный SQL, триггеры и события

*Вид занятия – практическое занятие*
*Время занятия – 2 часа*
*Программное обеспечение – среда InterBase (FireBird)*

## Триггеры

Под триггером понимается особая разновидность хранимой процедуры, вызов которой осуществляется автоматически, вне зависимости от пожеланий пользователя БД. Целевое назначение триггеров в первую очередь связано с обеспечением бизнес-правил на уровне таблицы. Код триггера выступает последней линией обороны по поддержке целостности и непротиворечивости данных.

Создание нового триггера в InterBase осуществляется командой `CREATE TRIGGER`, сразу после которой следует уникальное для проектируемой БД имя триггера и название обслуживаемой таблицы:

```
CREATE TRIGGER имя_триггера FOR имя_таблицы
[ACTIVE | INACTIVE]
{BEFORE | AFTER}
{DELETE | INSERT | UPDATE}
[POSITION номер]
AS
[локальные переменные]
BEGIN
 /*код триггера*/
END
```

По умолчанию, сразу после создания триггер приступает к выполнению своих обязанностей. Такое поведение соответствует установке в инструкцию SQL необязательного ключа `ACTIVE`. Если же по каким-либо причинам вам понадобится временно “спящий” триггер не реагирующий на изменения записей в таблице, то воспользуйтесь ключом `INACTIVE`. Подобный триггер не сложно оживить позднее с помощью инструкции `ALTER TRIGGER`.

Триггеры InterBase способны отвечать на 3 пары событий, связанных с модификацией данных в обслуживаемой таблице:

- события `BEFORE INSERT` и `AFTER INSERT` возникают соответственно перед тем как новая строка попадет в таблицу и после того как эта строка будет сохранена;
- события `BEFORE UPDATE` и `AFTER UPDATE` генерируется перед началом и после завершения редактирования данных;
- событие `BEFORE DELETE` предшествует, а `AFTER DELETE` и завершает операцию удаления данных из таблицы.

С одной и той же таблицей может быть сопоставлено практически неограниченное число триггеров (от 0 до 32 767), как вы понимаете все они не могут выполниться одновременно — они могут быть вызваны в неопределенном порядке. Если за определенное событие отвечает единственный триггер, то о последовательности его запуска можно не думать, однако если таких триггеров несколько, то стоит определить очередность их срабатывания. Для этой цели в состав инструкции `CREATE TRIGGER` обязательно включите ключ `POSITION` и укажите порядковый номер триггера в очереди.

### Контекстные переменные

Внутри тела триггера мы имеем право обращаться к двум разновидностям **контекстных переменных**: `NEW` и `OLD`. Для доступа к ним потребуется небольшое уточнение — имя столбца таблицы, с которым ассоциируется контекстная переменная, например, `NEW.SUPPLIER` или `OLD.VENDOR`.

Контекстная переменная `NEW` применяется в триггерах, вызываемых в момент вставки новой строки или модификации данных в таблице. В этой переменной окажется новое значение, претендующее на попадание в таблицу, но пока не сохраненное в ней.

Контролируя передаваемые в таблицу значения, программист сможет построить хорошо эшелонированную линию обороны. Например, в столбцах WEIGHT и GROSSWEIGHT таблицы GOODSLIST должны содержатся величины веса нетто и брутто хранящегося на складе товара. Здравая логика подсказывает, что величина чистого веса нетто не может превышать значение веса товара в упаковке. Исходя из этого утверждения разработаем триггер (листинг 23.1) стоящий на страже только что рассмотренного бизнес-правила.

**Листинг 13.1. Контроль правильности значений нетто и брутто**

```
CREATE TRIGGER T_GOODSLIST_BI2 FOR GOODSLIST
ACTIVE BEFORE INSERT POSITION 1
AS
BEGIN
     IF ((NEW.WEIGHT IS NOT NULL)
     AND (NEW.GROSSWEIGHT IS NOT NULL)
     AND (NEW.WEIGHT>NEW.GROSSWEIGHT)) THEN
            /* Вес нетто не может превышать вес брутто! */
            EXCEPTION E_WEIGHT_EXCEPTION;
END
```

Столкнувшись с нарушением условия нетто ≤ брутто триггер вызывает подготовленное заранее исключение E_WEIGHT_EXCEPTION, которое в свою очередь прекратит операцию вставки данных и известит об этом пользователя.

> В триггере разрешено применять почти все процедурные расширения языка SQL, в том числе условный оператор IF…THEN…ELSE и оператор цикла WHILE…DO.

В триггерах BEFORE значение контекстной переменной NEW может быть изменено (в триггерах AFTER это занятие не имеет смысла). При желании триггер контроля значений нетто и брутто можно переделать и вместо вызова исключения просто поправить содержание одного из столбцов, например, приравняв нетто и брутто

```
NEW.WEIGHT=NEW.GROSSWEIGHT
```

Обращение к переменной OLD имеет смысл в триггерах модификации и удаления данных — в ней хранится старое значение. Предложенный в листинге 23.2 код предназначен для осуществления каскадного удаления записей из таблицы классификатора товаров GOODSCLASS. Напомню, что таблица GOODSCLASS построена по принципу иерархической структуры в которой дочерние записи могут подчиняться родительским за счет связки ключевых полей GOODSCLASS_ID — PARENT_ID.

**Листинг 13.2. Триггер каскадного удаления записей из таблицы GOODSCLASS**

```
CREATE TRIGGER T_GOODSCLASS_BD FOR GOODSCLASS
ACTIVE BEFORE DELETE POSITION 0
AS
BEGIN
  IF (EXISTS (SELECT * FROM GOODSCLASS WHERE
              PARENT_ID=OLD.GOODSCLASS_ID)) THEN
     DELETE FROM GOODSCLASS WHERE PARENT_ID=OLD.GOODSCLASS_ID;
END
```

В коде триггера, вызываемого перед удалением строки из таблицы, осуществляется проверка наличия у подлежащей удалению строки идентифицируемой значением первичного ключа OLD.GOODSCLASS_ID подчиненных записей. Если такие обнаруживаются, то к ним применяется инструкция DELETE. Таким образом дочерние строки исключаются из таблицы до того, как это произойдет с родительской строкой.

## Ввод значений по умолчанию

Еще один аспект применения триггеров связан с обеспечением ввода значений по умолчанию в столбцы, не допускающие неопределенность NULL. Напомню, что значение по умолчанию определяется в момент создания таблицы (или доменного ограничения) с помощью ключевого слова DEFAULT. СУБД гарантирует попадание заданного значения в столбец только при выполнении инструкции вставки INSERT. Если же речь идет о модификации данных, то ничто не запрещает пользователю передать в соответствующий столбец неопределенность NULL. Исправить ситуацию поможет триггер предложенный в листинге 23.3.

**Листинг 13.3. Поддержка значения по умолчанию для столбца**

```
CREATE TRIGGER T_DELIVERYNOTE_BU FOR DELIVERYNOTE
```

```
ACTIVE BEFORE UPDATE POSITION 0
AS
BEGIN
  IF (NEW.DNDATE IS NULL) THEN NEW.DNDATE=CURRENT_DATE;
END
```

Как видите при работе со значениями по умолчанию триггер гораздо более эффективен, чем все остальные инструменты, имеющиеся в распоряжении разработчика БД.

> Если к таблице подключена цепочка триггеров BEFORE, то физически новое значение NEW попадет в столбец только после выполнения всей последовательности триггеров.

## Поддержка корпоративной целостности данных

Зачастую в таблицах могут находится данные, модификация которых недопустима или должна осуществляться в соответствии с особыми правилами. Эти правила определяются некоторыми внутренними, используемыми на данном предприятии установками. В свою очередь разработчики БД транслируют эти установки в правила поддержки корпоративной целостности данных. Например, в нашей БД при поступлении товаров на склад, товар автоматически закрепляется за складом по умолчанию, и только затем, по мере необходимости сотрудники склада осуществляют передачу товара на другие склады и торговые учреждения. Вполне естественно, что параметры склада по умолчанию (его название, атрибуты и самое главное — значение первичного ключа) должны быть постоянны и не допускать несанкционированных исправлений, иначе система может рухнуть.

В демонстрационной БД описание складов хранится в таблице WAREHOUSE. Понятно, что строка с информацией о складе по умолчанию заносится в эту таблицу еще на этапе введения БД в эксплуатацию и получает первичный ключ WAREHOUSE_ID=1. Как мы станем защищать эту запись от “нападок” не очень хорошо подготовленных пользователей которые зачастую ведут себя еще хуже, чем самые отъявленные вредители? Естественно с помощью триггера, так в листинге 23.4 предложен способ защиты записи от попыток удаления.

**Листинг 13.4. Запрет на удаление определенной записи**

```
CREATE TRIGGER T_WAREHOUSE_BD FOR WAREHOUSE
ACTIVE BEFORE DELETE POSITION 0
AS
BEGIN /*нельзя удалить склад по умолчанию*/
  IF (OLD.WAREHOUSE_ID=1) THEN EXCEPTION E_DEFAULTWAREHOUSE_EXCEPTION;
END
```

В рассматриваемом примере при попытке удалить запись с первичным ключом равным 1 немедленно генерируется исключение, и операция прерывается. Примерно такой же код необходимо разместить в триггере `BEFORE UPDATE` и тем самым запретить внесение изменений в эту очень важную запись.

## События

В хранимых процедурах и триггерах может использоваться особый механизм, благодаря которому InterBase получает возможность извещать свои клиентские приложения о том или ином событии, произошедшем внутри БД. В простейшем случае в качестве примеров подобных событий может стать какая-либо важная операция модификации строк в таблице. В качестве более глобальных поводов о необходимости немедленного информирования клиентов можно привести событие поступления в торговую сеть новой партии товаров; сообщения о том, что количество запасов какого-то материала достигло критического уровня; сообщение о том, что сервер будет перезагружен и т.д.

Для того, чтобы механизм событий заработал в полную силу необходимо описать порядок взаимодействия между вызывающим событие сервером и ожидающим событие клиентом. С сервером все предельно просто — ему достаточно известить клиентов о каком-то свершившемся факте с помощью оператора

```
POST_EVENT <имя_события>
```

Имя события чувствительно к регистру и ограничивается 64 символами.

Встретив в SQL-коде оператор `POST_EVENT` сервер передает бразды управления менеджеру событий, который (в случае корректного завершения транзакции) самостоятельно отправит известие о событии клиентам.

Так как в большинстве случаев пользовательское приложение в первую очередь интересуется процедурами вставки, модификации и удаления данных, то для генерации событий лучше всего подходят триггеры `AFTER`. Например, триггер из листинга 23.5 каждый раз после успешного добавления очередной записи в

таблицу поставщиков товаров SUPPLIER будет радовать клиентские приложения событием “SUPPLIER_INSERT”.

