МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Государственное образовательное учреждение высшего
профессионального образования
«Оренбургский государственный университет»

Ю.В. ПОЛИЩУК, С.И. СОРМОВ, Т.А. ЧЕРНЫХ

ПРОЕКТИРОВАНИЕ РЕЛЯЦИОННЫХ БАЗ ДАННЫХ

Рекомендовано Ученым советом государственного образовательного учреждения высшего профессионального образования «Оренбургского государственного университета» в качестве учебного пособия для студентов специальности 010503 – «Математическое обеспечение и администрирование информационных систем»

Оренбург 2008

УДК 004.65(075.8)
ББК 32.965я73
П50

Рецензенты
кандидат технических наук, доцент И. В, Влацкая
кандидат педагогических наук, доцент И. Н. Ващук

П50 Полищук, Ю.В.
Проектирование реляционных баз данных: учебное пособие/ Ю.В. Полищук, С.И. Сормов, Т.А. Черных - Оренбург: ГОУ ОГУ, 2008. - 132 с.

Учебное пособие представляет собой руководство по проектированию баз данных с применение бесплатных программных средств FireBird, IBExpert и IBAnalyst. Учебное пособие рассматривает вопросы проектирования систем операционной обработки данных (OLAP) с применением CASE-средства среды IBExpert. Материал разбит на главы, которые содержат краткое описание теории и примеры использования программных средств, необходимые для выполнения лабораторных работ. Для каждой лабораторной работы предлагается 16 вариантов заданий.

2402000000 ББК 32.965я73

Содержание

Введение

В настоящее время большинство предприятий и организаций используют в своей деятельности различные информационные системы. Информационные системы могут быть связаны с различными областями деятельности предприятия. В любом случае информационная система обрабатывает огромное количество информации. Основой информационной системы является база данных. Эффективность работы предприятия зависит от производительности информационной системы, которая напрямую зависит от эффективности работы базы данных. Эффективность работы базы данных в большей степени зависит от грамотно разработанного проекта базы данных, построить который помогут основы теории баз данных, рассматриваемые в настоящем пособии.

В учебном пособии рассматривается использование только бесплатно распространяемого программного обеспечения.

Примеры, представленные в пособии, были разработаны в СУБД Fire-Bird 2.0, они также являются работоспособными в совместимых СУБД таких как InterBase.

В пособии рассмотрены основные положения, необходимые для работы пользователей с базой данных.

Предлагается комплекс лабораторных работ по проектированию баз данных.

1 Проектирование реляционных баз данных

1.1 Общие положения

Проектирование базы данных (БД) – одна из наиболее сложных и ответственных задач, связанных с созданием информационной системы (ИС). В результате её решения должны быть определены содержание БД, эффективный для всех её будущих пользователей способ организации данных и инструментальные средства управления данными.

Основная цель процесса проектирования БД состоит в получении такого проекта, который удовлетворяет следующим требованиям:

– корректность модели данных (база данных должна быть гомоморфна моделируемой предметной области (ПО), где каждому объекту предметной области соответствуют данные в памяти ЭВМ, а каждому процессу – адекватные процедуры обработки данных);

– обеспечение ограничений (на объёмы внешней и оперативной памяти и другие ресурсы вычислительной системы);

– эффективность функционирования (соблюдение ограничений на время реакции системы на запрос и обновление данных);

– защита данных (от аппаратных и программных сбоев и несанкционированного доступа);

– простота и удобство эксплуатации;

– маштабируемость, т.е. возможность развития.

1.1.1 Этапы проектирования базы данных

Процесс проектирования включает в себя следующие этапы:

– инфологическое проектирование;

– определение требований к операционной обстановке, в которой будет функционировать информационная система;

– выбор системы управления базой данных (СУБД) и других инструментальных программных средств;

– логическое проектирование БД;

– физическое проектирование БД.

Инфологический подход не предоставляет формальных способов моделирования реальности, но он закладывает основы методологии проектирования баз данных.

1.2 Инфологическое проектирование

Основными задачами инфологического проектирования являются определение предметной области системы и формирование взгляда на ПО с позиций будущих пользователей БД, т.е. инфологической модели ПО.

Инфологическая модель ПО представляет собой описание структуры и динамики ПО, характера информационных потребностей пользователей в терминах, понятных пользователю и не зависимых от реализации БД. Это описание выражается в терминах не отдельных объектов ПО и связей между ними, а их типов, связанных с ними ограничений целостности и тех процессов, которые приводят к переходу предметной области из одного состояния в другое.

Рассмотрим основные подходы к созданию инфологической модели предметной области.

Функциональный подход к проектированию БД

Этот метод реализует принцип «от задач» и применяется тогда, когда известны функции некоторой группы лиц и/или комплекса задач, для обслуживания информационных потребностей которых создаётся рассматриваемая БД.

Предметный подход к проектированию БД

Предметный подход к проектированию БД применяется в тех случаях, когда у разработчиков есть чёткое представление о самой ПО и о том, какую именно информацию они хотели бы хранить в БД, а структура запросов не определена или определена не полностью. Тогда основное внимание уделя-

ется исследованию ПО и наиболее адекватному её отображению в БД с учётом самого широкого спектра информационных запросов к ней.

Проектирование с использованием метода "сущность-связь"

Метод «сущность–связь» (entity–relation, ER–method) является комбинацией двух предыдущих и обладает достоинствами обоих. Этап инфологического проектирования начинается с моделирования ПО. Проектировщик разбивает её на ряд локальных областей, каждая из которых (в идеале) включает в себя информацию, достаточную для обеспечения запросов отдельной группы будущих пользователей или решения отдельной задачи (подзадачи). Каждое локальное представление моделируется отдельно, затем они объединяются.

Выбор локального представления зависит от масштабов ПО. Обычно она разбивается на локальные области таким образом, чтобы каждая из них соответствовала отдельному внешнему приложению и содержала 6-7 сущностей.

Сущность – это объект, о котором в системе будет накапливаться информация. Сущности бывают как физически существующие (например, СОТОВЫЙ ТЕЛЕФОН), так и абстрактные (например, ФУНКЦИЯ СОТОВОГО ТЕЛЕФОНА).

Для сущностей различают тип сущности и экземпляр. Тип характеризуется именем и списком свойств, а экземпляр – конкретными значениями свойств.

Для каждой сущности выбираются свойства (атрибуты). Различают следующие типы атрибутов.

Идентифицирующие и описательные атрибуты. Идентифицирующие атрибуты имеют уникальное значение для сущностей данного типа и являются потенциальными ключами. Они позволяют однозначно распознавать экземпляры сущности. Из потенциальных ключей выбирается один первичный ключ (ПК). В качестве ПК обычно выбирается потенциальный ключ, по которому чаще происходит обращение к экземплярам записи. Кроме того, ПК должен включать в свой состав минимально необходимое для идентификации количество атрибутов. Остальные атрибуты называются описательными и заключают в себе интересующие свойства сущности.

Составные и простые атрибуты. Простой атрибут состоит из одного компонента, его значение неделимо. Составной атрибут является комбинацией нескольких компонентов, возможно, принадлежащих разным типам данных (например, ФИО или адрес). Решение о том, использовать составной атрибут или разбивать его на компоненты, зависит от характера его обработки и формата пользовательского представления этого атрибута.

Однозначные и многозначные атрибуты (могут иметь соответственно одно или много значений для каждого экземпляра сущности).

Основные и производные атрибуты. Значение основного атрибута не зависит от других атрибутов. Значение производного атрибута вычисляется

на основе значений других атрибутов (например, возраст студента вычисляется на основе даты его рождения и текущей даты).

Спецификация атрибута состоит из его названия, указания типа данных и описания ограничений целостности – множества значений (или домена), которые может принимать данный атрибут.

Далее осуществляется спецификация связей внутри локального представления. Связи могут иметь различный содержательный смысл (семантику). Различают связи типа «сущность-сущность», «сущность-атрибут» и «атрибут-атрибут» для отношений между атрибутами, которые характеризуют одну и ту же сущность или одну и ту же связь типа «сущность-сущность».

Каждая связь характеризуется именем, обязательностью, типом и степенью. Различают факультативные и обязательные связи. Если вновь порождённый объект одного типа оказывается по необходимости связанным с объектом другого типа, то между этими типами объектов существует обязательная связь (обозначается двойной линией). Иначе связь является факультативной.

По типу различают множественные связи «один к одному» (1:1), «один ко многим» (1:M) и «многие ко многим» (N:M). Пример различных типов связей, приведен на рисунке 1.1.

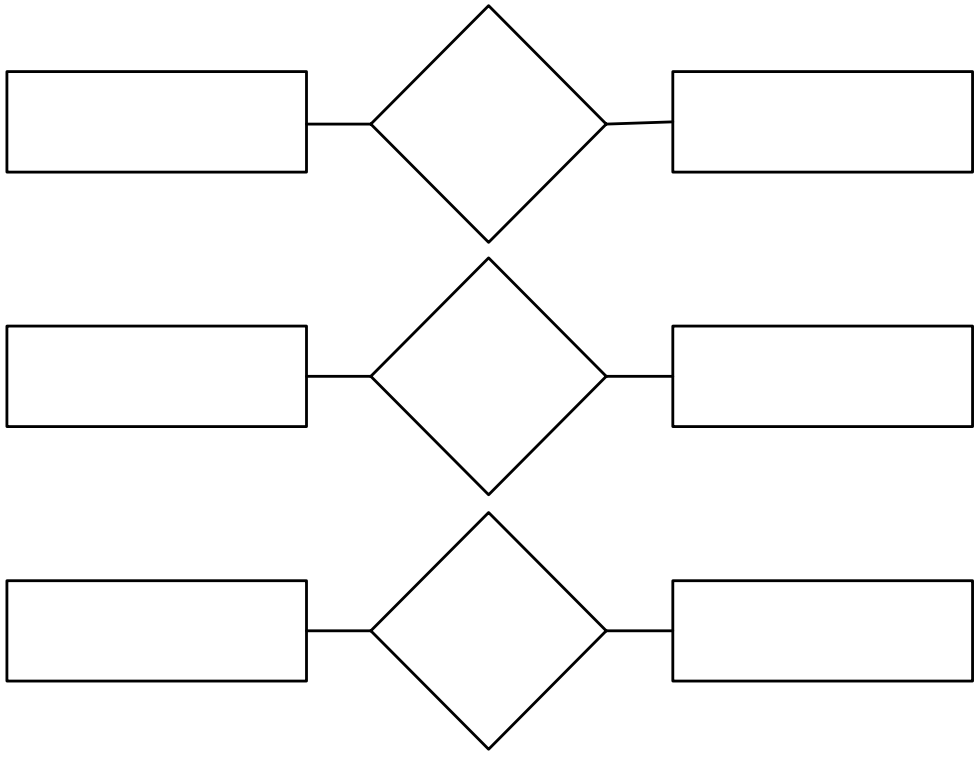

Рисунок 1.1 - Примеры типов связей

Рассмотрим рисунок подробнее. У фирмы производителя может быть только один единственный главный офис. Фирма производитель производит несколько моделей сотовых телефонов. В телефонах разных производителей

Фирма 1

могут быть реализованы одинаковые функции. Например: будильник и секундомер есть в телефонах Nokia и Samsung.

Степень связи определяется количеством сущностей, которые охвачены данной связью. Пример сущностей, их атрибутов и связей приведен на рисунке 1.2.

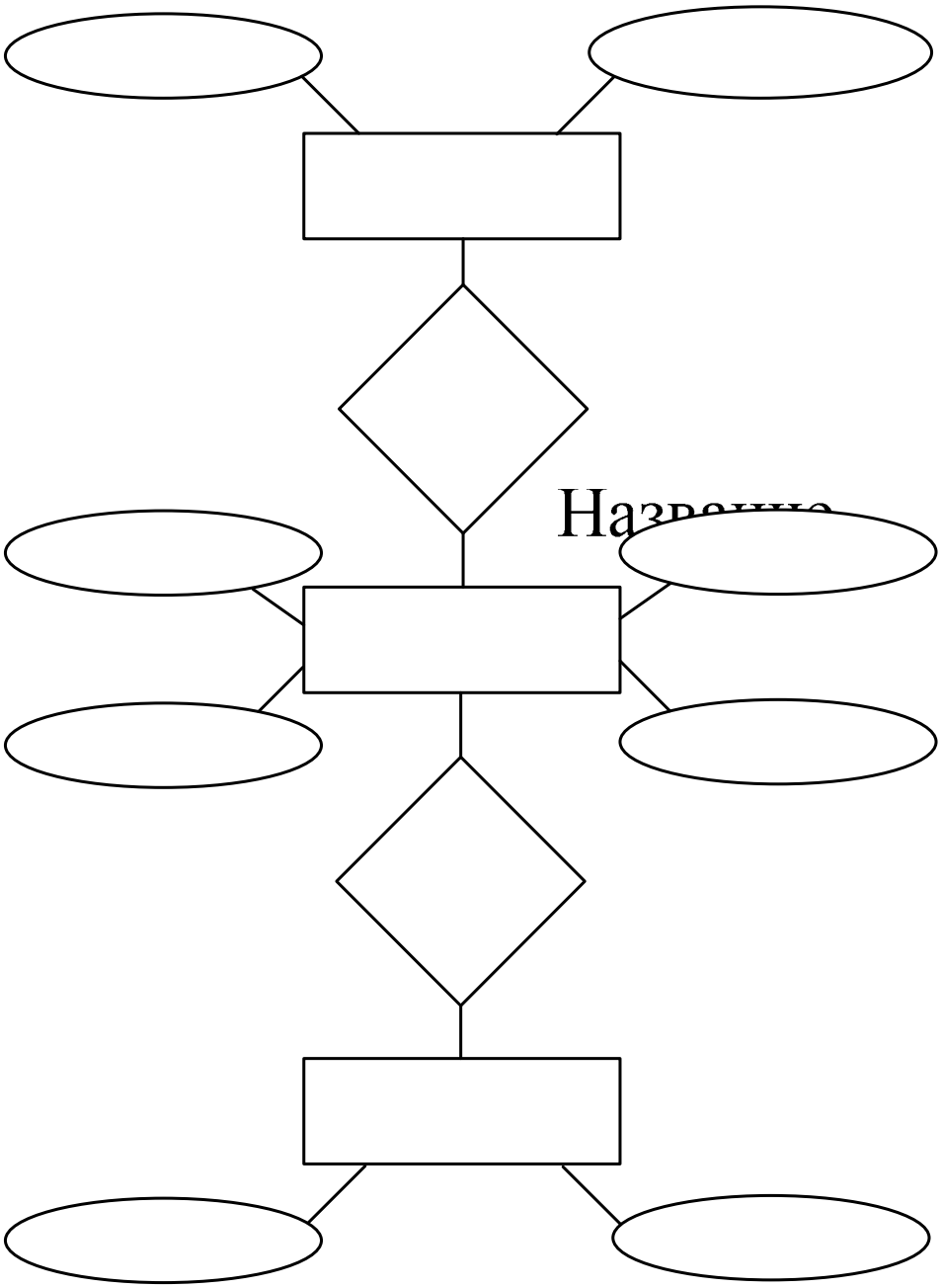

Рисунок 1.2 - Пример сущностей, их атрибутов и связей

После того, как созданы локальные представления, выполняется их объединение. При небольшом количестве локальных областей (не более пяти) они объединяются за один шаг. В противном случае обычно выполняют бинарное объединение в несколько этапов.

При объединении проектировщик может формировать конструкции, производные по отношению к тем, которые были использованы в локальных представлениях. Такой подход может преследовать следующие цели:

– объединение в единое целое фрагментарных представлений о различных свойствах одного и того же объекта;

– введение абстрактных понятий, удобных для решения задач системы, установление их связи с конкретными понятиями, использованными в модели;

– образование классов и подклассов подобных объектов (например, класс «изделие» и подклассы типов изделий, производимых на предприятии).