**Листинг 13.5. Триггер вызывающий событие “вставка новой записи”**

```
CREATE TRIGGER T_SUPPLIER_AI_EVENT FOR SUPPLIER
AFTER INSERT AS
BEGIN
 POST_EVENT 'SUPPLIER_INSERT';
END;
```

Заметим, что триггер просто осуществит широковещательную отправку сообщения всем прослушивающим сервер клиентам. Само по себе рассылаемое сообщение никакой дополнительной информационной нагрузки не несет, посему дальнейшее развитие ситуации целиком и полностью возлагается на обрабатывающее событие клиентское приложение.

> Событие генерируется только после успешного завершения транзакции, если же осуществляется откат транзакции вызов оператора POST_EVENT пропускается.

Если клиент пишется на языке Delphi (или C++ Builder), то для обеспечения реакции клиента на событие достаточно воспользоваться услугами штатного компонента TIBEvents.

## Задание

1. Предложите примеры 3-4 триггеров, обеспечивающих поддержку корпоративной целостности данных.
2. Для произвольной таблицы создайте три события, уведомляющие о:
   a. вставке новой записи;
   b. редактировании записи;
   c. удалении записи.

# 14. Функции UDF

*Вид занятия – практическое занятие*
*Время занятия – 2 часа*
*Программное обеспечение – среда InterBase (FireBird), C++ (Delphi)*

Сервер InterBase не может похвастаться большим набором встроенных функций, охватывающих весь спектр вероятных задач способных возникнуть перед разработчиком БД. Скажем больше — встроенных функций меньше десятка (приложение 1). Однако этот факт ни в коем случае не говорит о слабости InterBase. Почему? Потому, что создатели InterBase сделали так, чтобы сервер стал способен подключать к БД внешние, или как их еще называют — определенные пользователем функции (user–defined function, UDF).

Внешние функции могут быть разработаны на таких языках программирования, как Delphi, C++ Builder компании Embarcadero и Visual C++ корпорации Microsoft. Список языков программирования можно и расширить. По большому счету единственным условием попадания того или иного кандидата в перечень языков способных написать UDF выступает единственный критерий — умение компоновать и компилировать динамически подключаемые библиотеки (Dynamic Link Library, DLL). Именно DLL и выступает контейнером для пользовательских функций в случае если СУБД развернута на платформе Windows.

## Размещение UDF-библиотеки

Для того, чтобы сервер InterBase смог воспользоваться UDF написанными сторонними разработчиками динамические библиотеки должны быть скопированы в определенные папки. В простейшем случае, вне зависимости от операционной системы вы можете перенести файлы с библиотеками в папку:

*<папка_InterBase>\UDF*

или в папку:

*<папка_InterBase>\intl*

## Подключение внешней функции к БД

Размещение файла библиотеки в целевой папке необходимое, но недостаточное условие для работы с UDF. Для того чтобы наша база данных смогла воспользоваться услугами UDF–библиотеки стоит разобраться с процессом подключения внешних функций к нашему проекту. С этой целью рассмотрим синтаксическую конструкцию `DECLARE EXTERNAL FUNCTION`, отвечающую за импорт внешней функции в БД.

```
DECLARE EXTERNAL FUNCTION имя_функции
[тип_данных ;| CSTRING (int) | DESCRIPTOR
[,тип_данных | CSTRING (int) ...] DESCRIPTOR]
RETURNS { тип_данных [BY VALUE] | CSTRING (int) | PARAMETER n}[FREE_IT]
ENTRY_POINT 'имя_функции_в_модуле'
MODULE_NAME 'имя_модуля';
```

В листинге 24.1 продемонстрирован порядок подключения к БД функции LTRIM() из поставляемого в составе InterBase модуля внешних функций ib_udf.

**Листинг 14.1. Подключение внешней функции IB_UDF_ltrim()**

```
DECLARE EXTERNAL FUNCTION LTRIM
CSTRING(80)
RETURNS CSTRING(80) FREE_IT
ENTRY_POINT 'IB_UDF_ltrim' MODULE_NAME 'ib_udf';
```

Задача только что подключенной функции — подавление всех пробелов, находящихся перед текстом. Это крайне полезная услуга позволит нам бороться с ненужными пробелами, которые могут быть случайно вставлены пользователем в столбец той или иной таблицы.

## Подключение UDF в консоли администрирования

Консоль управления сервером InterBase позволяет осуществить процесс подключения к БД пользовательской функции не только за счет непосредственного использования SQL, но и с помощью

удобного визуального интерфейса. Для этого необходимо открыть БД и обратиться к узлу **External Functions** утилиты **IBConsole**.

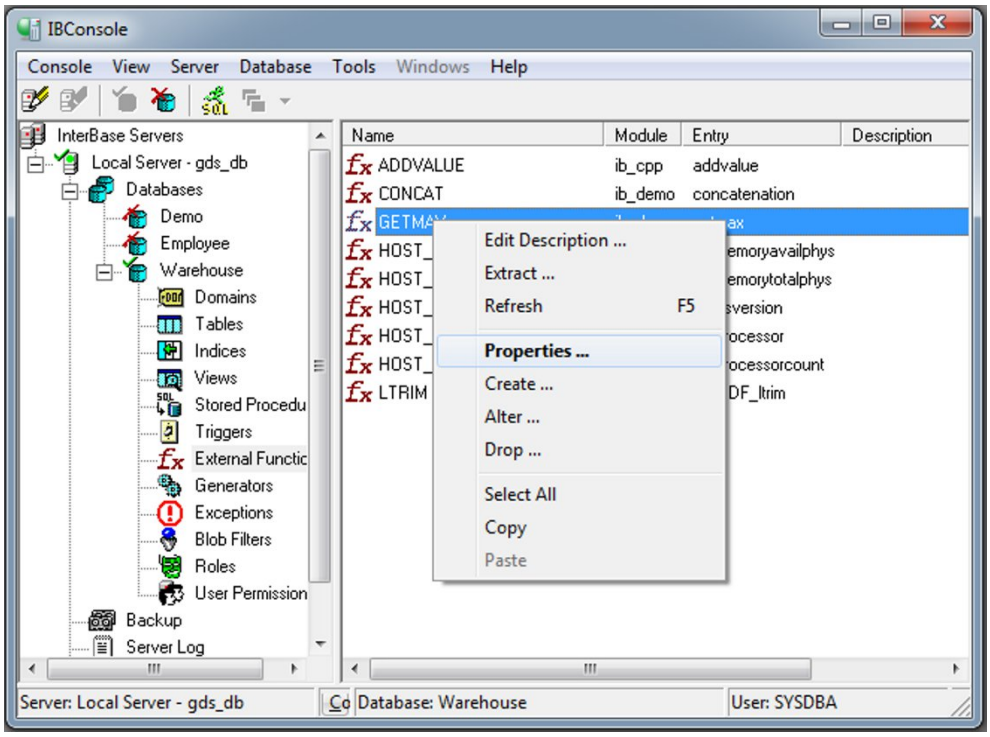



*Рис. 14.1. Управление UDF с помощью консоли администрирования IBConsole*

Выбор узла внешних функций заставит утилиту построить список всех задействованных в текущей БД пользовательских функций. Управление этими функциями, в том числе подключение, модификация и отключение осуществляется при посредничестве контекстного меню (рис. 14.1).

## вызов UDF

Как проверить работоспособность подключенной UDF–функции? Для тестирования можно воспользоваться любой из инструкций DML (`SELECT`, `INSERT`, `UPDATE` и `DELETE`). Простейший способ проверки — вызов функции в SQL-запросе основанном на ключевом слове `SELECT` (листинг 14.2).

**Листинг 14.2. вызов внешней функции LTRIM()**

```
SELECT LTRIM('     ABC') FROM rdb$database;
```

Запустив запрос на выполнение вместо “     ABC” в ответ вы получите строку с удаленными пробелами перед основным текстом — “ABC”.

Объявленные в БД пользовательские функции могут быть вызваны как из запросов динамического SQL, так и из хранимых процедур и триггеров.

## Разработка UDF–библиотек в Delphi

Для разработки библиотек содержащих подключаемые пользовательские функций можно воспользоваться услугами языка программирования Delphi. Для этого в главном меню среды разработки выбираем пункт **File | New | Other** и в окне **New Items** находим элемент **Dynamic-link Library**. В ответ на наши действия Delphi создаст шаблон кода, начинающийся с ключевого слова library — это и есть визитная карточка проекта библиотеки DLL. Сразу сохраните проект. При выборе имени для проекта помните, что после того как вы нажмете кнопку **F9** это имя определит название для библиотеки. В моем первом примере DLL станет называться ib_demo.

> Для 64-битной версии InterBase следует разрабатывать исключительно 64-битные библиотеки!

Напишем функцию getmax(), сравнивающую два целочисленных значения и возвращающих наибольшее из них (листинг 14.3).

**Листинг 14.3. Библиотека на языке Delphi с функцией getmax()**

```
library ib_demo;
```

```
uses
  System.SysUtils,
  System.Classes;

{$R *.res}

function getmax(var x:integer; var y:integer):integer; cdecl; export;
begin
  if x>y then result:=x
  else result:=y;
end;

exports getmax; //секция экспорта

begin
end.
```

Для того чтобы сервер InterBase (как, впрочем, и любое стороннее приложение) смогло воспользоваться описанными в библиотеке функциями, эти функции должны быть помечены, как экспортируемые. Синтаксис объявления экспортируемых библиотекой функций имеет две особенности:

- между экспортирующей функцию библиотекой и выступающим в роли импортера приложением должно быть заключено *соглашение о вызовах*, оно необходимо для обеспечения корректной передачи параметров и возврата результата выполнения функции. Определяя соглашение о вызовах для функции, предназначенной для вызова из InterBase задействуйте стандартное соглашение `stdcall` или соглашение в стиле языка Си `cdecl`;
- объявление экспортируемой функции со спецификатором export и использование секции exports для указания имен экспортируемых процедур и функций.

Откомпилировав библиотеку из листинга 14.3 перенесите dll в каталог *<папка_InterBase>\UDF*

и перейдите в консоль управления InterBase. Теперь нам предстоит подключить UDF к базе данных (листинг 14.4).

**Листинг 14.4. Подключение функции GETMAX() к БД**

```
DECLARE EXTERNAL FUNCTION GETMAX
INTEGER, INTEGER
RETURNS INTEGER BY VALUE
ENTRY_POINT 'getmax' MODULE_NAME 'ib_demo';
```

## Работа со строками

При необходимости работы в UDF с текстовыми параметрами учитывайте тот факт, что фирменные строки Delphi (тип данных `String`) несовместимы с текстовыми типами данных в InterBase. Посему, при определении типов данных для передачи аргументов и возврата результата выполнения функции задействуем исключительно указатели `pAnsiChar` (или `pChar`). Представленный в листинге 14.5 пример информирует о типе процессора, используемого в компьютере на котором развернута СУБД.