На этапе объединения необходимо выявить и устранить все противоречия. Например, одинаковые названия семантически различных объектов или связей или несогласованные ограничения целостности на одни и те же атрибуты в разных приложениях. Устранение противоречий вызывает необходимость возврата к этапу моделирования локальных представлений с целью внесения в них соответствующих изменений.

После объединения результаты проектирования представляют собой концептуальную инфологическую модель предметной области. Модели локальных представлений – это внешние инфологические модели.

1.2.1 Определение требований вычислительных ресурсов

На этом этапе производится оценка требований к вычислительным ресурсам, необходимым для функционирования системы, определение типа и конфигурации конкретной ЭВМ, выбор типа и версии операционной системы. Объём вычислительных ресурсов зависит от предполагаемого объёма проектируемой базы данных и от интенсивности их использования. Если БД будет работать в многопользовательском режиме, то требуется подключение её к сети и наличие соответствующей многозадачной операционной системы.

1.2.2 Выбор СУБД и других программных средств

Выбор СУБД является одним из важнейших моментов в разработке проекта БД, так как он принципиальным образом влияет на весь процесс проектирования БД и реализацию информационной системы. Теоретически при выборе СУБД нужно принимать во внимание десятки факторов. Но практически разработчики руководствуются лишь собственной интуицией и несколькими наиболее важными критериями, к которым, в частности, относятся:

– тип модели данных, которую поддерживает данная СУБД, её адекватность потребностям рассматриваемой предметной области;

– характеристики производительности системы;

– запас функциональных возможностей для дальнейшего развития ИС;

– степень оснащённости системы инструментарием для персонала администрирования данными;

– удобство и надежность СУБД в эксплуатации;
– стоимость СУБД и дополнительного программного обеспечения.

1.2.3 Логическое проектирование БД

На этапе логического проектирования разрабатывается логическая структура БД, соответствующая логической модели ПО. Решение этой задачи существенно зависит от модели данных, поддерживаемой выбранной СУБД.

Результатом выполнения этого этапа являются схемы БД концептуального и внешнего уровней архитектуры, составленные на языках определения данных (DDL, Data Definition Language), поддерживаемых данной СУБД.

1.2.4 Физическое проектирование БД

Этап физического проектирования заключается в увязке логической структуры БД и физической среды хранения с целью наиболее эффективного размещения данных, т.е. отображении логической структуры БД в структуру хранения. Решается вопрос размещения хранимых данных в пространстве памяти, выбора эффективных методов доступа к различным компонентам «физической» БД. Результаты этого этапа документируются в форме схемы хранения на языке определения данных (DDL). Принятые на этом этапе решения оказывают определяющее влияние на производительность системы.

Одной из важнейших составляющих проекта базы данных является разработка средств защиты БД. Защита данных имеет два аспекта: защита от сбоев и защита от несанкционированного доступа. Для защиты от сбоев разрабатывается стратегия резервного копирования. Для защиты от несанкционированного доступа каждому пользователю доступ к данным предоставляется только в соответствии с его правами доступа.

1.3 Особенности проектирования реляционной базы данных

Проектирование реляционной БД заключается главным образом в разработке структуры данных, т. е. в определении состава таблиц и связей между ними. При этом структура должна быть эффективной и обеспечивать:

– быстрый доступ к данным;
– отсутствие дублирования (повторения) данных;
– целостность данных.

Проектирование структуры данных (структуры БД) также называют логическим проектированием, или проектированием на логическом уровне.

При проектировании структур данных можно выделить три основных подхода:

– сбор информации об объектах решаемой задачи в рамках одной таблицы (одного отношения) и последующее разбиение ее на несколько взаимосвязанных таблиц на основе нормализации отношений;

– формулирование знаний о системе (определение типов исходных данных и их взаимосвязей) и требований к обработке данных, а затем получение с помощью средств CASE (Дизайнера БД) схемы БД или прикладной информационной системы;

– структурирование информации в результате системного анализа на основе совокупности правил и рекомендаций.

Проектирование может выполняться классическим способом, когда разработчик собирает и выделяет объекты системы и их характеристики, после чего вручную приводит их к требуемой структуре данных. Кроме того, для проектирования можно использовать так называемые CASE -системы, которые автоматизируют процесс разработки не только БД, но и информационной системы в целом.

1.4 Нормализация базы данных

Нормализация БД — это процесс уменьшения избыточности информации в БД. Метод нормализации основан на достаточно сложной теории реляционных моделей данных. Рассмотрим основные особенности и технику нормализации.

1.4.1 Избыточность данных и аномалии

При разработке структуры БД могут возникнуть проблемы, связанные:

– с избыточностью данных;

– с аномалиями.

Под избыточностью данных понимают дублирование данных, содержащихся в БД. При этом различают простое (неизбыточное) дублирование и избыточное дублирование данных.

Избыточность данных при выполнении операций с ними приводит к различным аномалиям — нарушению целостности БД. Выделим аномалии:

– удаления;

– обновления;

– ввода.

Неизбыточное дублирование является естественным и допустимым, его примером является список внутренних телефонов сотрудников организации, показанный в таблице 1.1.

Таблица 1.1 - Пример неизбыточного дублирования данных

Сотрудник	Телефон
Иванов П.Л.	123
Петров А.Ф.	123
Сидоров О.Е.	456
Кузнецова В.А.	789
Васин И.Г.	123

Три сотрудника имеют одинаковый номер телефона 123, что может быть, например, в случае, когда они находятся в одной комнате. Таким образом, номер телефона в таблице дублируется, однако для каждого сотрудника этот номер является уникальным. В случае удаления одного из дублированных значений номера телефона (удаления соответствующей строки таблицы) будет потеряна информация о фамилии сотрудника — Иванова П. Л., Петрова А. Ф. или Васина И. Г., что является аномалией удаления.

При смене номера телефона в комнате его необходимо изменить для всех сотрудников, которые в ней находятся. Если для какого-либо сотрудника этого не сделать, например, для Петрова А. Ф., то возникает несоответствие данных, связанное с аномалией обновления.

Аномалия ввода заключается в том, что при вводе в таблицу новой строки для ее полей могут быть введены недопустимые значения. Например, значение не входит в заданный диапазон или не задано значение поля, которое в обязательном порядке должно быть заполнено (не может быть пустым).

Теперь приведем пример избыточного дублирования данных. Список телефонов дополнен номерами комнат, в которых находятся сотрудники (таблица 1.2). При этом номер телефона указан только для одного из сотрудников — в примере для Иванова П.Л., который находится в списке первым. Вместо номеров телефонов других сотрудников этой комнаты поставлен прочерк. На практике кодировка прочерка зависит от особенностей конкретной таблицы, например, можно обозначать прочерк значением Null.

Таблица 1.2 - Пример избыточного дублирования данных

Сотрудник	Комната	Телефон
Иванов П.Л.	17	123
Петров А.Ф.	17	-
Сидоров О.Е.	22	456
Кузнецова В.А.	8	789
Васин И. Г.	17	-

Однако при таком построении таблицы появляются проблемы:

– номер телефона произвольного сотрудника можно получить только путем поиска в другом столбце таблицы (по номеру комнаты);

– при запоминании таблицы для каждой строки отводится одинаковая память вне зависимости от наличия или отсутствия прочерков;

– при удалении строки с данными сотрудника, для которого указан номер телефона комнаты, будет утеряна информация о номере телефона для всех сотрудников этой комнаты.

Если вместо прочерков указать номер телефона, то избыточное дублирование все равно остается. От него можно избавиться, например, с помощью разбиения таблицы на две новых (таблицы 1.3 и 1.4). Разбиение — это процесс деления таблицы на несколько таблиц с целью поддержания целостности данных, т. е. устранения избыточности данных и аномалий. Таблицы

связаны между собой по номеру комнаты. Для получения информации о номере телефона сотрудника из первой таблицы по фамилии сотрудника нужно считать номер его комнаты, после чего из второй таблицы по номеру комнаты считывается номер телефона.

Рассмотренное разбиение таблицы на две является примером нормализации отношений. При этом избыточность данных уменьшилась, однако одновременно увеличилость *время доступа* к ним. Для получения номера телефона сотрудника теперь необходимо работать с двумя таблицами, а не с одной, как в предыдущем случае.

Таблица 1.3 - Список сотрудников и номеров комнат

Сотрудник	Комната
Иванов П.Л.	17
Петров А.Ф.	17
Сидоров О.Е.	22
Кузнецова В.А.	8
Васин И.Г.	17

Таблица 1.4 - Список номеров комнат и телефонов

Комната	Телефон
17	123
8	789
22	456

1.4.2 Приведение к нормальным формам

Процесс проектирования БД с использованием метода нормальных форм является итерационным (пошаговым) и заключается в последовательном переводе по определенным правилам отношений из первой нормальной формы в нормальные формы более высокого порядка. Каждая следующая нормальная форма ограничивает определенный тип функциональных зависимостей, устраняет соответствующие аномалии при выполнении операций над отношениями БД и сохраняет свойства предшествующих нормальных форм.

Выделяют следующую последовательность нормальных форм:

- первая нормальная форма;
- вторая нормальная форма;
- третья нормальная форма;
- усиленная третья нормальная форма, или нормальная форма Бойса-Кодда;
- четвертая нормальная форма;
- пятая нормальная форма.

Проектирование начинается с определения всех объектов, информация о которых должна содержаться в БД, и выделении атрибутов (характери-

стик) этих объектов. Атрибуты всех объектов сводятся в одну таблицу, которая является исходной. Затем эта таблица последовательно приводится к нормальным формам в соответствии с их требованиями. На практике обычно используются три первых нормальных формы.

Для примера спроектируем БД для хранения информации сотовых телефонах и их функциях. Будем запоминать информацию о фирме производителе, модели телефона и функциях телефона. Объединим все данные в одну исходную таблицу, имеющую следующую структуру (поля):

- название фирмы производителя;
- информация о фирме;
- название модели сот. телефона;
- информация о модели телефона;
- функция телефона.

В качестве данных о фирме производителе сотовых телефонов будем хранить название фирмы и информацию о фирме.

Для модели сотового телефона будем хранить название модели сотового телефона, информацию о модели сотового телефона (год выпуска, вес и размеры) и функции сотового телефона.

Созданную таблицу можно рассматривать как однотабличную БД. Ее главным недостатком является значительная избыточность данных. Так, для каждой модели сотового телефона указывается его фирма производитель. Дублирование данных способно привести к аномалиям, а также к существенному увеличению размера базы.

1.4.3 Первая нормальная форма

Приведем исходную таблицу к первой нормальной форме, для которой должны выполняться следующие условия:

- поля содержат неделимую информацию;
- в таблице отсутствуют повторяющиеся группы полей.

В четвертом поле исходной таблицы содержится информация о модели телефона. Это противоречит первому требованию — информация о модели телефона является делимой. Эта информация может и должна быть разделена на три отдельных поля — год выпуска, вес и размеры.

Согласно второму требованию, в таблице должны отсутствовать повторяющиеся группы полей, т. е. группы, содержащие одинаковые функциональные значения. В исходной таблице таких групп нет, поэтому второму требованию она удовлетворяет.

Примером таблицы, имеющей повторяющиеся группы полей, может служить таблица 1.5 с результатами экзаменационной сессии.

Таблица 1.5 - Результаты экзаменационной сессии

Студент	Математика	Информатика	Физика	История	Английский язык
Семенов Р.О.	4	4	4	-	4
Костина В.К.	4	3	3	-	4
Папаев Д.Г.	5	5	5	-	-
Симонов П.С.	-	-	-	4	4

В этой таблице для оценок, полученных студентами на экзаменах, необходимо создать столько полей, сколько может быть различных предметов.

В связи с тем, что студент сдает не все экзамены (для них проставлены прочерки), размер записей и таблицы в целом неоправданно увеличивается. Кроме того, значительные трудности создает изменение состава предметов. Если организовать переименование предмета достаточно просто, то его удаление или добавление требует изменения структуры таблицы, что, вообще говоря, крайне нежелательно, поскольку структура таблицы обычно определяется еще на этапе проектирования БД.

Вторая и третья нормальные формы касаются отношений между ключевыми и неключевыми полями.

1.4.4 Вторая нормальная форма

Ко второй нормальной форме предъявляются следующие требования:

- таблица должна удовлетворять требованиям первой нормальной формы;
- любое неключевое поле должно однозначно идентифицироваться ключевыми полями.

Записи таблицы, приведенной к первой нормальной форме, не являются уникальными и содержат дублированные данные. Так, если в базе данных хранится информация о нескольких моделях сотовых телефонов одной фирмы, то таблица будет содержать одинаковые записи (названия одной фирмы). Чтобы обеспечить уникальность записей, введем в таблицу поле ключа – код (номер записи) фирмы. При этом значение ключа будет однозначно определять каждую запись таблицы.

Тогда структура таблицы примет такой вид:

- код фирмы (уникальный ключ);
- название фирмы производителя;
- информация о фирме;
- название модели сот. телефона;
- информация о модели телефона;
- функция телефона.

Записи этой таблицы имеют значительное избыточное дублирование данных, т. к. название модели сотового телефона, а также информация о

функциях указываются для каждой фирмы. Поэтому разобьем таблицу на две таблицы, одна из которых будет содержать данные о фирмах, а вторая — о моделях сотовых телефонов.

Структуры этих таблиц будут следующими.

Поля таблицы Фирма:

- код фирмы (уникальный ключ);
- название фирмы производителя;
- информация о фирме;

Поля таблицы Модели:

- код модели (уникальный ключ);
- код фирмы;
- название модели сот. телефона;
- информация о модели телефона;
- функция телефона.

Таблицы связаны по полю код фирмы, которое для таблицы Фирма имеет уникальное значение. Чтобы обеспечить уникальность записей таблицы моделей, в нее введено ключевое поле код модели.

В таблице моделей сотовых телефонов присутствует избыточное дублирование данных, т. к. функция телефона указывается для каждой модели телефона. Одна функция сотового телефона может присутствовать в нескольких моделях телефонов, поэтому выделяем функции сотового телефона в отдельную таблицу.

Структуры таблиц будет такой.

Поля таблицы Фирма:

- код фирмы (уникальный ключ);
- название фирмы производителя;
- информация о фирме;

Поля таблицы Модели:

- код модели (уникальный ключ);
- код фирмы;
- название модели сот. телефона;
- информация о модели телефона;

Поля таблицы функций:

- код функции (уникальный ключ);
- функция телефона.

Теперь для реализации отношений многие ко многим между записями таблиц Модели и таблицы Функции необходимо добавить еще одну таблицу. Назовем ее таблица Модель-функция. Для обеспечения уникальности ее записей можно добавить уникальное поле - код записи или создать составной ключ из полей код модели и код функции, так такое сочетание будет уникальным. Структура полей таблицы Модель-функция будет следующей.

Поля таблицы Модель-функция:

- код записи (уникальный ключ).
- код модели;

– код функции.

Для повышения функциональности в таблицу Модель-функция может быть добавлено поле примечание.

1.4.5 Третья нормальная форма

Требованиями третьей нормальной формы являются следующие:

– таблица должна удовлетворять требованиям второй нормальной формы;
– ни одно из неключевых полей не должно однозначно идентифицироваться значением другого неключевого поля (полей).

Приведение таблицы к третьей нормальной форме предполагает выделение в отдельную таблицу (таблицы) тех полей, которые не зависят от ключа. Все сформированные нами таблицы удовлетворяют третьей нормальной форме.

ER-диаграмма разработанной базы данных представлена на рисунке 3.12.

1.5 Вопросы для самопроверки

– Какие цели процесса проектировании базы данных Вы знаете?
– Перечислите основные этапы проектирования базы данных?
– Что подразумевается под термином «сущность»?
– Перечислите виды связей?
– Какие критерии выбора СУБД Вы знаете?
– Зачем используется язык определения данных?
– Сфера применения CASE-систем?
– Дайте определение нормализации базы данных?
– Какие основные нормальные формы Вы знаете?
– Перечислите требования первой нормальной формы?
– Перечислите требования второй нормальной формы?
– Перечислите требования третьей нормальной формы?

2 Основы работы с СУБД FireBird

2.1 Общая характеристика СУБД FireBird

Firebird ведет свое начало от исходных кодов Borland InterBase 6.0. Это программа с открытым исходным кодом. Вы можете использовать ее совершенно свободно как в коммерческих приложениях, так и в приложениях с открытым кодом (open source).

Технологии, на которых основан Firebird, используются более 20 лет, что сделало его стабильным продуктом.

Firebird - это полнофункциональная, мощная СУБД, она может обслуживать базы данных размером от нескольких килобайт до многих гигабайт, показывая хорошую производительность и практически не нуждаясь в обслуживании.

Основные характеристики Firebird:

- полная поддержка хранимых процедур и триггеров;
- транзакции, полностью совместимые с концепцией ACID;
- ссылочная целостность;
- версионная архитектура;
- небольшой размер;
- мощный внутренний язык для написания хранимых процедур и триггеров (PSQL);
- поддержка внешних пользовательских функций (UDF);
- Firebird практически не требует работы системного администратора или позволяет свести ее к минимуму;
- почти не требует настройки - использовать СУБД можно сразу же после ее установки;
- огромное интернет-сообщество пользователей и разработчиков, множество мест, где вы можете получить быструю и бесплатную помощь;
- возможность распространения встроенной в приложение (embedded) версии - замечательно подходит для создания каталогов на CD-ROM, однопользовательских и пробных версий программ;
- десятки специализированных приложений от сторонних разработчиков, включая средства администрирования, репликации, и так далее;
- безопасная запись данных (careful write) - быстрое восстановление после сбоев, отсутствие необходимости в журналировании транзакций;
- большое количество средств доступа к базе данных: native/API, драйверы dbExpress, ODBC, OLEDB, .Net provider, JDBC-драйвер, модули для Python, PHP, Perl и так далее;
- поддержка большинства распространенных операционных систем, включая Windows, Linux, Solaris, MacOS.

Сервер распространяется в трех видах: SuperServer, Classic и Embedded.

Для первого знакомства лучше подходит SuperServer как наиболее универсальное решение. Версию Classic можно рекомендовать для использования на многопроцессорных машинах, а также в некоторых других специфических случаях. Основное отличие этих версий состоит в том, что SuperServer имеет разделяемый между всеми соединениями к базе данных кэш и использует потоки для обслуживания каждого соединения, а версия Classic запускает отдельный процесс для независимой работы с каждым отдельным соединением к базе данных.

Embedded - это еще одна, удивительная версия сервера. Она включает в себя всего одну библиотеку (DLL) размером около полутора мегабайт, содержащую в себе полностью весь сервер Firebird. Это делает версию Embedded чрезвычайно удобной для распространения (так как в этом случае отсутствует необходимость установки сервера). Она идеальна для создания CDROM-каталогов, демонстрационных версий программ и приложений для однопользовательской работы.

Firebird включает в себя набор консольных программ, позволяющих создавать базы данных, исследовать их характеристики, выполнять операторы SQL и скрипты, производить резервное копирование данных, их восстановление из резервной копии и так далее.

Windows-версия сервера может быть запущена как в виде сервиса, так и в виде обычного приложения. Для управления сервером инсталлятор создает специальную иконку в «Панели Управления» операционной системы.

2.2 Обзор основных типов данных

При создании таблиц могут быть использованы следующие основные типы полей (таблица 2.1).

Столбцы могут определяться в SQL-операторах CREATE TABLE - создать таблицу БД, CREATE DOMAIN - создать домен и ALTER TABLE - изменить структуру таблицы.

Синтаксис определения столбцов:

```
<тип_данных> = {
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION} [<размерность_массива>]
| {DECIMAL | NUMERIC} [(точность [, масштаб])] <размерность_массива>]
| DATE [<размерность_массива>]
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
    [(целое)] [<размерность_массива>][CHARACTER SET набор_символов]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
    [VARYING] [(целое)] [<размерность_массива>]
| BLOB [SUB_TYPE {целое | имя_подтипа}] | [SEGMENT SIZE целое]
    [CHARACTER SET набор_символов]
| BLOB [(длина_сегмента [,подтип])]
}
```

Таблица 2.1 – Типы данных

Тип столбца	Размер, байт	Описание
SMALLINT	2	Целочисленные значения от -32768 до +32767.
INTEGER	4	Целочисленные значения от -2147483648 до +2147483647.
FLOAT	4	Вещественные числа до 7 значащих цифр в диапазоне от 3,4E-38 до 3,4E+38.
DOUBLE PRECISION	8	Вещественные числа до 15 значащих цифр в диапазоне от 1,7E-308 до 1,7E+308.
CHAR(n) или CHARACTER	0-32767	Символьный столбец длиной в **n** символов.
VARCHAR (n) или CHAR[ACTER] VARING	0-32767	Символьный столбец переменной длины, содержащий до **n** символов.
DATE	8	Дата в пределах от 01.01.0100 до 11.12.5941. Также может хранить сведения о времени.
BLOB	переменный	Любой тип двоичных данных.

Заметим, что указываемый в описании синтаксиса необязательный элемент <размерность_массива> определяет так называемый столбец-массив.

Для локальных баз данных типа Paradox и dBase колонки таблицы принято называть полями записи (fields), в то время как для серверных баз данных их обычно называют столбцами (columns). Далее будем следовать этой устоявшейся традиции.

Целочисленные значения

Столбец типа SMALLINT хранит целочисленные значения в диапазоне от -32768 до +32767.

Столбец типа INTEGER хранит целочисленные значения в диапазоне от +2 147 483 647 до -2 147 483 648.

Формат определения целочисленных столбцов:

```
<имя_столбца> = {SMALLINT | INTEGER} [<размерность_массива>]
```

Например:

```
CREATE TABLE MYTABLE (INT_COLUMN INTEGER);
```

Вещественные значения

Управлять количеством знаков в дробной части при объявлении вещественного числа нельзя. В один и тот же столбец типа FLOAT или DOUBLE PRECISION можно поместить значения и 222.3333, и 22.33. Однако в программе на значения таких полей можно наложить маску с точным количеством знаков после запятой. Например, наложение маски #,##1.00 приведет к тому, что будут показываться только два знака после запятой. Если значение столбца имеет в дробной части больше знаков, чем указано в маске, они будут округляться. Значение 100.678 в этом случае будет округлено как 100.68.

Если в дробной части значения столбца меньше знаков, чем указано в маске, недостающие разряды будут дополняться нулями. Так, значение 100.2 будет показано как 100.20.

Значения указанных типов следует применять в столбцах, где число знаков в дробной части при хранении значений несущественно и где может понадобиться большая точность представляемых вещественных значений, например, до 7 (FLOAT) или до 15 (DOUBLE PRICISION) знаков после запятой.

При попытке представления числа с большим количеством разрядов, чем это разрешено типом данных, хранимые данные округляются до последней значащей цифры в разрешенном разряде. Например, попытка записать в столбец типа FLOAT значение 123.456789 приведет к запоминанию значения 123.4568.

Формат определения целочисленных значений:

```
<тип_данных> = {FLOAT | DOUBLE PRECISION) [<размерность_массива>]
```

Например:

```
CREATE TABLE MYTABLE (FLOAT_COLUMN FLOAT);
```

Фиксировано-десятичные значения

Типы DECIMAL и NUMERIC задают вещественные значения и определяют в них фиксированное количество знаков после запятой. Формат определения этих типов

```
<тип_данных> = {DECIMAL | NUMERIC}   [(точность [,масштаб])]
[<размерность_массива>]
```

где точность определяет общее количество знаков в хранимом числе (максимум 15), а масштаб - количество знаков в дробной части и может быть равен нулю. Во всех случаях масштаб должен быть меньше точности.

Например:

```
CREATE TABLE MYTABLE (DECIMAL_COLUMN DECIMAL(9,2));
```

Специальных типов DECIMAL и NUMERIC не существует, а вместо них используются типы INTEGER или DOUBLE PRECISION: если точность (количество знаков в числе) меньше 10, то реальный тип столбца INTEGER, если больше или равно 10, реальный тип столбца DOUBLE PRECISION.

Поскольку в столбцах типа INTEGER хранить дробные значения нельзя, то независимо от того, указано количество знаков после десятичной точки или нет, объявление DECIMAL и NUMERIC с общим числом разрядов, меньшим 10, приведет к тому, что фактически в столбце будут храниться только целочисленные значения. Например, если в столбец DECIMAL_COLUMN таблицы MYTABLE поместить значение 123.456, фактически в таблице будет храниться число 123, а все цифры дробной части будут безвозвратно потеряны.

Значения типа даты

Столбцы типа DATE позволяют хранить значения даты в пределах от 01.01.0100 до 11.12.5941, а также вместе с датой - и значения времени. Формат объявления:

```
<тип_данных> = DATE [<размерность_массива>]
```

Например:

```
CREATE TABLE MYTABLE (DATE_COLUMN DATE);
```

Значения типа даты совместимы с рассматриваемыми ниже строковыми типами. Поэтому, если в SQL-операторах необходимо интерпретировать значения типа даты как строку, нет необходимости в приведении типов. Например, можно так записать в символьный столбец S1 значение типа даты (символы | | означают операцию сцепления строк, функция NOW возвращает текущие дату и время):

```
UPDATE SOMETABLE
(SET S1= "Дата отгрузки " || NOW);
```

Кроме того, присвоение полям значения типа дата также производится в символьном формате:

```
INSERT INTO SOMETABLE (DATA_PRIHODA, KOLVO)
VALUES("01-FEB-2007", 100);
```

Символьные типы данных

Столбцы типа CHAR(n) и VARCHAR(n) позволяют хранить строковые значения длиной до n символов. В любом случае n не может превышать 32 Кб.

Как указано в документации, CHAR(n) определяет строковые столбцы фиксированной длины. Определение «фиксированная длина» означает, что даже если в столбец записано меньше n символов, незаполненное пространство заполняется пробелами. При хранении хвостовые пробелы усекаются, что позволяет хранить действительно значащее содержимое столбцов типа CHAR(n) и сближает их со строками переменной длины. При чтении значения столбца хвостовые пробелы вставляются в столбец снова. Такой механизм придуман для экономии дискового пространства, когда действительная длина значений данного столбца в различных записях варьируется достаточно широко.

VARCHAR(n) определяет строковые столбцы переменной длины. Определение «переменная длина» означает, что в записи хранится ровно столько символов, сколько их имеется в значении столбца. Например, если для VARCHAR(100) значения столбцов для одной записи - «Петров», а для другой - «Барабанов», в первом случае хранится 6 символов, а во втором - 9.

Столбцы VARCHAR(n) позволяют экономить дисковое пространство, давая возможность серверу располагать больше записей на странице БД. К недостаткам относится то, что VARCHAR(n) читаются медленнее, чем CHAR(n).

Попытка записать в столбец более чем n символов приведет к усечению лишних символов.

Формат определения символьных столбцов:

```
<тип_данных> = {CHAR | CHARACTER | CHARACTER VARYING |
                VARCHAR} [(n)] [<размерность_массива>]
                [CHARACTER SET набор_символов]
```

Здесь n определяет длину символьного столбца. Если n опущено, по умолчанию подразумевается 1.

CHARACTER SET позволяет установить кодировку путем указания имени используемого набора символов. Если CHARACTER SET не указан, используется кодировка, принятая по умолчанию для БД (устанавливается в предложение DEFAULT CHARACTER SET оператора CREATE DATABASE.

Если определен набор символов, то в столбец невозможно ввести символы, не входящие в данную таблицу кодировки.

Для представления столбцов, которые будут содержать русскоязычные тексты, рекомендуем использовать набор символов WIN1251, например:

```
CREATE TABLE MYTABLE (SYMPOLE CHARACTER(20) CHARACTER SET WIN1251);
```

Если при создании БД и описании столбца набор символов не указан, принимается по умолчанию набор NONE, который позволяет вводить в символьные столбцы символы любых кодировок. Данные хранятся так, как они вводятся. Однако для столбцов с набором символов NONE могут возникнуть проблемы при записи их значения в столбцы, которым набор символов явно назначен.

Порядок сортировки символов определяет принцип, по которому символьные значения будут сравниваться и сортироваться в операторах SELECT (если в нем присутствуют разделы WHERE, ORDER BY, при обновлении индексов и т. д.) Он может явно указываться предложением COLLATE <collation order> при объявлении символьного поля. Для кириллического набора символов WIN 1251 рекомендуется использовать порядок сортировки PXW_CYRL.

Например:

```
CREATE TABLE MYTABLE (SYM_COL CHARACTER(20) CHARACTER SET WIN1251 COLLATE PXW_CYRL);
```

Следует учесть, что значения в строковых столбцах с набором WIN1251 и сортировкой PXW_CYRL упорядочиваются в стиле Windows*:* буквы идут по парам, строчные буквы предшествуют заглавным: аАбБвВгГ…Яя

С другой стороны, если набор символов и сортировка не указаны, столбцы упорядочиваются в стиле MS-DOS*:*
АБВГ...Яабвг...я

Значения типа BLOB

В столбцах типа BLOB хранят данные, которые невозможно хранить в столбцах других типов. BLOB имеет переменную длину и интерпретируется как последовательность байтов. В BLOB-столбцах можно хранить графические образы, тексты большой длины, звуковые файлы, видеоизображения, мультимедиа-информацию. В них можно также хранить указатели на неструктурированные файлы информации, которые расположены в ином, фиксированном месте. При этом интерпретация содержимого BLOB-столбца может ложиться не только на клиентскую часть, но также и на сервер. В последнем случае в серверной БД могут быть определены так называемые BLOB-фильтры, определяющие способы преобразования BLOB-значений от одного вида к другому. Например, преобразование изображений из одного графического формата в другой.

FireBird хранит значения BLOB-столбцов в самой БД в виде сегментов. Одна операция ввода-вывода при доступе к BLOB-информации оперирует с одним сегментом. В таблице БД, если в ней объявлен столбец типа BLOB, хранится указатель на начальный сегмент столбца в области хранения BLOB-информации этой БД.

Длина сегмента BLOB не влияет на производительность. Однако исходя из некоторых соображений ее можно определять явно при создании BLOB-столбца:

```
BLOBPOLE BLOB SEGMENT SIZE 1024;
```

По умолчанию длина сегмента составляет 80 байт, максимальная длина сегмента 32 Кбайт (32 768 байт).

Для BLOB может быть указан подтип, определяющий, какой тип данных будет записан в этот столбец. Подтип есть число. Положительные значения зарезервированы: 0 (по умолчанию) для указания двоичных данных и 1 для указания ASCII-текстов. Отрицательные значения могут использоваться программистами для определения собственных подтипов. В общем случае указание подтипа производится через опцию SUB_TYPE:

```
<имя_столбца> BLOB SUB_TYPE -1 CHARACTER SET WIN1251;
```

Однако автоматически не отслеживается правильность занесения информации в столбцы типа BLOB. Это необходимо делать в приложениях.

Столбцы-массивы

В FireBird могут определяться столбцы-массивы. Такие столбцы вместо единичного значения содержат массив значений одинакового типа, доступ к каждому из которых возможен с помощью индекса. Базовыми типами для элементов массива могут быть любые из рассмотренных типов кроме BLOB.

Для создания столбца-массива в конце его определения указывается в квадратных скобках целое число, определяющее количество элементов массива.