**Листинг 14.5. Возврат UDF на языке Delphi текстового значения**

```
function getprocessor:pAnsiChar; CDECL; export;
begin
  case TOSVersion.Architecture of
   arIntelX86 : Result:='Intel x86';
   arIntelX64 : Result:='Intel x64';
   arARM32    : Result:='Архитектура Acorn RISC Machine';
   else         Result:='Unknown';
  end;
end;
```

При подключении UDF возвращающей текстовые данные воспользуемся типом данных CSTRING(n) с указанием максимального числа символов (листинг 14.6).

**Листинг 14.6. Подключение функции getprocessor() к БД**

```
DECLARE EXTERNAL FUNCTION HOST_PROCESSOR
```

```
RETURNS CSTRING(10) CHARACTER SET NONE
ENTRY_POINT 'getprocessor' MODULE_NAME 'ib_host';
```

Рассмотрим еще один небольшой пример, на этот раз демонстрирующий порядок передачи двух текстовых аргументов на вход функции (листинг 14.7).

**Листинг 14.7. Передача текстовых аргументов в функцию Delphi**

```
function concatenation(A:pAnsiChar; B:pAnsiChar):pAnsiChar; cdecl; export;
var C:Ansistring;
begin
  C:=StrPas(A)+StrPas(B);
  result:=pAnsiChar(C);
end;
```

Мы написали функцию, позволяющую осуществлять конкатенацию двух строк, для ее подключения к БД воспользуйтесь листингом 14.8.

**Листинг 14.8. Подключение функции concatenation() к БД**

```
DECLARE EXTERNAL FUNCTION CONCAT
CSTRING(127),CSTRING(127)
RETURNS CSTRING(255)
ENTRY_POINT 'concatenation' MODULE_NAME 'ib_demo';
```

## Особенности разработки в C++ Builder

Пользовательские функции для InterBase в равной степени удобно проектировать как на Delphi, так и на C++ Builder. Так что если вы предпочитаете язык C++, то все в ваших руках. Листинг 14.9 содержит код элементарной пользовательской функции `addvalue()`, предназначенной для суммирования двух вещественных значений.

**Листинг 14.9. Функция суммирования addvalue() на языке C++**

```
#pragma hdrstop
#pragma argsused

extern "C" int _libmain(unsigned long reason)
{      return 1;}

extern "C" double __declspec(dllexport) addvalue(double *x, double *y);
//---------------------------------------------------------------------------
double addvalue(double *x, double *y)
{
 return *x+*y;
}
```

При работе с UDF в C++ (впрочем, как и в случае с Delphi) на вход функции аргументы должны передаваться исключительно по ссылке — это требование InterBase. Поэтому определяя параметры функции из листинга 14.9 мы воспользовались оператором `*`. Для возврата суммы мы напротив, воспользовались способом передачи данных по значению.

Листинг 14.10 содержит код подключения функции суммирования к БД.

**Листинг 14.10. Подключение функции addvalue() к БД**

```
DECLARE EXTERNAL FUNCTION ADDVALUE
DOUBLE PRECISION, DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'addvalue' MODULE_NAME 'ib_cpp';
```

Еще один пример написанной на C++ пользовательской функции демонстрирует порядок возврата текстовых данных (листинг 14.11).

**Листинг 14.11. Функция преобразования номера в название месяца на языке C++**

```
extern "C" char* __declspec(dllexport) monthname(int *monthnum);
//---------------------------------------------------------------------------
char* monthname(int *monthnum)
{
 static char* month[12]=
            {"Январь","Февраль","Март","Апрель","Май","Июнь","Июль",
             "Август", "Сентябрь", "Октябрь", "Ноябрь", "Декабрь"};
 if (*monthnum>=0 && *monthnum<12){
     return month[*monthnum];
    } else return "";
}
```

На этот раз мы разработали функцию которая получив на вход порядковый номер месяца года возвратит нам его текстовое название. Обратите внимание, что на этот раз возврат данных мы производим по ссылке, задействовав тип данных char*.

Импортируя в InterBase написанную на C++ функцию возвращающую текстовую строку хорошим тоном является использование оператора `FREE_IT` уведомляющую вызвавший динамическую библиотеку процесс о том, что теперь можно очистить выделенную для строки память (листинг 14.12)

**Листинг 14.12. Подключение функции monthname() к БД**

```
DECLARE EXTERNAL FUNCTION MONTHNAME
INTEGER
RETURNS CSTRING(10) CHARACTER SET NONE FREE_IT
ENTRY_POINT 'monthname' MODULE_NAME 'ib_cpp';
```

Завершающий штрих — проверка работоспособности функции. Для этого вызовем из интерактивного SQL последнюю в этой главе строку кода (листинг 14.13).

**Листинг 14.13. вызов внешней функции MONTHNAME()**

```
SELECT MONTHNAME(4) FROM rdb$database;
```

В ответ мы получим название месяца с индексом 4.

## Задание

1. Предложите пример работы с любой из встроенных функций (приложение 1).
2. Разработайте (на Delphi или C++ Builder) и подключите к БД UDF–библиотеку, содержащую три функции из представленного ниже перечня:
    1. произведение двух целых чисел;
    2. произведение двух вещественных чисел;
    3. возврат целочисленного значения текущего года;
    4. генерация случайного вещественного числа в диапазоне от 0 до 1;
    5. генерация случайного целого числа в диапазоне от 0 до 100;
    6. расчет выражения $(x+y)^2$;
    7. получение значения sin(x) где x — значение угла в градусах;
    8. расчет выражения $b^2-4ac$;
    9. поиск наибольшего значения из 3 целых чисел;
    10. поиск наименьшего значения из 4 вещественных чисел.

# 15. Подключение к БД из клиентского приложения

*Вид занятия – практическое занятие*
*Время занятия – 2 часа*
*Программное обеспечение – среда InterBase (FireBird), C++ (Delphi)*

Современные версии Embarcadero RAD Studio предоставляют разработчику программного обеспечения по крайней мере 4 альтернативных способа по организации доступа клиентского приложения к БД InterBase:

1. классический подход, основанный на линейке компонентов InterBase (IBX), именно ему мы и уделим максимум своего внимания;
2. доступ к данным на основе компонентов dbExpress (DBX);
3. технология FireDAC;
4. кроме того, в составе InterBase предусмотрены библиотеки для поклонников платформы .NET.

Структура опирающегося на компоненты IBX клиентского приложения представлена на рисунке 15.1. Весь механизм взаимодействия между СУБД и клиентом содержится в специальном клиентском программном обеспечении, ядро которого составляет библиотека gds32.dll.

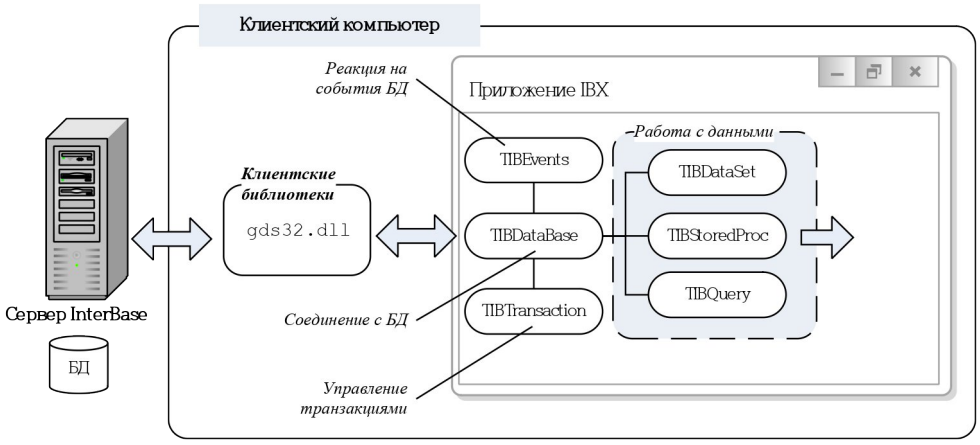



*Рис. 15.1. Архитектура клиентского приложения на основе компонентов IBX*

Первая версия СУБД InterBase была создана талантливым программистом Джимом Старки (Jim Starkey) для компании Groton Database Systems. Именно поэтому в базовой библиотеке gds32.dll применяется аббревиатура gds, кроме того gds зачастую используется в качестве расширения для файла с данными.

Для работы клиентского приложения InterBase необходимо наличие на компьютере библиотеки gds32.dll. При разработке программного обеспечения следует учитывать, что в составе клона InterBase (СУБД FireBird) также имеется одноименная библиотека gds32.dll, которая не совместима с файлом библиотеки InterBase.

## Доступ к базе данных, компонент TIBDatabase

Компонент TIBDatabase предназначен для организации доступа клиентского приложения к базе данных, обслуживаемой сервером InterBase. Наличие компонента TIBDatabase в клиентском приложении обязательное условие, так как его услугами пользуются практически все остальные компоненты доступа к данным (рис. 15.1), для соединения с базой данных у них объявлено свойство Database.

### Соединение с базой данных

Современные версии Delphi позволяют создавать как многопользовательские клиент-серверные проекты InterBase, так и простые однопользовательские системы. За режим работы приложения отвечает свойство

```
property ServerType : string; //по умолчанию IBServer
```

Книга посвящена клиент-серверным БД, поэтому в наших проектах свойство должно быть установлено в состояние по умолчанию — IBServer.

Для организации соединения с базой данных необходимо указать имя БД. Для этого предназначено свойство

```
property DatabaseName: string
```

Имя определяется названием файла базы данных, а формат его описания зависит от того, под какой операционной системой выполняется СУБД InterBase и сетевого протокола, с помощью которого осуществляется доступ к серверу. В простейшем случае, когда база данных размещается на том же компьютере, на котором установлено наше приложение, мы ограничиваемся обычным путем к файлу

```
IBDatabase.DatabaseName:= 'c:\ibdata\warehouse.ib';
```

Но если InterBase выполняется на удаленном сервере, то порядок определения имени слегка усложняется. Для соединения с СУБД мы можем выбирать любой из наиболее распространенных сетевых протоколов: TCP/IP или NetBEUI. По возможности предпочтение стоит отдавать современному протоколу TCP/IP.

При подключении к сетевой БД с помощью TCP/IP, путь к файлу должен начинаться с указания сетевого имени компьютера

```
IBDatabase.DatabaseName:= 'IBHost:C:\IBData\warehouse.ib';
```

Если по какой-то причине стек протоколов TCP/IP недоступен, то организуем работу с помощью протокола NetBEUI, но в этом случае несколько изменится порядок описания имени базы данных.

```
IBDatabase.DatabaseName:= '\\IBHost\C:\IBData\warehouse.ib'';
```

Параметры соединения передаются при посредничестве свойства

```
property Params: TStrings;
```

Во время проектирования (чтобы избежать повторения процедуры регистрации при каждом перезапуске приложения) в это свойство можно внести данные об имени и пароле пользователя, при этом, не забыв отключить автоматический вызов диалога регистрации, установив в false свойство

```
property LoginPrompt : Boolean;
```

Для обращения к отдельным параметрам соединения можно обратиться к свойству

```
property DBParamByDPB[const Idx: Integer]: string;
```

В свойство следует передать имя соответствующей константы, например user_name. Все константы объявлены в модуле IBHeader.