Например:

```
CREATE TABLE MYTABLE (ARR_COL INTEGER[30]);
```

В этом примере столбец ARR_COL будет содержать по 30 целочисленных значений в каждой записи таблицы MYTABLE. По умолчанию нумерация элементов начинается с 1, однако программист может явно указать границы индексов:

```
CREATE TABLE MYTABLE (ARR_COL INTEGER [0:29]);
```

Массив может содержать до 16 измерений, которые перечисляются через запятые:

```
CREATE TABLE MYTABLE (ARR_COL INTEGER [0:29, 15, 10]);
```

Совместимость типов

При выполнении операций над столбцами разного типа FireBird пытается автоматически «привести» типы таким образом, чтобы значения, участвующие в операции, принадлежали совместимым типам. Кроме того, для явного приведения типов можно использовать функцию CAST, которая приводит типы внутри оператора SELECT, обычно в предложении WHERE:

```
CAST (<значение> | NULL AS ТипДанных)
```

Совместимыми считаются только типы DATE, CHAR и NUMERIC. Вот как можно привести значение DATE1 (тип DATE) к типу CHAR и сравнить его со значением CHAR_DATE (типа CHAR):

```
SELECT ...
WHERE CHAR_DATE <= CAST(DATE1 AS CHAR);
```

2.3 Создание доменов

Понятие домена

Если в таблице БД или в нескольких таблицах БД присутствуют столбцы, обладающие одними и теми же характеристиками, можно предварительно описать тип такого столбца и его поведение с помощью домена, а затем поставить в соответствие каждому из одинаковых столбцов имя домена.
Например, создать домен POL_TYPE и затем использовать его при создании таблицы SOTR как тип столбца POL*:*

```
CREATE DOMAIN POL_TYPE AS CHAR(3) COLLATE PXW_CYRL;

CREATE TABLE SOTR(
FIO CHAR(20) NOT NULL,
POL POL_TYPE,
OTDEL CHAR(10),
DOLJ CHAR(20),
PRIMARY KEY(FIO));
```

Домен определяется оператором CREATE DOMAIN, имеющим такой формат:

```
CREATE DOMAIN домен [AS] <тип_данных>

[DEFAULT {литерал | NULL | USER}]
[NOT NULL] [CHECK (<усл_поиска_домена>)]
[COLLATE collation];
```

Предложение COLLATE задает порядок сортировки символов, например:

```
CREATE DOMAIN POL_TYPE AS CHAR(3) COLLATE PXW_CYRL;
```

Предложение DEFAULT определяет выражение, которое по умолчанию заносится в колонку, ассоциированную с доменом, при создании записи таблицы. Это значение будет присутствовать в соответствующем столбце записи до тех пор, пока пользователь не изменит его каким-либо образом. Зна-

чения по умолчанию могут быть выражены как DEFAULT-значение (числовое, строковое или дата), NULL-специфицирует пустое значение или USER - имя текущего пользователя.

Значения по умолчанию, присваиваемые столбцу и объявленные в операторах CREATE TABLE или ALTER TABLE, переопределяют значения, присваиваемые по умолчанию тому же столбцу согласно директивам, содержащимся в CREATE DOMAIN.

Предложение NOT NULL указывает, что столбцы, ассоциированные с доменом, обязательно должны содержать какое-либо значение, отличное от пустого. Необходимо следить за тем, чтобы в объявлении домена не было противоречий, например, в описании значения по умолчанию как NULL к объявлении директивы NOT NULL.

Ограничения на значения столбцов, ассоциированных с доменом

Предложение CHECK определяет требования к значениям каждого столбца, ассоциированного с доменом. Столбцу не могут быть присвоены значения, не удовлетворяющие ограничениям, наложенным в предложении CHECK. Формат ограничения, накладываемого на значения полей, ассоциированных с доменом:

```
<огранич_домена> = {
VALUE <оператор> <значение>
| VALUE [NOT] BETWEEN <значение1> AND <значение2>
| VALUE [NOT] LIKE <значение> [ESCAPE <значение>]
| VALUE [NOT] IN (<значение1> [, <значение2> ...])
| VALUE IS [NOT] NULL
| VALUE [NOT] CONTAINING <значение>
| VALUE [NOT] STARTING [WITH] <значение>
| (<огранич_домена>)
| NOT <огранич_домена>
| <огранич_домена> OR <огранич_домена>
|<огранич_домена> AND <огранич_домена>
}
```

где <оператор> = {=|<|>|<=|>=|!<|!>|<>|!=}.

Ключевое слово VALUE далее означает все правильные значения, которые могут быть присвоены столбцу, ассоциированному с доменом.

– VALUE <оператор> <значение> - значение домена находится с параметром значение во взаимоотношениях, определяемых параметром оператор. Например, значение, которое может быть записано в столбец, ассоциированный с доменом ID_TYPE, должно быть больше или равно 100: CREATE DOMAIN ID_TYPE AS INTEGER CHECK(VALUE >= 100);

– BETWEEN <значение1> AND <значение2> - значение домена должно находиться в промежутке между значение1 и значение2, включая их.

– LIKE <значение1> [ESCAPE <значение2>] - значение домена должно «походить » на параметр значение1. При этом символ «%» употребляется для указания любого значения любой длины и символ подчеркивания «_» - для указания любого единичного символа. Например, LIKE "%USD" - вводимое значение должно оканчиваться символами «USD» независимо от того, какие символы и сколько расположены перед ними; LIKE "_94" - вводимое значение может содержать 4 символа, из которых первые два - любые и последние два - «94». ESCAPE <значение2> используется, если в операторе LIKE служебные символы «%» или «_» должны использоваться в шаблоне подобия. В этом случае выбирается некоторый символ, например «!», после которого служебные символы теряют свой специальный статус и входят в поисковую строку как обычные символы. Символ «!» указывается после слова ESCAPE, например (значения столбца SUMMA должны заканчиваться символом «%»):

```
CREATE DOMAIN SUMMA AS
CHAR(10) CHECK(LIKE "%!%" ESCAPE "!");
```

– IN (<значение1> [, <значение2> ...]) - значение домена должно совпадать с одним из приведенных в списке параметров значением, например: CREATE DOMAIN POLJTYPE AS CHAR(3) CHECK (VALUE IN ("Муж","Жен"));

– CONTAINING <значение> - значение домена должно содержать вхождение параметра значение, неважно в каком месте. Например, в наименовании отдела вхождение «041» может встретиться где угодно («Отдел-041002», «003404192», и т.д.):

```
CREATE DOMAIN OTDEL_TYPE AS VARCHAR(10) CHECK(VALUE
CONTAINING "041") COLLATE PXW_CYRL;
```

– STARTING [WITH] <значение> - значение домена должно начинаться параметром значение. Например, название отдела должно начинаться с «041»: CREATE DOMAIN OTDEL_TYPE AS VARCHAR(10) CHECK(VALUE STARTING WITH "041") COLLATE PXW_CYRL; Может быть задана комбинация условий, которым должно соответствовать значение домена. В этом случае отдельные условия соединяются операторами AND или OR. Например:

```
CREATE DOMAIN OTDEL_TYPE AS
VARCHAR(10) CHECK(VALUE STARTING WITH "041" AND VALUE
CONTAINING "-12") COLLATE PXW CYRL;
```

Для большинства условий можно указать слово NOT, которое изменит условие с точностью до наоборот. Например: CHECK(VALUE NOT BETWEEN 1 AND 100).

Изменение определения домена

Оператор имеет следующий формат:

```
ALTER DOMAIN имя {
```

```
[SET DEFAULT {литерал | NULL | USER}]
| [DROP DEFAULT]
     | [ADD [CONSTRAINT] CHECK (<огранич_домена>)
     | [DROP CONSTRAINT]
};
```

позволяет изменить параметры домена, определенного ранее оператором CREATE DOMAIN. Однако нельзя изменить тип данных и определение NOT NULL. Следует помнить, что все сделанные изменения будут учтены для всех столбцов, определенных с использованием данного домена (в том случае, если параметры домена не были переопределены при создании столбцов таблицы или впоследствии).

– SET DEFAULT устанавливает значения по умолчанию подобно тому, как это делается в операторе CREATE DOMAIN.

– DROP DEFAULT отменяет текущие значения по умолчанию, назначенные домену.

– [ADD [CONSTRAINT} CHECK (<огрант_домена>)] добавляет условия, которым должны соответствовать значения столбца, ассоциированного с доменом. При этом возможно определение условий, рассмотренных выше для предложения CHECK оператора CREATE DOMAIN.

– DROP CONSTRAINT удаляет условия, определенные для домена в предложении CHECK оператора CREATE DOMAIN или предыдущих операторов ALTER DOMAIN. Например, пусть определен домен ID_TYPE:

```
CREATE DOMAIN ID_TYPE AS
INTEGER CHECK(VALUE >= 100);
```

и в дальнейшем он использован при создании таблицы AAA:

```
CREATE TABLE AAA(
ID ID_TYPE NOT NULL,
FIO VARCHAR(20),
PRIMARY KEY(ID));
```

Изменить условие CHECK так, чтобы значение было больше или равно 100 и меньше или равно 500, можно за два шага. Сначала нужно удалить старое условие:

```
ALTER DOMAIN IDJTYPE DROP CONSTRAINT;
```

после чего добавить новое (которое на самом деле есть модифицированное старое):

```
ALTER DOMAIN IDJTYPE
CHECK (VALUE >= 100 AND VALUE <= 500);
```

Заметим, что изменять определение таблицы AAA нет необходимости, и отныне в столбец ID этой таблицы можно занести значения, большие или равные 100 и меньшие или равные 500.

2.4 Создание таблиц

Общий формат оператора CREATE TABLE

Перед созданием таблиц БД необходимо продумать определение всех столбцов таблицы и характеристик каждого столбца (таких как тип, длина, обязательность для ввода, ограничения, накладываемые на значения и пр.), индексов, ограничений целостности по отношению к другим таблицам. Если при определении столбцов используются домены, эти домены должны быть предварительно созданы оператором CREATE DOMAIN. Та БД, в которую будет добавлена создаваемая таблица, должна быть открыта, т. е. с ней должно быть установлено активное соединение.

Создание таблицы БД осуществляется оператором

```
CREATE TABLE ИмяТаблицы [EXTERNAL [FILE] "<имя файла>"]
(<опр_столбца> [, <опр_столбца> | <ограничение> ...]);
```

[EXTERNAL [FILE] "<имя файла>"] относится к внешним, т. е. расположенным отдельно от БД, таблицам БД.
<опр_столбца> - определение столбца БД.
Определение столбца имеет формат:

```
<опр_столбца> = опр_столбца {тип данных COMPUTED
[BY] (< выражение>) | domain}
[DEFAULT {литерал | NULL | USER}]
[NOT NULL] [ <огранич_столбца>]
[COLLATE collation]
```

– опр_столбца - имя столбца;

– тип_данных - тип столбца и, возможно, размерность массива, если столбец - массив; для символьных столбцов может быть указан набор символов, отличный от принятого по умолчанию, при помощи предложения CHARACTER SET;

– COMPUTED [BY] (<выражение>) - служит для определения столбца вычисляемых значений (подробнее см. ниже);

– domain - имя домена, т. е. ранее описанного типа столбца;

– DEFAULT определяет значение, которое по умолчанию заносится в столбец, ассоциированный с доменом, при создании записи таблицы; это значение будет присутствовать в соответствующем столбце данной записи до тех пор, пока пользователь не изменит его каким-либо образом; значения по умолчанию;

– огранич_столбца - ограничения, накладываемые на значения столбца (подробно рассматриваются ниже);

– COLLATE collation - определяет порядок сортировки символов (для символьных столбцов).

Столбцы вычисляемых значений

Столбцы вычисляемых значений описываются с помощью предложения COMPUTED [BY] (<выражение>).

Значение таких столбцов не вводится пользователем, а вычисляется автоматически согласно выражению. Тип результирующего значения и будет служить типом вычисляемого столбца. Например, таблица MOBIL_PACK содержит массу одного телефона WEIGHT и массу упаковки сотовых телефонов (10 телефонов) WEIGHT_10. Она создана таким образом:

```
CREATE TABLE MOBIL_PACK (
N INTEGER NOT NULL,
WEIGHT INTEGER,
WEIGHT_10 COMPUTED BY (WEIGHT*10),
PRIMARY KEY (N));
```

Ограничения целостности

Ограничения целостности бывают двух уровней: ограничения, накладываемые на отдельный столбец и на всю таблицу.

В случае наложения ограничений целостности на отдельный столбец описание ограничения приводится следом за именем столбца и его типом:
N INTEGER NOT NULL PRIMARY KEY, ...

В данном случае столбец TOVAR составляет первичный ключ, что объявлено в предложении PRIMARY KEY.

В случае наложения ограничений на таблицу описание ограничений указывается после объявлений отдельных столбцов, например:

```
CREATE TABLE ...(
N INTEGER NOT NULL,
…,
PRIMARY KEY (N));
```

Первичный ключ

Если по столбцу строится первичный ключ, столбцу может быть приписан атрибут PRIMARY KEY:

```
CREATE TABLE MOBIL_PACK (
N INTEGER NOT NULL PRIMARY KEY,
…);
```

Первичный ключ может быть построен и путем включения имени (имен) ключевого столбца (столбцов) в качестве параметров отдельного предложения PRIMARY KEY:

```
CREATE TABLE MOBIL_PACK (
N INTEGER NOT NULL,
PRIMARY KEY (N));
```

Однако и в том, и в другом случае ключевой столбец (столбцы), входящий (входящие) в состав первичного ключа, должен быть помечен как NOT NULL, поскольку первичный ключ не может быть построен даже по одному пустому значению.

Первичный ключ, если он служит для обеспечения ссылочной целостности, должен корреспондировать с внешним ключом (FOREIGN KEY) другой (дочерней) таблицы.

Уникальный ключ

Атрибут UNIQUE, если он приписан столбцу, означает, что в столбце не могут содержаться два одинаковых значения. Уникальный ключ строится по столбцу (столбцам), когда столбец не входит в состав первичного ключа, но тем не менее его значение должно всегда быть уникальным. Например, для таблицы MOBIL («склад сотовых телефонов») первичный ключ строится по идентификатору записи ID, введенному для сокращения объема первичного ключа и времени поиска по нему (объем ключа по столбцу типа INTEGER много меньше объема ключа по символьному полю с максимальной длиной в 50 символов). Однако и IMAY-код сотового телефона уникален, по этому ему приписан атрибут UNIQUE:

```
CREATE TABLE MOBIL (
ID INTEGER NOT NULL PRIMARY KEY,
IMAY VARCHAR(50) NOT NULL UNIQUE);
```

Уникальность может быть приписана и на уровне таблицы:

```
CREATE TABLE MOBIL (
ID INTEGER NOT NULL PRIMARY KEY,
IMAY VARCHAR(50) NOT NULL
UNIQUE (IMAY));
```

Столбец, объявленный с атрибутом UNIQUE, как и первичный ключ, может применяться для обеспечения ссылочной целостности между родительской и дочерней таблицами. В этом случае столбец с атрибутом UNIQUE должен принадлежать к родительской таблице и должен корреспондировать с внешним ключом (FOREIGN KEY) другой (дочерней) таблицы.

Внешний ключ и определение ссылочной целостности

Внешний ключ строится в дочерней (детальной) таблице для соединения родительской (главной) и дочерних таблиц БД. Формат определения:

```
FOREIGN KEY (<список столбцов внешнего ключа>)
REFERENCES <имя родительской таблицы>
[<список столбцов родительской таблицы>]
[ON DELETE (NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
[ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
```

Список столбцов внешнего ключа определяет столбцы дочерней таблицы, по которым строится внешний ключ.

Имя родительской таблицы определяет таблицу, в которой описан первичный ключ (или столбец с атрибутом UNIQUE). На этот ключ (столбец) должен ссылаться внешний ключ дочерней таблицы для обеспечения ссылочной целостности.

Список столбцов родительской таблицы необязателен при ссылке на первичный ключ родительской таблицы. При ссылке в родительской таблице на столбец с атрибутом UNIQUE этот список лучше привести.

Параметры ON DELETE, ON UPDATE определяют способы изменения подчиненных записей дочерней таблицы при удалении или изменении поля связи в записи родительской таблицы. Перечислим эти способы:

- no action - запрет удаления/изменения родительской записи при наличии подчиненных записей в дочерней таблице;
- cascade - для оператора ON DELETE*:* при удалении записи родительской таблицы происходит удаление подчиненных записей в дочерней таблице; для ON UPDATE: при изменении поля связи в записи родительской таблицы происходит изменение на то же значение поля внешнего ключа у всех подчиненных записей в дочерней таблице;
- set default - в поле внешнего ключа у записей дочерней таблицы заносится значение этого поля по умолчанию, указанное при определении поля (параметр DEFAULT); если это значение отсутствует в первичном ключе, срабатывает исключение; заметим, что используется значение по умолчанию, имевшее месте на момент определения ссылочной целостности; если впоследствии это значение будет изменено, ссылочная целостность при SET DEFAULT все равно будет использовать прежнее значение;
- set null - в поле внешнего ключа у записей дочерней таблицы заносится значение NULL.

Пример.

Определим две таблицы:

- родительскую «Фирма производитель сотовых телефонов» FIRMA с полями N_FIRMA (номер фирмы), NAME (название фирмы) ,ABOUT (информация о фирме); первичный ключ по полю N_FIRMA;
- дочернюю «Модели сотовых телефонов» MODEL с полями N_MODEL (номер модели), N_FIRMA (фирма), MODEL (модель). Первичный ключ по полю N_MODEL, внешний - по полю N_FIRMA для обеспечения ссылочной целостности с таблицей FIRMA.

Для определения этих таблиц необходимо выполнить такие операторы:

```
CREATE TABLE FIRMA (
N_FIRMA INTEGER NOT NULL,
NAME VARCHAR(50),
ABOUT VARCHAR(300),
PRIMARY KEY(N_FIRMA));

CREATE TABLE MODEL (
N_MODEL INTEGER NOT NULL PRIMARY KEY,
N_FIRMA INTEGER NOT NULL,
MODEL VARCHAR(50),
FOREIGN KEY(N_FIRMA) REFERENCES FIRMA(N_FIRMA));
```

Теперь для таблицы FIRMA будет установлена блокировка удаления или изменения значения в столбце N_FIRMA, если в таблице MODEL имеются записи о моделях сотовых телефонов удаляемой фирмы.
Определение общих полей родительской и дочерней таблиц (полей связи) должно в точности совпадать.

Именование ссылочной целостности

Ссылочная целостность может именоваться следующим образом:

```
[CONSTRAINT <имя  ссылочной целостности>]
FOREIGN KEY   (<список столбцов внешнего ключа>)
REFERENCES <имя родительской таблицы> [<список столбцов родитель-
ской таблицы>]
```

Необязательное имя ссылочной целостности присутствует в системных сообщениях относительно нарушения целостности, а также может использоваться при анализе структуры БД и изменении структуры таблиц. Если это имя опущено, СУБД установит его сама.

Например, назначим в приведенной выше таблице MODEL имя ссылочной целостности:

```
CREATE TABLE MODEL (
N_MODEL INTEGER NOT NULL PRIMARY KEY,
N_FIRMA INTEGER NOT NULL,
MODEL VARCHAR(50),
CONSTRAINT MODEL_FIRMA FOREIGN KEY(N_FIRMA) REFERENCES
FIRMA(N_FIRMA));
```

Имя ссылочной целостности может назначаться также при определении первичного ключа и уникального набора атрибутов:

Требования к значениям столбцов

Требования к значениям отдельных столбцов могут определяться как на уровне конкретного столбца, так и на уровне таблицы. Например, для таблицы MODEL (модели сотовых телефонов) масса телефона (WEIGHT) должен быть больше 50 грамм. Тогда данное ограничение на уровне столбца определяется следующим образом:

```
CREATE TABLE MODEL (
N INTEGER NOT NULL PRIMARY KEY,
WEIGHT SMALLINT NOT NULL CHECK (WEIGHT > 50),
MODEL VARCHAR(50));
```

а на уровне таблицы - так:

```
CREATE TABLE MODEL (
N INTEGER NOT NULL,
WEIGHT SMALLINT NOT NULL,
MODEL VARCHAR(50),
PRIMARY KEY(N),
CHECK (WEIGHT > 50));
```

Ограничения, накладываемые на столбцы таблицы, определяются при помощи предложения CHECK, общий формат которого приводится ниже.

```
CHECK (<условия__лоиска>)

<условия_поиска> =
     {<значение> <оператор> { <значение1> | (<выбор_одного>)}
     | <значение> [NOT] BETWEEN <значение1> AND <значение2>
     | <значение> [NOT] LIKE <значение> [ESCAPE <значение>]
     | <значение> [NOT] IN ( <значение1> [ , <значение2> ...] |
<условия_поиска>)
     | <значение> IS [NOT] NULL
     | <значение> {[NOT] {= | < | >} | >= | <=}
          { ALL | SOME | ANY} (<выбор_многих>)
     | EXISTS (<выражение_выбора>)
     | SINGULAR (<выражение_выбора>)
     | <значение> [NOT] CONTAINING <значение1>
     | <значение> [NOT] STARTING [WITH] <значение1>
     | (<условия_поиска>)
     | NOT <условия_поиска>
     | <условия_поиска> OR <условия_поиска>
     | <условия_поиска> AND <условия_поиска>}

<значение> = {столбец | <константа> | <выражение> | <функция>
     | NULL | USER | RDB$DB_KEY
     } [COLLATE collation]
```

```
<константа> = число | “строка”

<функция> = {
COUNT (*| [ALL] <значение> | DISTINCT <значение>)
      | SUM ([ALL] <значение> | DISTINCT <значение>)
      | AVG ([ALL] <значение> | DISTINCT <значение>)
      | MAX ([ALL] <значение> | DISTINCT <значение>)
      | MIN ([ALL] <значение> | DISTINCT <значение>)
      | CAST (<значение> AS <тип_данных>)
      | UPPER (<значение>)
      | GEN_ID (генератор, <значение>)
}
```

<onepamop> = {=|<|>|<=|>=|!<|!>|<>|!=}.
<выбор_одного> = оператор SELECT, возвращающий одно значение или не возвращающий ничего.
<выбор_многих> = оператор SELECT, который может возвращать более одного значения (список значений) или ни одного.
<выражение_выбора> = оператор SELECT, который может возвращать более одного значения (список значений) или ни одного.
<значение> <оператор> <значение!> определяет, что значение столбца находится со значением! во взаимоотношениях, определяемых оператором, который может принимать одно из следующих значений:
<оператор> = {=|<|>|<=|>=|!<|!>|<>|!=}.

Например, значение столбца STOLBEZ не должно быть меньше 100:

```
CREATE TABLE TBL(
CHECK(STOLBEZ >= 100);
```

<значение> <оператор> (<выбор_одного>) устанавливает, что значение столбца находится во взаимоотношениях, определяемых оператором оператор, с результатом выполнения запроса SELECT к одной или нескольким таблицам БД, причем в качестве результата выполнения запроса возвращается единичное значение (выбор_одного).
<значение> BETWEEN <значение1> AND <значение2> - значение столбца должно находиться в диапазоне от значение1 до значение2.
LIKE <значение> [ESCAPE <значение>] - содержимое столбца должно «походить » на значение.
<значение> { ALL | SOME | ANY} (<выбор_многих >)- значение столбца больше, меньше и т. д. всех (ALL) или некоторых (SOME или ANY) значений в списке выбор_многих. Список значений выбор_многих выдается как результат выполнения оператора SELECT по отношению к одной или нескольким таблицам БД.
EXISTS (<выражение_выбора>) - возвращает True, если список выражение_выбора не пустой, т. е. содержит хотя бы одну строку. Список выражение_выбора выдается как результат выполнения оператора SELECT по отношению к одной или нескольким таблицам БД.

SINGULAR (<выражение_выбора>) - возвращает True, если список выражение_выбора содержит только одну строку. Список выражение_выбора выдается как результат выполнения оператора SELECT по отношению к одной или нескольким таблицам БД.
<значение> CONTAINING <значение!> - значение столбца должно содержать вхождение значение1, не важно, в каком месте.
<значение> STARTING [WITH] <значение!> определяет, что значение столбца должно начинаться с символов значение!.

В качестве <значение> можно указывать имя столбца, константу, выражение, функцию. Могут использоваться следующие стандартные функции: COUNT - счетчик повторения, SUM - сумма, AVG - среднее значение, MAX - максимальное значение, MIN - минимальное значение, CAST - приведение типов, UPPER - приведение всех букв к заглавным, GEN_ID - возвращает уникальное значение генератора.

Изменение объявления таблицы

Оператор ALTER TABLE позволяет:
- добавить определение нового столбца;
- удалить столбец из таблицы;
- удалить атрибуты целостности таблицы или отдельного столбца;
- добавить новые атрибуты целостности.

Перед изменением каких-либо атрибутов столбца данные, которые хранятся в нем, нужно сохранить. Для этого в таблице определяют временный столбец, в точности повторяющий все характеристики того столбца, который планируется изменить. Затем данные из изменяемого столбца копируют во временный столбец (используя, например, оператор UPDATE). После этого столбец, подлежащий изменению, попросту удаляют из таблицы, а на его месте создают новый, одноименный столбец с желаемыми атрибутами. В заключение в него копируют данные из временного столбца, а временный столбец уничтожают.

Пример. Пусть имеется таблица:

```
CREATE TABLE SOTR (
ID_SOTR INTEGER NOT NULL PRIMARY KEY,
FIO CHAR (10) COLLATE PXW_CYRL,
OTDEL VARCHAR (10) COLLATE PXW_CYRL,
DOLJNOST CHAR (10) COLLATE PXW CYRL);
```

Пусть необходимо изменить характеристики столбца FIO, изменив тип столбца с CHAR(10) на VARCHAR(25).
1. Добавляем в таблицу новый временный столбец FIO_TMP, полностью повторяющий характеристики изменяемого столбца FIO:

```
ALTER TABLE SOTR
ADD FIOJTMP CHAR(10) CHARACTER SET WIN1251 COLLATE PXW_CYRL;
```

2. Копируем данные из FIO в FIO_TMP:

```
UPDATE SOTR
SET FIO_TMP = FIO;
```

3. Удаляем столбец FIO:

```
ALTER TABLE SOTR DROP FIO;
```

4. Создаем новый столбец FIO с необходимыми характеристиками:

```
ALTER TABLE SOTR
ADD FIO VARCHAR(25) COLLATE PXW_CYRL;
```

5. Переписываем данные из временного столбца FIO_TMP в новый столбец FIO*:*

```
UPDATE SOTR
SET FIO = FIO_TMP;
```

6. Удаляем временный столбец FIO_TMP:

```
ALTER TABLE SOTR DROP FIO_TMP;
```

Необходимо отметить, что изменение характеристик столбца, а также удаление столбца может закончиться неудачей, если:

– столбец приобретает атрибуты PRIMARY KEY или UNIQUE, но старые значения в столбце нарушают требования уникальности данных;

– удаляемый столбец входил как часть в первичный или внешний ключ, что привело к нарушению ссылочной целостности между таблицами;

– столбцу были приписаны ограничения целостности CHECK на уровне таблицы;

– столбец использовался в иных компонентах БД - в просмотрах, триггерах, в выражениях для вычисляемых столбцов.

Это свидетельствует о том, что в случае необходимости изменения атрибутов столбца или в случае удаления столбца сначала необходимо тщательно проанализировать, какие последствия для таблицы и базы данных в целом может повлечь такое изменение или удаление.

Изменение атрибутов столбца

Добавление нового столбца в таблицу БД производится оператором

```
ALTER TABLE <имя_таблицы> ADD <определения_столбца>;
```

Добавление новых ограничений целостности:

```
ALTER TABLE <имя_таблицы> ADD [CONSTRAINT  <имя_ограничения>]
<определения_целостности>;
```

Удаление столбца (столбцов) из таблицы:

```
ALTER TABLE <имя_таблицы> DROP <имя_столбца!> [,<имя
столбца2>...];
```

Удаление ограничений целостности (уровень таблицы):

```
ALTER TABLE <имя_таблицы> DROP <имя_ограничения_целостности>;
```

Пример. Для таблицы MODEL

```
CREATE TABLE MODEL (
N_MODEL INTEGER NOT NULL PRIMARY KEY,
N_FIRMA INTEGER NOT NULL,
MODEL VARCHAR(50),
CONSTRAINT MODEL_FIRMA FOREIGN KEY(N_FIRMA) REFERENCES
FIRMA(N_FIRMA));
```

удалить целостность MODEL_FIRMA:

```
ALTER TABLE MODEL MODEL_FIRMA;
```

Для удаления непоименованной целостности придется использовать ее системное имя. Его можно узнать из системной таблицы БД с именем RDB$RELATION_CONSTRAINTS.

Удаление таблицы

Удаление таблицы целиком производится оператором

```
DROP TABLE <имя таблицы>;
```

Удаление может быть блокировано для родительских таблиц, для которых в дочерних таблицах (на данный момент не удаленных) имеются ссылки по внешнему ключу этих таблиц. Удаление родительской таблицы разрушает ссылочную целостность. Поэтому следует удалить ограничения ссылочной целостности во всех дочерних таблицах, либо - по необходимости - сначала удалить сами дочерние таблицы, а затем уже удалять родительскую.

2.5 Работа с индексами

Логическое разделение на ключи и индексы

В FireBird разделено понятие ключей и индексов. Это разделение, впрочем, имеет логическую окраску и не затрагивает физическую организацию данных.

Первичный (PRIMARYKEY) и внешний (FOREIGN KEY) ключи строятся для обеспечения ссылочной целостности реляционносвязанных таблиц. Первичный ключ выполняет функции поддержания уникальности своих значений, что обусловлено его основным назначением - однозначно характеризовать запись в таблице. Для таких же целей может использоваться и просто уникальный ключ (UNIQUE).

В отличие от ключей индексы, создаваемые оператором CREATE INDEX, служат для сортировок и оптимизации доступа к данным. В конечном счете ключи и индексы преобразуются в физические индексы - специальный механизм, обеспечивающий быстрый доступ к данным. Однако, если индексам можно назначить имя, то физические индексы, реализованные на основе определений ключей, строятся и именуются системой автоматически. Например, для таблицы SOTR будут построены два физических индекса - по первичному ключу и по полю DLJ:

```
CREATE TABLE SOTR (
ID_SOTR INTEGER NOT NULL,
OTDEL VARCHAR(10) CHARACTER SET WIN1251 COLLATE PXW_CYRL,
DOLJNOST CHAR(10) CHARACTER SET WIN1251 COLLATE PXW_CYRL,
FIO VARCHAR(25) CHARACTER SET WIN1251 COLLATE PXW_CYRL, PRIMARY
KEY (ID_SOTR));
CREATE INDEX DLJ ON SOTR (DOLJNOST);
```

но именоваться они будут следующим образом:

```
DLJ INDEX ON SOTR(DOLJNOST)
RDBSPRIMARY18 UNIQUE INDEX ON SOTR(ID_SOTR);
```

Вывести определения всех индексов БД можно оператором SHOW INDEX, а проделать то же для конкретной таблицы можно оператором SHOW INDEX <имя таблицы>;

Необходимость создания индексов

Индексы необходимо создавать в случае, когда по столбцу или группе столбцов:

– часто производится поиск в базе данных (столбец или группа столбцов часто перечисляются в предложении WHERE оператора SELECT);

– часто строятся объединения таблиц;

– часто производится сортировка (т.е. столбец или столбцы часто используются в предложении ORDER BY оператора SELECT),

Не рекомендуется строить индексы по столбцам или группам столбцов, которые:

– редко используются для поиска, объединения и сортировки результатов запросов;

– часто меняют значение, что приводит к необходимости часто обновлять индекс и способно существенно замедлить скорость работы с БД;

– содержат небольшое число вариантов значения.