Для очень забывчивых разработчиков может пригодиться метод

```
procedure CheckDatabaseName;
```

Он проверяет факт заполнения свойства DatabaseName, и если свойство пусто — вызывает исключительную ситуацию.

Завершив все подготовительные операции, вызываем метод

```
procedure Open;
```

либо переводим в true свойство:

```
property Connected : Boolean;
```

Ко всему прочему свойство Connected может использоваться для проверки факта соединения с БД (листинг 15.1).

**Листинг 15.1. Пример подключения к локальной БД**

```
procedure TfrmDataModule.DataModuleCreate(Sender: TObject);
begin
 with IBDatabase1 do
 begin
  DatabaseName:= 'C:\IBData\warehouse.ib';
  LoginPrompt:=False;
  Params.Clear;
  Params.Add('USER "SYSDBA"');    //имя пользователя
  Params.Add('PASSWORD "masterkey"');  //пароль
  Connected:=True;
 end;
end;
```

Такой код применяется только во время разработки проекта, а окончательная версия нашего программного продукта просто обязана потребовать от пользователя ввода регистрационных данных.

Для доступа к серверу InterBase необходимо указать имя и пароль пользователя. По умолчанию права системного администратора предоставит следующая учетная запись: пользователь "*SYSDBA*", пароль "*masterkey*".

Обычно программисты предпочитают управлять ходом соединения клиентского приложения с БД вручную в коде программы. Но зачастую контакт с базой данных необходим и во время разработки, например для инициализации свойств и параметров компонентов хранимых процедур или для генерации списка статических полей у компонентов наборов даных. После этого перед стартом программы важно не забыть вернуть свойство `Connected` в состояние `false`, в противном случае Delphi сгенерирует исключительную ситуацию. Если подключение/отключение надо произвести пару раз за день, то ничего страшного, но если проект перекомпилируется многократно… К счастью в компоненте `TIBDatabase` острота проблемы существенно снижена благодаря свойству

```
property AllowStreamedConnected : Boolean; //по умолчанию true
```

Свойство предназначено исключительно для упрощения процесса отладки проекта. Переведя свойство `AllowStreamedConnected` в состояние `false`, мы освобождаемся от необходимости постоянно включать/отключать `Connected` во время разработки. Компонент самостоятельно, без какого-то постороннего вмешательства, отсоединится от базы данных в момент старта проекта (ожидая явной команды на соединение в коде программы), и вновь подключится к БД после его остановки.

Еще одно полезное качество класса `TIBDatabase` — безопасное тестирование соединения. В процессе подготовки соединения перед вызовом метода `Open()` хорошей идеей может стать вызов проверочного метода

```
procedure TestConnected: Boolean;
```

Процедура проверит корректность параметров соединения и при успехе вернет значение true, при ошибке мы получим false, но (что очень важно) при этом не будет сгенерирована исключительная ситуация.

Для настройки параметров соединения во время разработки проекта программист может обратиться к контекстному меню компонента и воспользоваться пуктом "Database edittor" (рис.25.2).

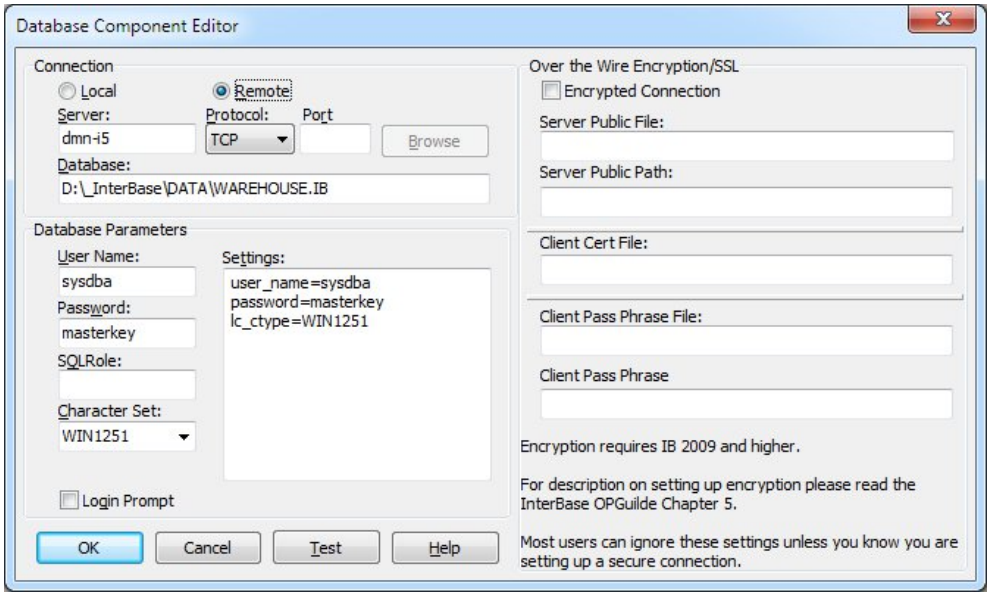



*Рис. 15.2. Редактор настройки соединения*

С процессом соединения связано два события

```
property BeforeConnect: TNotifyEvent; //перед началом соединения
property AfterConnect: TNotifyEvent;  //после успешного соединения
```

Они могут пригодиться для инициализации других компонентов проекта.

## Регистрация пользователя

Для вызова штатного диалога регистрации пользователя установите в `true` свойство `LoginPrompt`. В этом случае, обращение к методу `Open()` инициирует событие:

```
property OnLogin: TIBDatabaseLoginEvent;
TIBDatabaseLoginEvent = procedure(Database: TIBDatabase;
                                  LoginParams: TStrings) of object;
```

Регистрационные данные пользователя направляются параметр `LoginParams`.

Допустим, что у нас имеется отдельная форма `TfrmUserLogin` со строками ввода `edUser:TEdit` и `edPassword:TEdit` предназначенными для ввода имени и пароля пользователя. Тогда процесс регистрации можно организовать так, как предложено в листинге 25.2.

**Листинг 15.2. Регистрация пользователя на сервере InterBase**

```
procedure TfrmDataModule.IBDatabase1Login(Database: TIBDatabase;
                                          LoginParams: TStrings);
var frmUserLogin: TfrmUserLogin;
begin
  frmUserLogin:=TfrmUserLogin.Create(nil); {создаем форму}
  if frmUserLogin.ShowModal=mrOK then
    with LoginParams do
    begin
     Clear;
     Add('USER'+frmUserLogin.edUser.Text); //имя пользователя
     Add('password'+frmUserLogin.edPassword.Text); //пароль
    end;
    frmUserLogin.Release;
end;
```

### Разрыв соединения

Кроме анализа состояния свойства `Connect` для проверки наличия соединения с БД программист может использовать методы:

```
procedure CheckActive;   //проверка активности соединения
procedure CheckInactive; //проверка пассивности
```

Особенность процедур `Check…` в том, что при невыполнении условия проверки они генерируют исключительную ситуацию (листинг 25.3).

**Листинг 15.3. Шаблон кода завершения работы с БД**

```
uses IB;
…
procedure …
begin
try
  IBDatabase1.CheckInactive;
  {какие-то операции, осуществление которых возможно только
   при отсутствии соединения}
 except
   on EIBClientError do //проверяем класс ИС
      begin
           MessageBox(Handle,
                      PChar('Данная операция невозможна
                             при активном соединении!'),
                      PChar('Ошибка'),
                      MB_ICONERROR+MB_OK);
      end;
end;
end;
```

Для отключения от базы данных обращаемся к методу `Close()` либо переводим в `false` свойство `Connected`.

Для отключения всех активных наборов данных с сохранением соединения с БД вызываем метод
`procedure CloseDataSets;`

После обращения к методу все обслуживаемые нашим компонентом потомки `TDataSet` перейдут в пассивный режим (`Active=false`), но `TIBDatabase` по прежнему сохранит контакт с базой.

Процесс разрыва соединения сопровождается парой событий

```
property BeforeDisconnect: TNotifyEvent;
property AfterDisconnect: TNotifyEvent;
```

События возникают соответственно перед началом разрыва соединения, и по факту разрыва.

При необходимости можно ограничить время простоя соединения. Для этого предназначено свойство:

```
property IdleTimer: Integer; //по умолчанию 0 — неограниченно
```

Время исчисляется в миллисекундах. По умолчанию свойство установлено в ноль, это означает, что ограничения отсутствует. По истечению времени простоя генерируется событие:

```
property OnIdleTimer: TNotifyEvent;
```

Это событие может быть использовано для разрыва соединения с БД.

## Управление транзакциями, компонент TIBTranscation

Компонент TIBTransaction предоставляет услуги по управлению транзакциями из клиентского приложения с использованием одного или нескольких соединений с БД. Это обязательный ингредиент любого клиентского приложения InterBase, так как в услугах транзакций нуждаются все наборы данных (потомки класса TIBCustomDataSet) и компонент TIBSQL — выполняемые ими SQL-инструкции всегда должны выполняться в контексте транзакции.

Клиентское приложение должно как минимум содержать один компонент-транзакцию. Но единственного компонента достаточно только для простейших проектов. В реальности компонентов TIBTransaction должно быть ровно столько, сколько требует логика работы приложения. Конечно же, не стоит присоединять личную транзакцию к каждой таблице, хранимой процедуре и запросу InterBase, это окажется явным перебором. Имеет смысл выделить индивидуальные компоненты TIBTransaction для обслуживания наиболее сложных SQL команд и запросов.

Работа с компонентом обычно начинается с подключения к базе данных. Ссылка на базу данных по умолчанию находится в свойстве

```
property DefaultDatabase: TIBDatabase;
```

### Параметры транзакции

По умолчанию транзакция устанавливается в наиболее востребованное для большинства клиентских приложений InterBase состояние:

- уровень изоляции — SNAPSHOT;
- доступны операции записи и чтения;
- при обращении к заблокированному ресурсу транзакция переходит в режим ожидания освобождения блокировки.

Вполне естественно, что предустановленные значения не могут подойти ко всем случаям жизни, поэтому компонент TIBTransaction предоставляет нам возможность перенастроить поведение транзакции. Доступ из кода приложения к содержащему параметры буферу, обеспечивает свойство

```
property Params: TStrings;
```

или

```
property TPB: PChar; //только для чтения
```

В параметрах транзакции содержится информация об уровне изоляции транзакции, порядке использования точек сохранения, разрешение на редактирование данных. Для определения параметров транзакции проще всего воспользоваться услугами редактора транзакции, который вызывается из контекстного меню компонента `TIBTransaction` (рис. 15.3).