В случае, когда при выполнении запросов используется сортировка по одним и тем же столбцам, необходимо помнить, что создание единого индекса по этим столбцам способно ускорить выполнение запросов. Следует выделить следующие случаи.

– При использовании в запросах не всех столбцов из этого индекса следует использовать только непрерывную их последовательность; например, если индекс построен по столбцам P1,P2,P3,P4, то в некотором операторе SELECT можно указать
SELECT ... ORDER BY P1,P2,P3
но никак не
SELECT ... ORDER BY P1,P2,P4
или
SELECT ... ORDER BY P1,P3,P4
это недопустимо так, как в последних двух случаях индекс не будет использован.

– Последовательность указания в предложении ORDER BY столбцов является важной; так, для индекса, построенного по столбцам P1,P2,P3,P4, указание
SELECT ... ORDER BY P2,P1,P3
не приведет к использованию индекса для сортировки результирующего набора данных.

– При частом использовании в условной части WHERE оператора SELECT нескольких столбцов, связанных между собой операцией «или» (OR):
SELECT ... WHERE P1 = значение! OR P2 = значение2 OR P3 = ...
вместо индекса по столбцам P1,P2,P3 лучше создать несколько индексов, построенных по каждому из этих полей, поскольку в противном случае будет осуществлен последовательный просмотр всей таблицы. Индексно-последовательный доступ для индекса по столбцам P1,P2,P3 может быть осуществлен только для столбца P1; значения столбцов P2 и P3 в этом случае спонтанно разбросаны по индексу.

Создание индекса

Индекс может быть создан оператором

```
CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]]
INDEX ИмяИндекса ON ИмяТаблицы (столбец1 [,столбец2   ...]};
```

UNIQUE - требует создания уникального индекса, не допускающего одинаковых значений индексных полей для разных записей таблицы;
ASC[ENDING] - указывает на необходимость сортировки значений индексных полей по возрастанию (режим принят по умолчанию);
DESC[ENDING] - указывает на необходимость сортировки значений индексных полей по убыванию;
ИмяИндекса - имя создаваемого индекса;
ИмяТаблицы - имя таблицы, для которой создается индекс;
столбцы - имена столбцов, по которым создается индекс.

Например, для таблицы MOBIL

```
CREATE TABLE MOBIL (
ID INTEGER NOT NULL PRIMARY KEY,
DATAPRIH DATE NOT NULL,
MODEL VARCHAR(20) NOT NULL COLLATE PXW_CYRL,
KOLVO INTEGER NOT NULL);
```

создать индекс в порядке убывания значений DATAPRIH и MODEL:

```
CREATE DESC INDEX D_P
ON PRIHOD (DATAPRIH, MODEL);
```

Улучшение производительности индекса

После многократного внесения изменений в таблицу базы данных индексы этой таблицы могут быть разбалансированы. Разбалансировка приводит к тому, что «глубина» индекса (depth) возрастает сверх критического значения.

«Глубина» индекса - параметр, показывающий максимальное количество операций, необходимых для нахождения искомого значения в таблице БД с использованием данного индекса. В случае разбалансировки индекса его ценность при выполнении запросов снижается из-за увеличения времени выполнения запроса.

Поэтому время от времени необходимо производить одно из перечисленных ниже действий:

- выполнять перестройку индекса оператором ALTER INDEX;

– пересчитать показатель «выбираемого» индекса оператором SET STATISTICS;

– уничтожать индекс оператором DROP INDEX и заново создавать его оператором CREATE INDEX.

Перестройка индекса

Перестройка индекса заключается в пересоздании и балансировке индекса. Эти операции начинаются после деактивизации индекса

```
ALTER INDEX <имя_индекса> DEACTIVATE;
```

и последующей его активизации

```
ALTER INDEX <имя_индекса> ACTIVATE;
```

Деактивизация индекса полезна также в том случае, когда в таблицу БД вставляется большое число записей одновременно: при активном индексе изменения в него вносятся при добавлении каждой записи, что замедляет доступ к данным. Нельзя перестроить индекс, если он используется в данный момент в запросах других пользователей;

2.6 Оператор SELECT

Оператор SELECT - наиболее часто используемый оператор SQL. Он позволяет производить выборки данных из базы данных и преобразовывать к нужному виду полученные результаты. С его помощью можно реализовать весьма сложные условия выбора данных из различных таблиц.

В самом общем виде он имеет следующий формат:

```
SELECT [DISTINCT | ALL] {* | <значение1> [, <значение2> ...]}
FROM <таблица1> [, < таблица2> ...]
     [WHERE <условия_поиска>]
     [GROUP BY столбец [COLLATE collation]
                       [,столбец1 [COLLATE collation] ...]
     [HAVING <условия_поиска>] ...
     [UNION <оператор_select>]
     [PLAN <план_выполнения_запроса>]
     [ORDER BY <список_столбцов>]
```

Простейший вид оператора SELECT

В простейшем случае, когда требуется создание набора данных, состоящего из всех записей одной или нескольких таблиц, оператор SELECT имеет такой вид:

```
SELECT {* | <значение1> [, <значение2>...]}
FROM <таблица1> [, <таблица2>...]
```

После ключевого слова SELECT приводится список значений, каждое из которых определяет столбец возвращаемого оператором результирующего набора данных. В большинстве случаев - это имена столбцов таблиц, перечисленных после слова FROM. Символ «*» указывает, что в результат выполнения запроса нужно включить все столбцы той или иной таблицы. После FROM указывается список таблиц, из которых будет происходить выборка данных.

Например, создать набор данных, состоящий из всех столбцов и всех записей таблицы FIRMA, можно с помощью такого оператора:

```
SELECT * FROM FIRMA;
```

Такой же набор данных можно получить, если вместо звездочки перечислить все столбцы таблицы:

```
SELECT N, NAME, ABOUT FROM FIRMA;
```

Использование предложения WHERE

Предложение WHERE используется для включения в набор данных лишь нужных записей. В этом случае оператор SELECT имеет следующий формат:

```
SELECT {* | <значение1> [, <значение2> ...]}
FROM <таблица1> [, <таблица2> ...]
WHERE <условия_поиска>
```

В набор данных, возвращаемый оператором SELECT, будут включаться только те записи, которые удовлетворяют условию поиска, указанному после WHERE. Рассмотрим варианты формирования разнообразных условий поиска.

Сравнение значения столбца с константой

При сравнении значения столбца с константой условие поиска имеет такой вид:

```
<имя столбца> <оператор> <константа>,
```

где <оператор> - одна из следующих операций отношения:
= равно
< меньше
> больше
<= меньше или равно

>= больше или равно
!< не меньше (т. е. больше или равно)
!> не больше (т. е. меньше или равно)
<> не равно
!= не равно

В качестве константы явно указываются строковые или числовые значения.

Например, выбрать из таблицы FIRMA все фирмы с номерами больше трех можно следующим образом:

```
SELECT * FROM FIRMA WHERE N>3;
```

Внутреннее соединение таблиц

При сравнении значения столбца одной таблицы со значением столбца другой таблицы условие поиска имеет следующий вид:

```
<имя столбца таблицы 1> <оператор> <имя столбца таблицы 2>
```

Чтобы получить набор данных, состоящий из названий фирм производителей и моделей сотовых телефонов, у которых масса менее 100 граммов, необходимо выполнить следующую выборку:

```
SELECT FIRMA.NAME, MODEL.NAME
FROM FIRMA, MODEL
WHERE FIRMA.N=MODEL.N_FIRMA AND MODEL.WEIGHT<100;
```

При выполнении оператора для каждой записи из таблицы FIRMA ищется запись в таблице MODEL, у которой значение в поле N_FIRMA совпадает со значением в поле N текущей записи таблицы FIRMA.

При этом безразлично, в каком порядке перечислять таблицы в условии поиска, т. е. безразлично, какая из таблиц будет упомянута слева, а какая справа.

При внутреннем соединении двух таблиц A и B логический порядок формирования результирующего набора данных можно представить себе следующим образом:

1. из столбцов, которые указаны после слова SELECT, составляется промежуточный набор данных путем сцепления результирующих столбцов каждой записи из таблицы A и результирующих столбцов записи из таблицы B;
2. из получившегося набора данных отбрасываются все записи, не удовлетворяющие условию поиска в предложении WHERE.

Пример. Пусть имеются такие таблицы **A** и **B**:

Таблица **A**

Столбец P1	Столбец P2	Столбец P3
A	X	400
B	X	200
C	Y	500
D		

Таблица **B**

Столбец P1	Столбец P2
X	1
Y	2
Z	2

Тогда выполнение оператора

```
SELECT A.P1, A2.P2, B.P2
FROM A, B
WHERE A.P2 = B.P1
```

с логической точки зрения приведет к формированию такого промежуточного набора данных:

Столбец A.P1	Столбец A.P2	Столбец B.P1	Столбец B.P2
A	X	X	1
A	X	Y	2
A	X	Z	2
B	X	X	1
B	X	Y	2
B	X	Z	2
C	Y	X	1
C	Y	Y	2
C	Y	Z	2

Столбец B.P1, который не включен в результат запроса, показан для наглядности.

Затем путем исключения из промежуточного набора записей, не удовлетворяющих условию A.P2=B.P1, будет получен такой результирующий набор данных:

Столбец A.P1	Столбец A.P2	Столбец B.P2
A	X	1
B	X	1
C	Y	2

Использование псевдонимов таблиц

В рассмотренных выше примерах связывания таблиц в перечнях возвращаемых столбцов после слова SELECT и в условии поиска после WHERE перед именем столбца через точку пишется название таблицы, например:
FIRMA.N=MODEL.N_FIRMA

Указание имени таблицы перед именем столбца необходимо в том случае, когда в разных таблицах есть одноименные столбцы (как в нашем примере). Использование имен таблиц для идентификации столбцов неудобно из-за своей громоздкости. Намного лучше присвоить каждой таблице какое-нибудь краткое имя. Такие имена называются псевдонимами таблиц.

Они отделяются пробелом от фактического имени таблицы в списке таблиц после слова FROM:

```
SELECT
FROM <таблица1 псевдоним1>[, < таблица2 псевдоним2>]
WHERE ...
```

Например, запрос

```
SELECT FIRMA.NAME, MODEL.NAME
FROM FIRMA, MODEL
WHERE FIRMA.N=MODEL.N_FIRMA AND MODEL.WEIGHT<100;
```

после введения в него псевдонимов таблиц выглядит намного компактнее:

```
SELECT F.NAME, M.NAME
FROM FIRMA F, MODEL M
WHERE F.N=M.N_FIRMA AND M.WEIGHT<100;
```

Предложение ORDER BY – определение сортировки

Результирующий набор данных можно отсортировать с помощью предложения ORDER BY <список_столбцов>
Список столбцов содержит имена столбцов, по которым будет производиться сортировка. Если указаны два и более столбцов, первый столбец будет использован для глобальной сортировки, второй столбец - для сортировки внутри группы, определяемой единым значением первого столбца, и т. д.

Пример.
Показать все записи таблицы FIRMA, отсортировав их по названиям фирм:

```
SELECT F.*
FROM FIRMA F
ORDER BY F.NAME;
```

Устранение повторяющихся значений

Часто в результирующий набор данных необходимо включать не все записи с одинаковым значением какого-либо столбца (комбинации столбцов), а только одну из них. В этом случае после SELECT указывают ключевое слово DISTINCT, с помощью которого устраняются повторяющиеся записи:

```
SELECT [DISTINCT | ALL] {* | <значение1>[,<значение2> …]}
FROM <таблица1> [,< таблица2> ...]
```

Повторяющимися считаются записи, содержащие идентичные значения во всех столбцах результирующего наборе данных. Наоборот, если в результирующий запрос нужно включить все записи, после SELECT указывают слово ALL. В FireBird, как и во многих других серверах, нет необходимости использовать ALL явно, поскольку это значение действует по умолчанию.

Чтобы получить список всех функций сотовых телефонов из базы данных (функция может быть объявлена в базе данных, но не реализована ни в одной из моделей), можно использовать такой запрос:

```
SELECT DISTINCT F.NAME
FROM MODEL_FUNCTIONS MF, FUNCTIONS F
WHERE MF.N_FUNCTIONS=F.N;
```

Следует заметить, что использование DISTINCT может существенно замедлять выполнение запросов, т. к. сервер будет тратить дополнительные ресурсы на проверку каждой записи. Обычно DISTINCT применяется в запросах, использующих агрегатные функции.

Расчет значений вычисляемых столбцов

Для расчета значений вычисляемых столбцов результирующего набора данных используются арифметические выражения. При этом в списке возвращаемых столбцов после SELECT вместо имени вычисляемого столбца указывается выражение:

```
SELECT [DISTINCT | ALL] {* | <столбец1>[, <выражение1> . . . ] }
FROM <таблица1> [, <таблица2> ...]
```

Например, следующий запрос выводит массу 100 телефонов модели «6233»:

```
SELECT M.WEIGHT*100
FROM MODEL M
WHERE M.NAME='6233';
```

Если вычисляемому столбцу нужно присвоить нестандартное имя, это имя указывается после выражения вслед за ключевым словом AS:

```
SELECT ... {* | <значение1> [,<выражение1 [AS <имя столбца>]> ...]}
```

Например, использованному в предыдущем примере вычисляемому столбцу можно присвоить имя WEIGHT_100 следующим образом:

```
SELECT M.WEIGHT*100 AS WEIGHT_100
FROM MODEL M
WHERE M.NAME='6233';
```

Агрегатные функции

Агрегатные функции предназначены для вычисления итоговых значений операций над всеми записями набора данных. К агрегатным относятся следующие функции:

- COUNT(<выражение>) - подсчитывает число вхождений значения выражения во все записи результирующего набора данных;
- SUM(<выражение>) - суммирует значения выражения;
- AVG(<выражение>) - находит среднее значение;
- MAX(<выражение>) - определяет максимальное значение;
- MIN(<выражение>) - определяет минимальное значение.

Если из группы одинаковых записей нужно учитывать только одну, перед выражением в скобках включают слово DISTINCT*:*
COUNT(DISTINCT F.NAME)

Чаще всего в качестве выражения выступают имена столбцов, при этом выражение может вычисляться и по значениям нескольких таблиц.
В следующем запросе подсчитывается количество моделей сотовых телефонов, хранимых в базе данных:

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

Группировка записей

Иногда требуется получить агрегированные значения (минимум, максимум, среднее) не по всему результирующему набору данных, а по каждой из входящих в него групп записей, характеризующихся одинаковым значением какого-либо столбца. Например, выдать количество функции для каждой модели сотового телефона. В этом случае в оператор SELECT перед предложением WHERE вводится предложение

```
GROUP BY столбец [,столбец1 ...]
```

столбец – может быть указано имя столбца или его номер по порядку.

При этом необходимо, чтобы один из столбцов результирующего набора данных был представлен агрегатной функцией.

Следующим запросом можно получить список моделей сотовых телефонов с количеством заложенных в них функций:

```
SELECT M.NAME, COUNT(MF.N) AS COUNT_F
FROM MODEL M, MODEL_FUNCTIONS MF
WHERE M.N=MF.N_MODEL
GROUP BY 1;
```

Предложение HAVING – наложение ограничений на группировку записей

Если нужно в результирующем наборе данных выдавать агрегацию не по всем группам, а только по тем из них, которые отвечают некоторому условию, после предложения GROUP BY указывают предложение

```
HAVING <агрегатная_функция> <отношение> <значение>,
```

где:

- агрегатная функция - одна из функций MIN, MAX, AVG, SUM;
- отношение - одна из операций отношения =, <>, <, <=, >, >=;
- значение - константа, результат вычисления выражения или единичное значение, возвращаемое вложенным оператором SELECT.