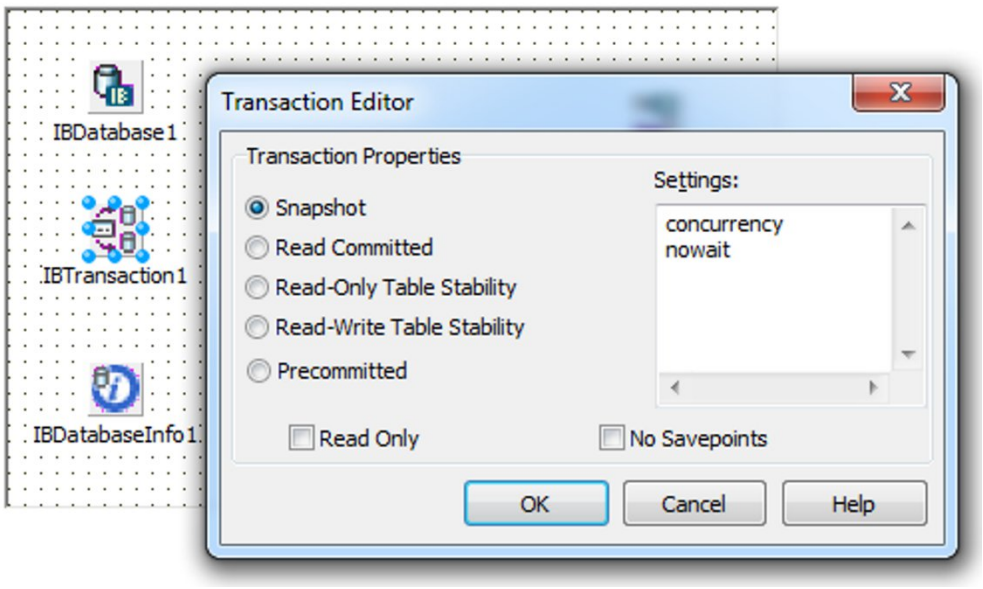



*Рис. 15.3. Окно редактора параметров транзакции*

Как уже упоминалось на лекции уровни изоляции транзакций InterBase несколько отличаются от требований стандарта SQL. вы можете выбрать наиболее подходящий уровень просто щелкнув мышкой по соответствующей кнопке выбора:

- самую низкую степень изолированности обеспечит уровень Read Committed (неповторяемое чтение).
- уровень Precommitted соответствует уровню Read Committed в режиме только для чтения
- уровень Snapshot (моментальный снимок) соответствует средней степени изоляции.
- наибольшую степень изоляции обеспечивают уровни воспроизводимого чтения Read-Only Table Stability и Read-Write Table Stability, в режиме Read-Only операторы внутри транзакции не могут модифицировать данные, в режиме Read-Write — модификации допускаются.

## Автоматическое управление транзакцией

Компонент TIBTransaction достаточно умен для того, чтобы самостоятельно начинать и завершать выполнение транзакции. Степень доверия программиста к компоненту определяется свойствами

```
property AllowAutoStart : Boolean; //по умолчанию true
property AutoStopAction : TAutoStopAction; //по умолчанию saNone
```

Свойство `AllowAutoStart` установленное в состояние true указывает компоненту на то, что при необходимости он должен самостоятельно запустить транзакцию. Свойство `AutoStopAction` определит действие по умолчанию в момент завершения транзакции. Варианты допустимых значений `TAutoStopAction` представлены в таблице 15.1.

*Таблица 15.1. Возможные значения TAutoStopAction*

| ***Значение*** | ***Описание*** |
|---|---|
| `saNone` | Компонент не может самостоятельно управлять завершением транзакций |
| `saRollback` | Компонент имеет право на самостоятельность при откате транзакции |
| `saCommit` | Компоненту разрешено фиксировать транзакцию |
| `saRollbackRetaining` | Возврат к точке сохранения. Поддерживается, начиная с InterBase 6.0 |
| `saCommitRetaining` | Транзакция не завершается пока есть незакрытые наборы данных. Все изменения в связанных с транзакцией данных отменяются. Другими словами, производится откат транзакции с сохранением ее контекста<br>Транзакция не завершается до тех пор, пока имеются незакрытые наборы данных. Все изменения в охваченных транзакцией записях фиксируются. Контекст транзакции сохраняется |

Если транзакция обслуживает компонент `TIBSQL`, то свойство `AutoStopAction` должно быть установлено в состояние `saNone`.

Если транзакция обслуживает наборы данных (компоненты TIBTable, TIBQuery или TIBDataSet) то заложенное в свойство AutoStopAction отработает только после того, как будет закрыт связанный с транзакцией набор данных. Другими словами, все изменения, сделанные набором данных в рамках транзакции, не зафиксируются до тех пор пока набор данных остается активным.

Для принудительного вызова действия, указанного в свойстве AutoStopAction следует вызвать метод

```
procedure CheckAutoStop;
```

В автоматический режим управления транзакцией лучше всего ставить тот компонент TIBTransaction на который компонент-соединение с БД TIBDatabase ссылается с помощью свойства DefaultTransaction.

## Управление транзакцией в ручном режиме

В тех ситуациях, когда автоматическая работа транзакции нежелательна программисту следует брать бразды правления компонентом TIBTransaction в свои руки. В первую очередь для этого в инспекторе объектов следует отключить свойства AllowAutoStart и AutoStopAction и приступить к программированию.

Первое правило с проверки которого должны начинаться стоки кода заключается в проверке активности компонента–транзакции. Если в текущий момент времени выполняется стартовавшая ранее транзакция, то попытка вновь воспользоваться услугами компонента может отменить предыдущую транзакцию. Об активности судят по свойству

```
property Active: Boolean;
```

Если свойство возвратит true, то компонент лучше пока не трогать — он в работе. Заметим, что свойство доступно и для записи, оно позволяет стартовать транзакцию даже во время визуального проектирования, поэтому не стоит злоупотреблять доброжелательностью компонента (это может сослужить плохую службу), тем более что перевод Active в состояние false аварийно завершает текущую транзакцию вызывая метод Rollback. Поэтому будем полагать, что свойство Active следует использовать только в режиме для чтения.

Альтернативным способом проверки состояния транзакции может стать обращение к свойству

```
property InTransaction: Boolean; //только для чтения
```

Значение true свидетельствует, что транзакция еще не завершена.

Убедившись, что транзакция не активна, для явного запуска новой транзакции обращаемся к процедуре

```
procedure StartTransaction;
```

При нормальном завершении транзакции вызываем метод

```
procedure Commit;
```

фиксирующий изменения в БД, в противном случае, для отката транзакции идем на поклон к методу

```
procedure Rollback;
```

Предложенный в листинге 15.4 фрагмент кода раскрывает порядок управления транзакцией, в состав которой входит две хранимых процедуры.

**Листинг 15.4. Пример работы с компонентом TIBTransaction**

```
if IBTransaction1.Active=False then
 begin
 IBTransaction1.StartTransaction; //старт транзакции
 try
  with IBStoredProc1 do // операция с 1-ой процедурой
    begin
    //заполняем параметры процедуры
    ExecProc;
    end;

  with IBStoredProc2 do //операция со 2-ой процедурой
    begin
    //заполняем параметры процедуры
    ExecProc;
    end;
  IBTransaction1.Commit;    //фиксируем транзакцию
 except                     //действия при возникновении ошибки
```

```
  IBTransaction1.Rollback; //откат транзакции
 end;
end;
```

> Методы `Commit()` и `Rollback()` завершают выполнение транзакции и освобождают ее контекст. После вызова любого из перечисленных методов возвратится к транзакции невозможно

Если мы работаем с InterBase не ниже 6 версии, то для отката всех изменений в данных, осуществленных в рамках текущей транзакции, разрешено воспользоваться процедурой

```
procedure RollbackRetaining;
```

При этом сохраняется контекст транзакции, другими словами транзакция не завершается и может быть продолжена. Для фиксации изменений осуществленных транзакцией с сохранением ее контекста используйте метод

```
procedure CommitRetaining;
```

Еще раз заострю внимание читателя на том, что метод не завершает транзакцию, а просто сохраняет все изменения в текущей версии. Транзакция останется открытой до тех пор, пока не будет вызван метод `Commit()` или `Rollback()`.

## Задание

С использованием языка Delphi (C++ Builder) разработайте клиентское приложение VCL способное соединиться с БД InterBase (Firebird). Приложение должно позволять пользователю:

1. выбирать/вводить имя сервера;
2. указывать путь и имя с файлом БД;
3. вводить имя и пароль пользователя;
4. подготовьте к работе компонент-транзакцию, он понадобится для выполнения следующих работ.

# 16. Основные компоненты IBX

*Вид занятия – практическое занятие*
*Время занятия – 2 часа*
*Программное обеспечение – среда InterBase (FireBird), C++ (Delphi)*

## Компонент быстрой разработки TIBTable

Компонент–таблица TIBTable предоставляет возможность создания простейших клиент-серверных приложений за кратчайшее время с минимальными затратами на программирование. Для простейшего приложения понадобится помощь шести компонентов (рис. 26.1):

1. Компонент `IBDataBase1` отвечающий за соединение с БД.
2. Транзакция `IBTrandaction1` обеспечит доступ к контексту транзакции.
3. Компонент `IBTable1` предоставит доступ к таблице.
4. Источник данных `DataSource1` (страничка Data Access).
5. Навигатор данных `DBNavigator1` (страничка Data Controls).
6. Сетка для отображения данных `DBGrid1` (страничка Data Controls).

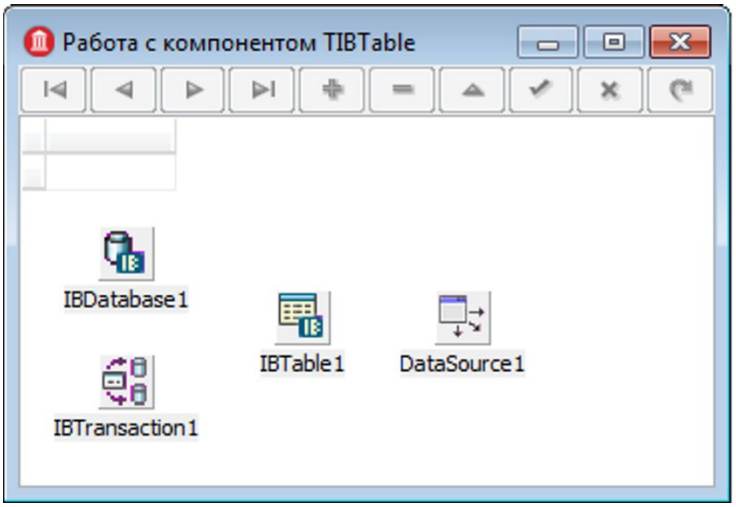



*Рис. 16.1. Состав простейшего приложения БД на основе TIBTable*

Разработка клиентского приложения всегда начинается с организации соединения с базой данных — эту задачу вы решили в предыдущей лабораторной работе.

Выберите компонент-транзакцию `IBTrandaction1` и воспользовавшись свойством `DefaultDatabase` присоедините его к `IBDataBase1`.

Переходим к компоненту `IBTable1`. С помощью свойства `Database` подключите его к базе данных, затем задействовав свойство `TableName` и соединяемся с таблицей. Настройка компонента завершена, нам осталось только активировать его, для этого устанавливаем свойство компонента `Active` в состояние `true`.

“Программирование” завершено. Нам осталось нажать кнопку <F9>. Наш проект базы данных готов к работе!

## Запрос, компонент TIBQuery

Запрос `TIBQuery` построен на базе рассмотренного ранее `TIBCustomDataSet` и описан в модуле `IBQuery`. Основное назначение компонента — организация доступа к данным одной или нескольких таблиц. Как и у всех подобных элементов управления, текст запроса описывается в свойстве

```
property SQL: TStrings;
```

Текст запроса может быть задан как во время проектирования (рис. 16.2), так и в коде программы.

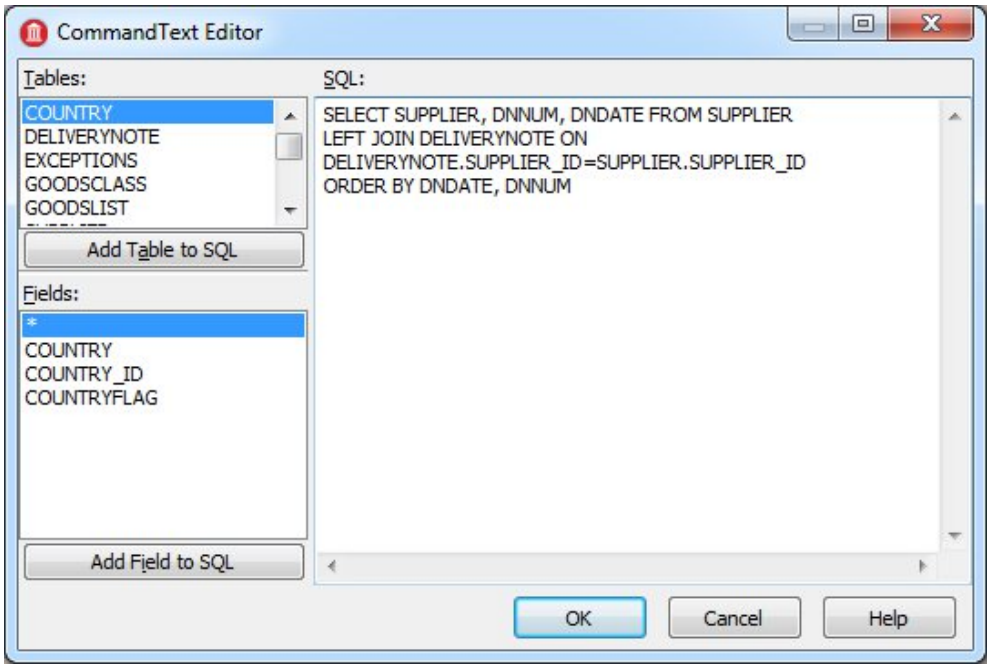



*Рис. 16.2. Встроенный редактор SQL с текстом статического запроса*

Если запрос основан на инструкции SELECT, то для получения записей следует воспользоваться методом

```
procedure Open;
```

во всех остальных случаях выполнение инструкции ISQL доверяется процедуре

```
procedure ExecSQL;
```

Следующий пример (листинг 26.1) подготавливает компонент к получению данных из таблиц поставщиков и заказчиков.

**Листинг 16.1. Пример работы с компонентом TIBQuery – статическая инструкция**

```
with IBQuery1 do
begin
  Database:=IBDatabase1;
  SQL.Clear;
  SQL.Add('SELECT SUPPLIER, DNNUM, DNDATE FROM SUPPLIER');
  SQL.Add('LEFT JOIN DELIVERYNOTE ON');
  SQL.Add('DELIVERYNOTE.SUPPLIER_ID=SUPPLIER.SUPPLIER_ID');
  SQL.Add('ORDER BY DNDATE, DNNUM');
  Prepare;
  Open;
end;
```

Возвращаемый набор данных доступен только в режиме “для чтения”. Дабы добиться возможности его редактирования нам понадобятся услуги `TIBUpdateSQL`, подключаемого к свойству

```
property UpdateObject: TIBDataSetUpdateObject;
```

Это ссылка на компонент TIBUpdateSQL, способный оказать помощь при работе с недоступными для редактирования данными.

Создатели VCL научили работающие с SQL компоненты-наборы данных передавать в инструкцию-SQL внешние параметры без необходимости модификации текста всего запроса. В проектах, опирающихся на применение параметров, текст SQL-запроса формируется единожды. Это может произойти во время проектирования или, как в очередном примере (листинг 16.2) во время выполнения проекта.

**Листинг 16.2. Динамическая инструкция SQL с параметрами**

```
with IBQuery1 do
begin
  SQL.Clear; //очистка свойства
  SQL.Add('INSERT INTO SUPPLIER'); //формируем инструкцию
```

```
  SQL.Add('(SUPPLIER) VALUES :SUPPLIER ');//объявляем параметр
  Prepare; //подготовка запроса на сервере
end;
```

Обратите внимание на то, что в строке `SQL "(SUPPLIER) VALUES :SUPPLIER"` вместо явного указания имени нового поставщика мы передаем начинающееся с символа двоеточия ":" имя параметра.

Если запрос содержит параметры, то для их обслуживания следует воспользоваться коллекцией

```
property Params: TParams;
```

Для того, чтобы компонент автоматически регенерировал список параметров запроса следует установить в true свойство. Теперь, для отправки в БД команды на добавление новой строки, достаточно следующих строк кода (листинг 16.3).

**Листинг 16.3. Заполнение параметра новым значением**

```
IBQuery1.Params.ParamByName('SUPPLIER'):=Edit1.Text;
IBQuery1.ExecSQL;
```

Таким образом, применение параметров в SQL запросе существенно упрощает процесс разработки приложения и, что очень важно, ускоряет отправку и выполнение запроса на сервере.

## Хранимая процедура, компонент TIBStoredProc

Компонент `TIBStoredProc` предназначен для обращения к откомпилированным и хранящимся в системном каталоге сервера БД процедурам. Название процедуры, услугами которой мы намерены воспользоваться указываются в свойстве

```
property StoredProcName: string;
```

Параметры процедуры содержатся в коллекции

```
property Params: TParams;
```

О готовности процедуры к выполнению сигнализирует свойство

```
property Prepared: Boolean;
```

Для отправки команды на выполнение обращаемся к процедуре

```
procedure ExecProc;
```

> К сожалению, компоненты `TIBStoredProc` не любят работать с хранимыми процедурами возвращающими набор из нескольких записей. Поэтому, несмотря на наличие у `TIBStoredProc` метода `Open()`, основной способ отправки команды на выполнение должен быть `ExecSQL()`. Вместе с тем никто не запрещает нам создавать в InterBase хранимые процедуры, основанные на цикле `FOR SELECT … DO`, но в этом случае для получения результатов выборки на стороне клиента следует задействовать не компонент `TIBStoredProc`, а компонент `TIBQuery`. В этом случае в текст SQL TIBQuery надо просто записать инструкцию `SELECT * FROM <имя_процедуры>` и вызвать метод `Open()`.

Допустим, что в составе БД имеется хранимая процедура `PR_SUPPLIER_INSERT` вставляющая новую запись в таблицу поставщиков (листинг 26.4).

**Листинг 16.4. Код хранимой процедуры**

```
CREATE PROCEDURE PR_SUPPLIER_INSERT
(ASUPPLIER VARCHAR(100))
RETURNS
(ASUPPLIER_ID INTEGER)
AS
begin
 INSERT INTO SUPPLIER (SUPPLIER) VALUES (:ASUPPLIER);

 SELECT SUPPLIER_ID FROM SUPPLIER
 WHERE SUPPLIER=:ASUPPLIER INTO :ASUPPLIER_ID;
end
```

Процедура обладает единственным входным параметром ASUPPLIER, в который следует передать имя нового поставщика и одним выходным параметром ASUPPLIER_ID возвращающим значение первичного ключа только-что добавленной записи.

Для подготовки компонента TIBStoredProc к работе с процедурой PR_SUPPLIER_INSERT следует воспользоваться инспектором объектов или фрагментом кода из листинга 26.5.

**Листинг 16.5. Подключение компонента TIBStoredProc к хранимой процедуре**

```
with IBStoredProc1 do
begin
  Database:=IBDatabase1;
  Transaction:=IBDatabase1.DefaultTransaction;
  StoredProcName:='PR_SUPPLIER_INSERT';
end;
```

Для выполнения процедуры PR_SUPPLIER_INSERT достаточно обратиться к листингу 26.6.

**Листинг 16.6. Вызов хранимой процедуры**

```
var ID:integer;
begin
  with IBStoredProc1 do
  begin
    Params.ParamByName('ASUPPLIER').AsString:=Edit1.Text;
    ExecProc;
    ID:= Params.ParamByName('ASUPPLIER_ID').AsInteger;
  end;
end;
```

В представленном коде мы передаем новое значение в параметр ASUPPLIER и выполняем процедуру. После выполнения процедуры в переменную ID помещается очередное значение первичного ключа.

## Универсальный набор данных TIBDataSet

Классический представитель наборов данных класс TIBDataSet фактически способен заменить всех остальных своих коллег, базирующихся на фундаменте класса TIBCustomDataSet. При желании с помощью компонентов TIBDataSet мы сможем разработать полноценное клиентское приложение любой степени сложности без использования таблиц TIBTable, запросов TIBQuery и хранимых процедур TIBStoredProc.

Ключевая особенность компонента заключается в том, что он предоставляет программисту возможность тонкой настройки всех наиболее важных операций (просмотр, вставка, редактирование и удаление данных). Еще одна положительная черта компонента заключается в том, что полученный от сервера набор строк буферизируется на клиентской стороне, что позволяет нам применять все преимущества объектно-ориентированного подхода при работе с реляционными данными.

### Формирование запросов

Универсальность компонента TIBDataSet достигается за счет инкапсуляции в него целых пяти объектов TIBSQL, каждый из которых отвечает за свой фронт работы:

- выборку данных;
- обновление отдельной записи;
- вставку новой записи;
- редактирование записи;
- удаление записи.