Таким образом, после HAVING указываются условия, которые отличаются от условий, определяемых в предложении WHERE: в HAVING обязательно должна быть указана одна из агрегатных функций, в то время как в предложении WHERE такие функции указывать нельзя.

Предположим, необходимо вывести список многофункциональных (более 20 функций) сотовых телефонов. Для этого необходимо выполнить следующий запрос:

```
SELECT M.NAME, COUNT(MF.N) AS COUNT_F
FROM MODEL M, MODEL_FUNCTIONS MF
WHERE M.N=MF.N_MODEL
GROUP BY 1
HAVING COUNT(MF.N)>20;
```

Условия в HAVING и в WHERE имеют такие различия:

- HAVING исключает из результирующего набора данных группы с результатами агрегированных значений;
- WHERE исключает из расчета агрегатных значений по группировкам записи, не удовлетворяющие условию;
- в условии поиска WHERE нельзя указывать агрегатные функции.

Задание сложных условий поиска

Использование логических выражений

Сложные логические выражения строятся при помощи операторов AND, OR и NOT. Их использование, а также построение из них сложных выражений подчиняется стандартным правилам, принятым для большинства алгоритмических языков, с одним важным исключением: операции отношения в них имеют больший приоритет, чем логические операции, что избавляет от необходимости расстановки многочисленных скобок.

В следующем запросе будут получены все модели сотовых телефонов, причем для каждой модели будет указана фирма производитель и количество реализованных в ней функций:

```
SELECT F.NAME, M.NAME, COUNT(MF.N) AS COUNT_F
FROM FIRMA F, MODEL M, MODEL_FUNCTIONS MF
WHERE F.N=M.N_FIRMA AND M.N=MF.N_MODEL
GROUP BY 1,2;
```

Использование BETWEEN

В условие поиска можно указать, что некоторое значение (столбец или вычисленное значение) должно находиться в интервале между значением1 и значением2:

```
<значение> [NOT] BETWEEN <значение1> AND <значение2>
```

Показать сведения обо всех моделях сотовых телефонов масса которых от 90 до 100 грамм:

```
SELECT M.*
FROM MODEL M
WHERE M.WEIGHT BETWEEN 90 AND 100;
```

Использование IN

Если нужно, чтобы значение какого-либо столбца (или результат вычисления некоторого выражения) совпадало с одним из дискретных значений, в условии поиска указывается предложение

```
<значение> [NOT] IN (<значение1> [, <значение2> ...])
```

В этом случае в результирующий набор данных будут включены только те записи, для которых значение, стоящее слева от IN, равно одному из значений, указанных в списке (<значение1> [, <значение2> ...]).

Показать сведения обо всех моделях сотовых телефонов, появившихся в продаже в 2005 и 2007 годах. Для этого необходимо выполнить следующий запрос:

```
SELECT M.*
FROM MODEL M
WHERE M.PYEAR IN ('2005','2007');
```

Существует вторая форма использования IN, в которой список возможных значений возвращается вложенным запросом SELECT. Этот вариант рассматривается в подразделе, посвященном вложенным запросам.

Использование STARTING

Если в условии поиска нужно, чтобы значение какого-либо символьного столбца или выражения начиналось с определенной подстроки, в запрос включается предложение

```
<значение> [NOT] STARTING [WITH] <подстрока>
```

Например, получить список фирм производителей сотовых телефонов, название которых начинающихся с буквы «S», можно:

```
SELECT *
FROM FIRMA
WHERE NAME STARTING WITH 'S';
```

Использование CONTAINING

Если нужно, чтобы значение какого-либо символьного столбца или выражения включало в себя (не важно, начиная с какого символа) определенную подстроку, в запрос включается предложение

```
<значение> [NOT] CONTAINING <значение>
```

Показать список записей таблицы FIRMA у которых в поле About содержит подстроку «Стр»:

```
SELECT *
FROM FIRMA
WHERE ABOUT CONTAINING 'Стр';
```

Использование функции UPPER

Функция UPPER(<значение>) преобразует все буквы аргумента <значение> (содержимого столбца, результата вычисления выражения) к заглав-

ным. Обычно эта функция используется в условиях поиска, когда необходимо игнорировать возможную разницу в высоте букв. Функция UPPER может фигурировать как в списке столбцов результирующего набора данных (после SELECT), так и в условии поиска в предложении WHERE.

Пример.
Пусть необходимо вывести названия всех фирм заглавными буквами. Для этого необходимо выполнить следующий запрос:

```
SELECT UPPER(F.NAME)
FROM FIRMA F;
```

Использование LIKE

Предложение LIKE определяет шаблоны сравнения строковых значений. Если необходимо, чтобы сравниваемое значение (значение столбца или результат вычисления строкового выражения) удовлетворяло шаблону, в условии поиска необходимо указать

```
<значение> [NOT] LIKE <шаблон> [ESCAPE <подшаблон>]
```

В шаблоне используются специальные символы - «%» и «_». Символ «%» (процент) означает, что на его месте может быть строка любой длины, а символ «_» (подчеркивание) используется для указания любого единичного символа. Например:

LIKE "%USD" требует, чтобы строковое значение оканчивалось символами «USD» независимо от того, какие символы (и сколько их) предшествуют этому окончанию.

LIKE "__94" указывает, что сравниваемое значение может содержать 4 символа, из которых первые два - любые, а два последних - «94».

Если в предложении LIKE символы «%» или «_» должны использоваться в шаблоне подобия как обычные символы (без учета их специальных функций), в запрос включается предложение ESCAPE <символ>.

ESCAPE определяет символ <символ>, появление которого в шаблоне отменяет специальные функции следующего за ним символа:
WHERE STOLBEZ LIKE "%!%" ESCAPE "!"
(строки STOLBEZ должны заканчиваться символом «%»).

Пусть нужно найти информацию о фирмах в названии которых присутствует слово «SONY». Выборку необходимо произвести без учета регистра. Для этого необходимо выполнить следующий запрос:

```
SELECT F.NAME
FROM FIRMA F
WHERE UPPER(F.NAME) LIKE '%SONY%';
```

Использование функции CAST

Иногда возникает потребность трактовать значение одного типа как значение другого типа. Например, использовать числовое значение как символьную строку или наоборот. В этом случае применяют функцию

```
CAST (<значение> AS <тип_данных>)
```

Функция CAST делает копию значения, преобразуя его к указанному типу данных. При этом не следует забывать о множестве типов данных, в которые может быть преобразовано значение:

Тип данных	Можно привести к типам
NUMERIC	CHARACTER, DATE
CHARACTER	NUMERIC, DATE
DATE	CHARACTER, NUMERIC

Пусть требуется найти модели сотовых телефонов значение массы которых оканчивается не «0». Для этого необходимо выполнить следующий запрос:

```
SELECT *
FROM MODEL M
WHERE CAST(M.WEIGHT AS CHAR(3)) NOT LIKE '__0';
```

Использование подзапросов

Часто невозможно решить поставленную задачу путем использования единственного запроса. Например, в тех случаях, когда при использовании условия поиска <сравниваемое_значение> <оператор> <значение,с которым сравнивать> в предложении WHERE параметр <значение, с которым сравнивать> заранее не определен и должен вычисляться в момент выполнения оператора SELECT или представляет собой не одно, а несколько значений.

В такого рода случаях используются подзапросы (вложенные запросы). Оператор SELECT с подзапросом имеет такой вид:

```
SELECT ...
FROM ...
WHERE <сравниваемое_значение> <оператор> (SELECT ...)
```

Синтаксис вложенного запроса ничем не отличается от синтаксиса основного запроса, и, следовательно, вложенный запрос может, в свою очередь, содержать подзапрос.

Следующий запрос выдает информацию по моделям сотовых телефонов, у которых масса выше средней массы сотового телефона:

```
SELECT  M.*
FROM MODEL M
WHERE M.WEIGHT > (SELECT AVG(M.WEIGHT)  FROM MODEL M);
```

Дополнительные возможности использования подзапросов, возвращающих единичное значение

Использование EXISTS

Бывают случаи, когда в условии поиска нужно указать, что из таблицы требуется отобрать только те записи, для которых подзапрос возвращает одно или более значений. В этом случае в условии поиска указывается предложение EXISTS (<подзапрос>).

Пример.
Вывести функции сотовых телефонов реализованные хотя бы в одной модели сотового телефона. Для этого выполним следующий запрос:

```
SELECT F.NAME
FROM FUNCTIONS F
WHERE EXISTS (SELECT MF.N
              FROM MODEL_FUNCTIONS MF
              WHERE MF.n_functions=F.N);
```

Использование SINGULAR

Если в условии поиска нужно указать, что из таблицы требуется выбрать лишь те записи, для которых подзапрос возвращает только одно значение, указывается предложение SINGULAR (<оператор_select>).
Пример.
Вывести функции сотового телефона, реализованные только в одной модели телефона. Для этого выполним следующий запрос:

```
SELECT F.NAME
FROM FUNCTIONS F
WHERE SINGULAR (SELECT MF.N
              FROM MODEL_FUNCTIONS MF
              WHERE MF.n_functions=F.N);
```

Использование подзапросов, возвращающих множество значений

Использование ALL, SOME

Если в условиях поиска необходимо указать, что сравниваемое значение (значение столбца, результат вычисления выражения) должно находиться

в определенных отношениях со всеми или некоторыми значениями из множества значений, возвращаемых подзапросом, применяют предложение типа

```
<сравниваемое значение> {[NOT] <оператор>
{ALL | SOME | ANY} (<подзапрос>)
```

Отношение сравниваемого значения и значений, возвращаемых подзапросом, устанавливается словами ALL и SOME (ANY):

– ALL указывает, что условие поиска будет истинно только тогда, когда сравниваемое значение находится в нужном отношении со всеми значениями, возвращаемыми подзапросом; например, условие WHERE STOLBEZ > ALL (SELECT POLE FROM TABLIZA) требует, чтобы текущее значение столбца STOLBEZ было больше любых значений в столбце TABLIZA.POLE;

– SOME (вместо него можно указать ANY) - условие поиска истинно, когда сравниваемое значение находится в нужном отношении хотя бы с одним значением, возвращаемым подзапросом, например: WHERE STOLBEZ > SOME (SELECT POLE FROM TABLIZA) определяет, что текущее значение столбца STOLBEZ должно быть больше хотя бы одного значения в столбце POLE из таблицы TABLIZA.

Использование HAVING и агрегатных функций для вложенных подзапросов

Если в условиях поиска для вложенного запроса нужно указать агрегатную функцию, используется предложение HAVING.

Определим фирмы производители сотовых телефонов, у которых среднее количество функций в сотовых телефонах больше среднего количества функций сотовых телефонов среди фирм из базы данных. Т.е. определим фирмы, в среднем наделяющие свои модели сотовых телефонов наибольшей функциональностью. Для этого выполним запрос:

```
SELECT F.NAME
FROM MODEL_FUNCTIONS MF, MODEL M, FIRMA F
WHERE F.N=M.N_FIRMA AND M.N=MF.N_MODEL
GROUP BY 1
HAVING AVG(CAST(MF.N_FUNCTIONS AS FLOAT)) > ALL
       (SELECT AVG(CAST(MF.N_FUNCTIONS AS FLOAT))
        FROM MODEL_FUNCTIONS MF, MODEL M, FIRMA F
        WHERE F.N=M.N_FIRMA AND M.N=MF.N_MODEL);
```

Внешние соединения

Внешнее соединение определяется в предложении FROM согласно такой спецификации:

```
SELECT {* | <значение1> [, <значение2> ...]} FROM <таблица1> <вид
соединения> JOIN <таблица2> ON <условие  поиска>
```

Внешнее соединение отличается от внутреннего тем, что в результирующий набор данных включаются также записи ведущей таблицы соединения, которые объединяются с пустым множеством записей другой таблицы. Какая из таблиц будет ведущей, определяет вид соединения:

– LEFT - (левое внешнее соединение), когда ведущей является таблица 1 (расположена слева от вида соединения);

– RIGHT - (правое внешнее соединение), когда ведущей является таблица 2 (расположена справа от вида соединения);

– FULL - (полное внешнее соединение), когда ведущими таблицами являются и таблица 1, и таблица 2. В результирующий набор данных включаются все записи обеих таблиц по следующему алгоритму. Если для записи таблицы 1 имеются записи таблицы 2, удовлетворяющие условию соединения, в результирующий набор данных будут включены все комбинации соединения таких записей таблиц 1 и 2; в противном случае в результирующий набор данных будет включена запись таблицы 1, соединенная с пустой записью. То же относится и к записям таблицы 2: если для записи таблицы 2 имеются записи таблицы 1, удовлетворяющие условию соединения, в результирующий запрос будут включены все комбинации соединения таких записей таблиц 2 и 1; в противном случае в результирующий набор данных будет включена запись таблицы 2, соединенная с пустой записью.

Пусть имеем такие таблицы:

Таблица A

Столбец P1	Столбец P2	Столбец P3
A	X	400
B	X	200
C	Y	500
D		

Таблица B

Столбец P1	Столбец P2
X	1
Y	2
Z	2

Тогда выполнение оператора SELECT, реализующего внешнее левое соединение:

```
SELECT A.P1, AP2, B.P2
FROM A LEFT JOIN B ON A.P2=B.P1;
```

приведет к выдаче такого результирующего набора данных:

Столбец A.P1	Столбец A.P2	Столбец B.P3
A	X	1
B	X	1
C	Y	2
D		

Пунктиром показаны столбцы ведущей таблицы A.

Выполнение оператора SELECT, реализующего внешнее правое соединение:

```
SELECT A.P1, AP2, B.P2
FROM A RIGHT JOIN B ON A.P2=B.P1;
```

приведет к выдаче такого результирующего набора данных:

Столбец A.P1	Столбец A.P2	Столбец B.P3
A	X	1
B	X	1
C	Y	2
		2

Пунктиром показаны столбцы ведущей таблицы B.

UNION – объединение результатов выполнения нескольких операторов SELECT

Иногда бывает полезным объединять два или более результирующих набора данных, возвращаемых в результате выполнения операторов SELECT. Такое объединение производится при помощи оператора UNION. Результирующие наборы данных должны иметь одинаковую структуру, т. е. одинаковый состав возвращаемых столбцов. Если в результирующих наборах данных имеется одна и та же запись, в сводном наборе данных она не дублируется.
Соединим результаты выполнения двух запросов:

```
SELECT F.NAME AS "FIRMA", M.NAME AS "MODEL"
FROM FIRMA F, MODEL M
WHERE F.N=M.N_FIRMA AND F.NAME='Samsung';

SELECT F.NAME AS "FIRMA", M.NAME AS "MODEL"
FROM FIRMA F, MODEL M
WHERE F.N=M.N_FIRMA AND F.NAME='Nokia';
```

Результат объединения будет следующим:

```
SELECT F.NAME AS "FIRMA", M.NAME AS "MODEL"
FROM FIRMA F, MODEL M
WHERE F.N=M.N_FIRMA AND F.NAME='Nokia'
UNION
SELECT F.NAME AS "FIRMA", M.NAME AS "MODEL"
FROM FIRMA F, MODEL M
WHERE F.N=M.N_FIRMA AND F.NAME='Samsung';
```

Использование IS NULL

Если требуется выдать все записи, в которых некоторый столбец (или результат вычисления выражения) имеет значение **NULL** (т. е. не имеет никакого значения), достаточно в условии поиска указать предложение

```
<значение> IS [NOT] NULL
```

Следующий запрос возвращает все функции и модели сотовых телефонов, для которых отсутствует комментарий:

```
SELECT M.NAME, F.NAME, MF.NOTES
FROM MODEL M, MODEL_FUNCTIONS MF, FUNCTIONS F
WHERE M.N=MF.N_MODEL AND MF.N_FUNCTIONS=F.N AND MF.NOTES IS
NULL;
```

Использование операции сцепления строк

Операция || соединяет два строковых значения, которые могут быть представлены выражениями:

```
<строковое выражение1> || <строковое выражение2>
```

Эту операцию можно использовать как после слова SELECT для указания возвращаемых значений, так и в предложении WHERE.