Прямого обращения к инкапсулированным классам не требуется — все необходимые рычаги управления предоставлены в форме свойств и методов головного класса TIBDataSet. В первую очередь выделим свойства, несущие ответственность за хранение текста запросов.

```
property SelectSQL: TStrings;  //запрос на выборку данных
property RefreshSQL: TStrings; //запрос обновления
property InsertSQL: TStrings;  //запрос вставки
property ModifySQL: TStrings;  //запрос редактирования
property DeleteSQL: TStrings;  //запрос удаления
```

Во время визуального проектирования доступ к перечисленным свойствам реализован как традиционно через инспектора объектов, так и через встроенный редактор для вызова которого необходимо обратиться к контекстному меню компонента и выбрать пункт **Dataset Editor**. В простейшем случае разработчик может просто нажать на кнопку **Generate SQL** (рис. 16.3) в результате чего будет автоматически сгенерированы и сохранены в соответствующих свойствах компонента все пять запросов.

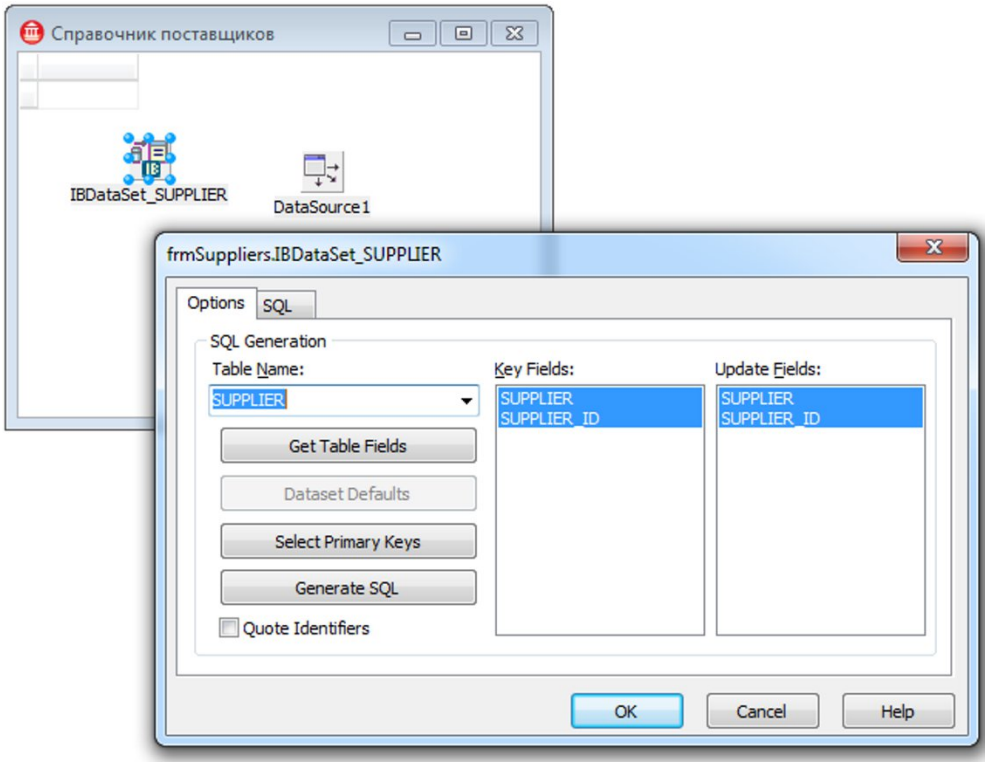



*Рис. 16.3. Встроенный редактор текста запросов*

Если вы сторонник управления текстом запроса непосредственно из кода программы, то обратите внимание на листинг 16.7 в котором мы готовим компонент IBDataSet1 к работе с таблицей поставщиков SUPPLIER.

**Листинг 16.7. Подготовка текста SQL во время работы приложения**

```
procedure TForm1.FormShow(Sender: TObject);
begin
With IBDataSet1 do
begin
  SelectSQL.Clear;
  SelectSQL.Add('SELECT * FROM SUPPLIER ORDER BY SUPPLIER');
  RefreshSQL.Clear;
  RefreshSQL.Add('SELECT * FROM SUPPLIER');
  RefreshSQL.Add('WHERE SUPPLIER_ID=: SUPPLIER_ID ORDER BY SUPPLIER');
  InsertSQL.Clear;
  InsertSQL.Add('INSERT INTO SUPPLIER (SUPPLIER,SUPPLIER_ID)');
  InsertSQL.Add('VALUES (:SUPPLIER,:SUPPLIER_ID)');

  ModifySQL.Clear;
  ModifySQL.Add('UPDATE SUPPLIER SET SUPPLIER=:SUPPLIER');
  ModifySQL.Add('WHERE SUPPLIER_ID=:OLD_SUPPLIER_ID');
  DeleteSQL.Clear;
  DeleteSQL.Add('DELETE FROM SUPPLIER');
  DeleteSQL.Add('WHERE SUPPLIER_ID=:OLD_SUPPLIER_ID');
  //настроим поле генератора значения первичного ключа
  GeneratorField.ApplyEvent:=gamOnServer;
  Open;
end;
end;
```

Просматривая листинг обязательно обратите внимание на то, что в запросах модификации и удаления строки при объявлении параметра мы воспользовались префиксом “OLD_”. Дело в том, что в OLD_ полях хранится исходное (не модифицированное) значение данного столбца таблицы. В нашем случае OLD_SUPPLIER_ID хранит значение первичного ключа редактируемой записи.

> Компонент `TIBDataSet` автоматически создает дополнительные параметры, названия которых строятся по принципу “OLD_”+“ИМЯ_СТОЛБЦА” и “NEW_”+ “ИМЯ_СТОЛБЦА”. Параметры, имена которых начинаются с префикса “OLD_” хранят значения столбцов до их изменения пользователем, а параметры “NEW_” — значения после изменения. Эти параметры можно использовать при формировании текста запросов.

## Задание

На основе созданного во время выполнения предыдущей лабораторной работы клиентского приложения создайте 2-3 формы, демонстрирующие порядок просмотра и редактирования данных в 2-3 таблицах вашей БД. В лабораторной работе должны быть продемонстрированы варианты использования следующих компонентов:

- таблицы `TIBTable`;
- запроса `TIBQuery`;
- хранимой процедуры `TIBStoredProc`;
- набора данных `TIBDataSet`.

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**
**Федеральное государственное автономное образовательное учреждение высшего образования**
**«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

# ТЕХНОЛОГИИ ПРОЕКТИРОВАНИЯ И АДМИНИСТРИРОВАНИЯ БАЗ ДАННЫХ

## МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К САМОСТОЯТЕЛЬНОЙ РАБОТЕ СТУДЕНТОВ

Направление подготовки 10.05.01 «Компьютерная безопасность»
Направленность (профиль): «Информационно-аналитическая и техническая экспертиза компьютерных систем»
Квалификация выпускника – специалист
Год начала обучения - 2026 г.
Форма обучения - очная

Ставрополь
2026

**Пояснительная записка**

В ходе изучения дисциплины Технологии проектирования и администрирования баз данных студенты должны выполнить домашнее задание, способствующее приобретению навыков проектирования баз данных и работы с реляционными БД. Студенту предлагается смоделировать предметную область и спроектировать базу данных для БД. По итогам выполнения работы преподавателю предоставляется набор скриптов, разработанная студентом программа и оформленный отчёт с полученными результатами.

Текущий контроль усвоения знаний осуществляется путем прохождения 6 контрольных точек, подготовки и сдачи отчетов по итогам выполнения лабораторных работ, проверки выполнения домашнего задания, опросов на практических занятиях.

**План-график выполнения СРС по дисциплине**

| № контрольной точки | Вид самостоятельной работы | Форма контроля | Зачетные единицы (часы) |
|---|---|---|---|
| | **6 семестр** | | |
| 3, 4 | Подготовка к лабораторным занятиям | Защита лаб.работ | 25 |
| 3 | Самостоятельное изучение темы | Тестирование | 25 |
| 4 | Написание доклада | Проверка | 12 |
| | **Итого за 6 семестр** | | **64** |
| | | | |
| | **Итого** | | **64** |

## Методические рекомендации к СРС

**Раздел 1. Основы реляционных БД**: Формирование SQL запросов, обеспечивающих
выполнение второго раздела задания, связанного с формированием отчетов. Демонстрация преподавателю отработанных запросов.
**Задание**: Написание разделов пояснительной записки, связанных с формированием посредством SQL запросов отчетов по предметной области.
**Литература**: О[1, 2].

**Радел 2. Проектирование БД**
**Основные понятия**: Построение модели сущность-связь. Написание скриптов создания таблиц базы данных. Создание таблиц и наполнение их небольшим (5-10) количеством записей.
**Задание**: Написание (в Word) разделов пояснительной записки, связанных с проектированием модели сущность-связь, созданием таблиц БД и наполнением их информацией.

**Литература**: О[1-3].
**Раздел 3. Современные технологии проектирования ПО** Формирование SQL запросов.
**Задание** Написание скриптов создания таблиц базы данных.
**Литература**: Д[1-6].

## Форма контроля СРС

Качество усвоения знаний студентами осуществляется в форме тестирования, индивидуального собеседования, проверки доклада, реферата и проверки индивидуального творческого задания.

## Задания для СРС

### Примерные темы рефератов

1. История развития, назначение и роль баз данных
Содержание реферата: Этапы развития информационных систем. Понятие базы данных. Файловые системы и базы данных. Классификация задач, решаемых с использованием СУБД.

2. Физическая организация баз данных
Содержание реферата: Структуры данных и базы данных. Способы хранения информации в базах данных. Способы повышения эффективности обработки данных за счет их организации. Инвертированные файлы.

3. Общие принципы построения СУБД
Содержание реферата: Общая характеристика, назначение, возможности, состав и архитектура СУБД. Классификация СУБД. Информационное, лингвистическое, математическое, аппаратное, организационное, правовое обеспечения СУБД.

4. Средства поддержания целостности базы данных
Содержание реферата: Метаданные. Словарь-справочник данных. Ссылочная целостность. Механизм транзакций. Управление доступом. Средства дублирования и восстановления. Особенности реализации баз данных с высокими требованиями на надежность хранения и обработки.

5. Эксплуатация баз данных
Содержание реферата: Состав, порядок планирования и проведения регламентных работ. Сервисные средства СУБД. Задачи администратора базы данных. Организация труда обслуживающего персонала.

6. Технология и модели архитектуры клиент/сервер
Содержание реферата: Достоинства и недостатки моделей архитектуры

клиент/сервер и их влияние на функционирование сетевых СУБД.

7. Серверы баз данных
Содержание реферата: Использование средств прямого ввода-вывода, управления памятью, поддержания целостности, защиты от сбоев. Возможности по обработке неструктурированных данных большого объема (Oracle Multimedia Server).

8. Серверы баз данных
Содержание реферата: Поддержка Internet (Oracle Web Server). Оценка эффективности и адаптации функционирования сервера баз данных (тесты производительности). Проблемы оптимизации доступа к базе данных.

9. Клиентская часть архитектуры клиент/сервер
Содержание реферата: Средства поддержания интерфейса с различными категориями пользователей. Языки запросов. Языки описания данных. Языки манипулирования данными. Стандарты Xbase, SQL. Языки четвертого поколения 4GL.

10.Клиентская часть архитектуры клиент/сервер
Содержание реферата: Интерфейс языков СУБД с языками программирования. Средства реализации диалогового интерфейса и подготовки отчетов в языках СУБД. Стандарты на графический пользовательский интерфейс GUI.

11.Интерфейс между клиентом и сервером.
Содержание реферата: Протоколы согласованной работы. Распределенные базы данных в сетях ЭВМ. Средства интеграции и взаимодействия разнородных распределенных баз данных.

12.Централизация логики приложения на сервере базы данных
Содержание реферата: Создание и использование процедур, функций, триггеров, пакетов. Программные утилиты СУБД Oracle11.

13.Автоматизированное проектирование
Содержание реферата: Средства автоматизации проектирования баз данных: общая характеристика, назначение и возможности, классификация, универсальные и специализированные генераторы программ для СУБД.

14.Объектно-ориентированное программирование в СУБД
Содержание реферата: Принципы объектно-ориентированного программирования. Недостатки реляционных СУБД. Объектные расширения реляционных СУБД. Объектно-реляционные адаптеры. Объектно-реляционные СУБД. Объектные СУБД. Стандарты на объектные СУБД.

15. Многоплатформные СУБД
Содержание реферата: СУБД Oracle, Informix, Sybase, DB2. Область применения. Особенности их реализации. Сетевые компоненты многоплатформных СУБД. Требования по их эксплуатации.

16. СУБД, ориентированные на конкретные платформы
Содержание реферата: СУБД DBManager в OS/2. SQL/400 в AS/400. СУБД Access в Microsoft Windows. Связь компонентов СУБД с особенностями операционной среды. Аппаратная поддержка управления данными. Использование возможностей пакетов прикладных программ конкретных платформ совместно с СУБД. Средства распределенной обработки данных.

17. СУБД семейства InterBase
Содержание реферата: InterBase, Yaffi, Firebird. История развития и причины популярности СУБД данного семейства. Языки программирования и языки манипулирования данными в СУБД семейства. Проектирование и эксплуатация информационных систем. Механизмы блокирования и управления доступом в многопользовательской среде.

### Примерные темы докладов

5. Управление данными во внешней памяти.
6. Управление буферами оперативной памяти.
7. Управление транзакциями.
8. Журнализация.
9. Поддержка языков БД.
10. Типовая организация современной СУБД.
11. Основные особенности систем, основанных на инвертированных списках.
12. Структуры данных.
13. Файловая система.
14. Ограничения целостности БД.
15. Иерархические системы.
16. Иерархические структуры данных.
17. Нормальные формы.
18. Сериализация транзакций.
19. Сетевые системы.
20. Сетевые структуры данных.
21. Манипулирование данными.
22. Модели клиент\сервер.
23. Достоинства и недостатки реляционных СУБД.
24. Перспективы развития баз данных.

## Примерные темы индивидуальных творческих заданий

Разработайте базу данных:

1. “База данных поликлиники”. Учёту подлежат: больные (ФИО, домашний адрес), лечащий персонал (ФИО, специальность), дата посещения больным врача и поставленный диагноз.
2. “База данных футбольных матчей”. Учёту подлежат: спортивный клуб, игроки команд (ФИО), дата матча, игравшие команды, забитые мячи, авторы голов.
3. “База данных “Бюро по туризму”. Учёту подлежат: клиенты фирмы (ФИО, домашний адрес), список курортов с ценами и продолжительностью путёвок, даты поездок клиентов на курорт.
4. “База данных “Магазин видеофильмов”. Учёту подлежат: название фильма, жанр, год выхода, киностудия, режиссёр, цена, носитель (DVD/Видеокассета), число экземпляров.
5. “База данных “Домашняя фонотека”. Учёту подлежат: исполнитель, страна к которой принадлежит исполнитель, название альбома, год выхода альбома, музыкальный жанр альбома, наименование и продолжительность музыкальных композиций входящих в альбом, формат хранения (CD, mp3, …).
6. База данных “Склад автомобильных запчастей”. Учёту подлежат: название детали, производитель, цена, в какой марке автомобиля применяется, поставщик детали.
7. База данных “Метеостанция”. Учёту подлежат: названия станций, их географические координаты, персонал станций (ФИО, должность), каждая метеостанция ежечасно сохраняет данные о температуре окружающей среды.
8. База данных “Библиотека”. Учёту подлежат: жанр книги, авторы книги, название книги, количество экземпляров, место хранения книги, читатель (ФИО и адрес), дата получения читателем, дата возврата книги.
9. База данных “Склад телевизоров”. Учёту подлежат: поставщик, производитель телевизора, модель телевизора, дата поставки телевизора, дата списания телевизора со склада, не менее 5 характеристик телевизора (производитель, диагональ, вес, и т.п.), цена телевизора от поставщика, отпускная цена телевизора, количество телевизоров.
10. База данных “Автовокзал.” Учёту подлежат: автобусы (марка, дата выпуска и государственный регистрационный номер), водители (ФИО, дата рождения, водительская категория, какой автобус закреплён), расписание движения автобусов (наименование населённого пункта, день и время отправления, день и время прибытия, стоимость билета, автобус выполняющий рейс).
11. База данных “Магазин одежды.” Учёту подлежат: товар (название, размер, цена, материал, производитель), дата поступления, дата продажи.
12. База данных “Почтовое отделение” Учёту подлежат: поступающая на почту заказная корреспонденция (письма, ценные бандероли и посылки). Дата поступления, дата выдачи, паспортные данные получателя корреспонденции.
13. База данных “Аптечный киоск” Учёту подлежат: лекарства (наименование, дозировка, производитель, отпускная цена, место хранения, выдача только по рецепту или нет, показания и противопоказания).
14. База базы данных “Расписание занятий в институте” Учёту подлежат: специальность, учебная группа, учебные дисциплины (название и вид занятия), день недели, номера часов проведения занятий, преподаватель, номер аудитории
15. База данных “Гостиница” Учёту подлежат: номера гостиницы (номер помещения, число спальных мест, наличие душа, туалета, холодильника, телевизора, сплит-системы, стоимость проживания), занятость номера (дата заселения, паспортные данные жильцов, дата освобождения номера)
16. База данных “Кинотеатр”. Учёту подлежат: название кинофильма, жанр кинофильма, возрастные ограничения, дата и время киносеанса, название кинозала, номер посадочного места, цена билета.

17. База данных “Торговый зал”. Учёту подлежат: название товара, отпускная цена товара, количество товара в зале, дата выпуска, срок годности. База данных должна предусматривать отпуск товара покупателю со списанием товара из зала.
18. База данных “Учёт электроэнергии”. База учитывает количество потреблённой электроэнергии в небольшом посёлке и ежемесячно подготавливает индивидуальные квитанции об оплате. Учёту подлежат: адрес дома, владелец дома, количество потреблённой электроэнергии за месяц, стоимость за кВт/ч. Стоимость электроэнергии может изменяться!
19. База данных “Отдел кадров завода”. Учёту подлежат: ФИО, рабочая специальность, квалификация (разряд), образование работника, дата трудоустройства, стаж работы, должностной оклад, надбавка (зависит от квалификации и стажа).
20. База данных “Художественный музей” Учёту подлежат: автор картины, название картины, краткое описание сюжета, дата написания картины, художественное направление (реализм, абстракция, и т.д.), зал в котором выставлена картина.

Требования к представлению и оформлению результатов СРС

База данных разрабатывается на основе Microsoft Access (или на другой БД по согласованию с преподавателем). БД должна быть нормализована до 4 нормальной формы.
Клиентское приложение разрабатывается на языках Delphi, C++ Builder, C# или на другом языке по согласованию с преподавателем.
Приложение должно включать формы ввода данных, формы просмотра и редактирования данных. В рабочих формах приложения должна быть предусмотрена возможность:

1. Упорядочивания данных;
2. Фильтрации данных;
3. Поиску данных.

Программа должна предоставлять возможность вывода отчётов. В отчётах должны быть предусмотрены возможности:

- Вывода всех данных из всех таблиц БД;
- Использования агрегирующих функций SQL (Avg, Count, Min, Max, и т.д.).

Бумажный отчёт курсовой работы оформляется по требованиям установленным в ВУЗе. Отчёт должен содержать:

- ER-модель БД;
- Полную спецификацию таблиц.
- Примеры используемых в работе SQL запросов.
- Описание клиентского приложения.
- Инструкцию по работе с приложением.

Объём отчёта 25-30 стр. К отчёту прилагается CD-диск с БД и исходным кодом проекта. Допускается один CD- на несколько человек или всю учебную группу.

## Рекомендуемая литература и интернет-ресурсы

Основная литература:

1. Хомоненко А.Д. и др. Базы данных. Учебник для высших учебных заведений – СПб.: КОРОНА принт, 2010. – 736 с.
2. Б.Я. Советов, В.В. Цехановский. Базы данных: теория и практика. 2-е издание, учебник для бакалавров - М.: ЮРАЙТ-ИЗДАТ, 2011. – 463 с.
3. В.Ю. Пирогов Информационные системы и базы данных: организация и проектирование – СПб.: Издательский дом БХВ, 2009. – 528 с.: ил.

Дополнительная литература:

1. Конноли, Бегг. Базы данных. Проектирование, реализация и сопровождение. Теория и практика. 3-е издание. М.: Вильямс, 2003, - 1436 с.
2. Дейт К. Дж. Введение в системы баз данных, 8-е издание.: Пер.с англ. - М. Издательский дом “Вильямс”, 2005,- 1328с., ил.
3. Марков А.С., Лисовский К.Ю. Базы данных. Введение в теорию и методологию: Учебник. – М.: Финансы и статистика, 2006. – 512 с., ил.
4. Осипов Д. Delphi. Профессиональное программирование. Профессиональное программирование. – СПб.: Символ-Плюс, 2006. – 1056 с.: ил.
5. Осипов Д. Базы данных и Delphi. Теория и практика. – СПб.: Издательский дом БХВ, 2011. – 720 с.: ил.
6. Основы проектирования и разработки реляционных баз данных: Учебное пособие. – Ставрополь: Изд-во СГУ, 2007. – 203 с.

Интернет-ресурсы:

1. www.microsoft.com – сайт компании Microsoft
2. www.embarcadeo.com – сайт компании Embarcadero
3. http://algolist.manual.ru/ - сайт посвящённый разработке алгоритмов