Следующий запрос возвращает в одном столбце фирму и модель сотового телефона:

```
SELECT F.NAME || ' - ' || M.NAME AS "FIRMA_MODEL"
FROM FIRMA F, MODEL M
WHERE F.N=M.N_FIRMA;
```

2.7 Добавление, изменение, удаление записей

Язык SQL ориентирован на выполнение операций над группами записей, хотя в некоторых случаях операция может проводится и над отдельной записью.

Оператор INSERT

Оператор INSERT применяется для добавления записей в объект. В качестве объекта может выступать таблица базы данных или просмотр (VIEW), созданный оператором CREATE VIEW. В последнем случае записи могут добавляться в несколько таблиц.

Формат оператора INSERT:

```
INSERT INTO <объект> [(столбец1 [, столбец 2 …])]
{VALUES (<значение1> [, <значение2> …]) | <оператор SELECT>}
```

Список столбцов указывает столбцы, которым будут присвоены значения в добавляемых записях. Список столбцов может быть опущен. В этом случае подразумеваются все столбцы объекта, причем в том порядке, в котором они определены в данном объекте.

Поставить в соответствие столбцам списка значения можно двумя способами. Первый состоит в явном указании значений после слова VALUES, второй – в формировании значений при помощи оператора SELECT.

Явное указание списка значений

В этом случае оператор INSERT применяется для добавления одной записи и имеет формат:

```
INSERT INTO <объект> [(столбец1 [, столбец 2 …])]
VALUES (<значение1> [, <значение2> …])
```

Значения назначают столбцам по порядку следования тех и других в операторе: первому по порядку столбцу назначается первое значение, второму столбцу – второе значение и т.д.

Пример:

Добавить в таблицу FIRMA новую фирму. Для этого выполним следующий запрос.

```
INSERT INTO FIRMA(NAME, ABOUT) VALUES ('Motorola','Информация о
компании Motorola . . .');
```

Поле N таблицы FIRMA заполняется автоматически с помощью триггера.

Следующий пример иллюстрирует добавление записи в таблицу FIRMA с явным указанием значения поля N:

```
INSERT INTO FIRMA(N, NAME, ABOUT) VALUES
(GEN_ID(GEN_FIRMA_N,1),'Philips','Информация о компании Philips .
. .');
```

Указание значений при помощи оператора SELECT

Второй формой оператора INSERT является:

```
INSERT INTO <объект> [(столбец1 [, столбец 2 …])]
<оператор SELECT>}
```

При этом значениями, которые присваиваются столбцам, являются значениями, возвращаемыми оператором SELECT. Порядок их назначения столбцам аналогичен предыдущей форме оператора INSERT: значение первого по порядку столбца результирующего набора данных оператора SELECT присваивается первому столбцу оператора INSERT, второй – второму и т.д. Следует обратить внимание на важную особенность: поскольку оператор SELECT в общем случае возвращает множество записей, то и оператор INSERT в данной форме приведет к добавлению в объект аналогичного количества новых записей.

Оператор UPDATE

Оператор UPDATE применяется для изменения значения в группе записей или – в частном случае – в одной записи объекта. В качестве объекта могут выступать таблицы базы данных или просмотр, созданный оператором CREATE VIEW. В последнем случае могут изменяться значения записей из нескольких таблиц.

Формат оператора UPDATE:

```
UPDATE <объект>
SET столбец1=<значение1>[,столбец2=<значение2>…]
[WHERE <условие>]
```

При корректировке каждому из перечисленных столбцов присваивается соответствующее значение. Корректировка выполняется для всех записей, удовлетворяющих условию поиска. Условие поиска задается также, как в операторе SELECT.

Если опустить WHERE <условие>, в объекте будут изменены все записи.

Используя команду UPDATE, добавим комментарий в таблицу MODEL_FUNCTION:

```
update model_functions mf
set mf.notes = 'продвинутый будильник'
where mf.n=2;
```

Оператор DELETE

Оператор Delete предназначен для удаления группы записей из объекта. В качестве объекта могут выступать таблицы базы данных или просмотр, созданный оператором CREATE VIEW. В последнем случае могут удаляться записи из нескольких таблиц. В частном случае может быть удалена только одна запись.

Формат оператора Delete:

```
Delete FROM <объект>
[WHERE <условие поиска>];
```

Удаляются все записи из объекта, удовлетворяющие условию поиска. Условие поиска задается также, как в операторе SELECT.

Если опустить WHERE <условие>, из объекта будут удалены все записи.

Удалим из таблицы MODEL_FUNCTION одну запись с индексом 9:

```
DELETE
FROM model_functions mf
where mf.n=9;
```

2.8 Работа с просмотрами VIEW

Понятие просмотра как виртуальной таблицы

В базе данных может быть определен просмотр, представляющий собой виртуальную таблицу, в которой представлены записи из одной или нескольких таблиц. Порядок формирования записей в просмотре определяется оператором SELECT. Для создания просмотра применяется оператор

```
CREATE VIEW Имя_Просмотра
[(столбец_view1[,столбец_view…])]
AS <оператор_select>[WITH CHECK OPTION];
```

где после Имя_Просмотра следует необязательный список столбцов, оператор_select есть полнофункциональный оператор SELECT, а необязательный параметр WITH CHECK OPTION определяет, допускать ли для обновляемых просмотров ввод записей, не удовлетворяющих условию форматирования просмотра.

Для удаления просмотра используется оператор

```
DROP VIEW ИмяПросмотра;
```

Преимущества создания просмотров:

– единожды определив просмотр, не нужно всякий раз формировать оператор SELECT; это важно для сложных операторов SELECT, выполняющих соединение одной или нескольких таблиц;

– просмотр может предоставлять подмножество столбцов из таблицы, что важно для обеспечения сохранности данных и, возможно, усиления безопасности.

Способы формирования просмотров

Просмотр может создаваться как

– вертикальный срез таблицы, когда в просмотр включается подмножество столбцов таблицы;

– горизонтальный срез таблицы, когда в просмотр включаются все столбцы, но не все записи таблицы:

– вертикально-горизонтальный срез таблицы, когда в просмотр включается подмножество столбцов и подмножество строк:

– подмножество строк и столбцов соединения разных таблиц.

Указание столбцов просмотра в операторе CREATE VIEW

Имена столбцов просмотра должны указываться в операторе CREATE VIEW в том случае, когда в качестве столбца просмотра определяется выражение.

В противном случае имена столбцов просмотра указывать не обязательно. В случае, если список имен столбцов опущен, имена столбцов считаются идентичными именам полей, возвращаемых оператором SELECT.

Обновляемые и необновляемые просмотры

Чтобы просмотр можно было обновлять, то есть применять к нему операции добавления, изменения и удаления записей, необходимо одновременное выполнение трех условий:

– просмотр должен формироваться из записей только одной таблицы;

– в просмотр должен быть включен каждый столбец таблицы, имеющий атрибут NOT NULL;

– оператор SELECT просмотра не должен использовать агрегирующих функций, режима DISTINCT, предложения HAVING, соединения таблиц, хранимых процедур и функций, определенных пользователем.

Если просмотр удовлетворяет этим условиям, к нему могут применяться операторы INSERT, UPDATE и DELITE.

Для того, чтобы к просмотру можно было применить операторы UPDATE и DELITE, для него одновременно должны выполняться два условия:

– просмотр должен формироваться из записей только одной таблицы;

– оператор SELECT просмотра не использует агрегатных функций, режима DISTINCT, предложения HAVING, соединения таблиц, хранимых процедур и функций, определённых пользователем.

Использование CHECK OPTION

Если для обновляемого просмотра указан параметр CHECK OPTION, будут отвергаться все попытки добавления новых или изменения существующих записей таким образом, чтобы нарушалось условие WHERE оператора SELECT данного просмотра.

Создадим просмотр, отображающий количество функций по каждой модели сотового телефона:

```
CREATE VIEW MODELCOUNT (MODEL_NAME, FUNCT_COUNT) AS
SELECT M.NAME,
       COUNT(MF.N)
FROM MODEL M, MODEL_FUNCTIONS MF
WHERE M.N=MF.N_MODEL
GROUP BY 1;
```

2.9 Работа с хранимыми процедурами

Понятие хранимой процедуры

Хранимая процедура - это модуль, написанный на процедурном языке и хранящийся в базе данных как метаданные (то есть как данные о данных).

Хранимую процедуру можно вызывать из программы.

Существует две разновидности хранимых процедур:

– процедуры выбора;
– процедуры действия.

Процедуры выбора могут возвращать более одного значения. В приложении имя хранимой процедуры выбора подставляется в оператор SELECT вместо имени таблицы или просмотра.

Процедуры действия вообще могут не возвращать данных и используются для реализации каких-либо действий.

Хранимым процедурам можно передавать параметры и получать обратно значения параметров, измененные в соответствии с алгоритмами работы хранимых процедур.

Преимущества использования хранимых процедур:

– одну процедуру можно использоваться многими приложениями;
– разгрузка приложений клиента путем переноса части кода на сервер и вследствие этого - упрощение клиентских приложений;
– при изменении хранимой процедуры все изменения немедленно становятся доступны для всех клиентских приложений; при внесении же измене-

ний в приложение клиента требуется повторное распространение новой версии клиентского приложения между пользователями;

– улучшенные характеристики выполнения, связанные с тем, что хранимые процедуры выполняются сервером, в частности - уменьшенный сетевой трафик.

Создание хранимой процедуры

Хранимая процедура создается оператором

```
CREATE PROCEDURE ИмяПроцедуры
[(входной_параметр тип_данных
[,входной_лараметр тип_данных...])] •
[RETURNS (выходной_параметр тип_данных
[,выходной_параметр тип_данных ...])] AS
<тело процедуры>;
```

Входные параметры служат для передачи в процедуру значений из вызывающего приложения. Изменять значения входных параметров в теле процедуры бессмысленно: эти изменения будут забыты после окончания работы процедуры.

Выходные параметры служат для возврата результирующих значений. Значения выходных параметров устанавливаются в теле процедуры и после окончания ее работы передаются в вызывающее приложение. И входные, и выходные параметры могут быть опущены, если в них нет необходимости.

Тело процедуры имеет следующий формат:

```
[<объявление локальных переменных процедуры>] BEGIN
<оператор>
[<оператор> ...] END
```

Следующая хранимая процедура FIND_MAX_WEIGHT возвращает в выходном параметре MAX_WEIGHT максимальную массу сотового телефона среди моделей фирмы, наименование которой передается во входном параметре IN_FIRMA:

```
CREATE PROCEDURE FIND_MAX_WEIGHT (IN_FIRMA VARCHAR(15))
RETURNS (MAX_WEIGHT INTEGER)
AS
BEGIN
SELECT MAX(M.WEIGHT)
FROM FIRMA F, MODEL M
WHERE F.N=M.N_FIRMA and F.NAME=:IN_FIRMA
INTO :MAX_WEIGHT;
SUSPEND;
END
```

В приведенной процедуре за ненадобностью отсутствует определение локальных переменных.

Алгоритмический язык хранимых процедур

Для написания тела хранимой процедуры применяют особый алгоритмический язык. Этот язык применяется также для написания триггеров.

Рассмотрим конструкции алгоритмического языка хранимых процедур и триггеров.

Объявление локальных переменных

Локальные переменные, если они определены в процедуре, имеют срок жизни от начала выполнения процедуры и до ее окончания. Вне процедуры такие локальные переменные неизвестны, и попытка обращения к ним вызовет ошибку. Локальные переменные используют для хранения промежуточных значений.

Формат объявления локальных переменных:

```
DECLARE VARIABLE <имя переменной > <тип данных>;
```

Пример объявления:

```
CREATE PROCEDURE A_A (A INTEGER)
RETURNS (R INTEGER)
AS
DECLARE A1 INTEGER;
DECLARE A2 INTEGER;
BEGIN
END
```

Операторные скобки BEGIN ... END

Операторные скобки BEGIN...END, во-первых, ограничивают тело процедуры, а во-вторых, могут использоваться для указания границ составного оператора.

Под простым оператором понимается единичное разрешенное действие, например: Model = "Nokia";

Под составным оператором понимается группа простых или составных операторов, заключенная в операторные скобки BEGIN...END.

Оператор присваивания

Оператор присваивания служит для занесения значений в переменные. Его формат: Имя переменной = выражение;

где в качестве выражения могут выступать переменные, арифметические и строковые выражения, в которых можно использовать встроенные функции, функции, определенные пользователем, а также генераторы.
Пример:

```
OUT_TOVAR = UPPER(TOVAR);
```

Оператор IF... THEN ... ELSE

Условный оператор IF ... THEN ... ELSE имеет такой формат:

```
IF (<условие>) THEN
<оператор1> [ELSE
<оператор2>]
```

В случае, если условие истинно, выполняется оператор 1, если ложно - оператор 2.

Оператор SELECT

Оператор SELECT используется в хранимой процедуре для выдачи единичной строки. По сравнению с синтаксисом обычного оператора SELECT в процедурный оператор добавлено предложение
INTO :переменная [, переменная...]

Оно служит для указания переменных или выходных параметров, в которые должны быть записаны значения, возвращаемые оператором SELECT (те результирующие значения, которые перечисляются после ключевого слова SELECT).

Приведенный ниже оператор SELECT возвращает среднюю массу сотового телефона для фирмы, название которой задается переменной: IN_FIRMA, и записывает ее в AVG_WEIGHT, которая может быть как локальной переменной, так и выходным параметром процедуры.

```
SELECT AVG(M.WEIGHT)
FROM FIRMA F, MODEL M
WHERE F.N=M.N_FIRMA AND F.NAME=:IN_FIRMA
INTO :AVG_WEIGHT
```

Оператор FOR SELECT ... DO

Оператор FOR SELECT ... DO имеет следующий формат:

```
FOR
<оператор SELECT>
DO
<оператор>;
```

Оператор SELECT представляется в расширенном синтаксисе для алгоритмического языка хранимых процедур и триггеров, то есть в нем может присутствовать предложение INTO.

Алгоритм работы оператора FOR SELECT ... DO заключается в следующем. Выполняется оператор SELECT, и для каждой строки полученного результирующего набора данных выполняется оператор, следующий за словом DO. Этим оператором часто бывает SUSPEND, который приводит к возврату выходных параметров в вызывающее приложение.

Процедура FIRMA_MODEL_COUNT выдает количество моделей сотовых телефон по каждой фирме.

```
CREATE PROCEDURE FIRMA_MODEL_COUNT
RETURNS (
    FIRM_NAME VARCHAR(15),
    MODEL_COUNT INTEGER)
AS
BEGIN
  FOR SELECT F.NAME AS "FIRMA",
      COUNT(M.NAME) AS "MODEL_COUNT"
      FROM FIRMA F, MODEL M
      WHERE F.N=M.N_FIRMA
      GROUP BY 1
      INTO :FIRM_NAME, :MODEL_COUNT
  DO SUSPEND;
END
```

Оператор SUSPEND

Оператор SUSPEND передает в вызывающее приложение значения результирующих параметров (перечисленных после слова RETURNS в описании функции), имеющие место на момент выполнения SUSPEND. После этого выполнение хранимой процедуры приостанавливается. Когда от оператора SELECT вызывающего приложения приходит запрос на следующее значение выходных параметров, выполнение хранимой процедуры возобновляется.

В качестве примера можно рассмотреть процедуру FIRMA_MODEL_COUNT, описанную ранее.

Оператор WHILE ... DO

Оператор имеет такой формат:

```
WHILE (<условие>) DO <оператор>
```

Алгоритм выполнения оператора: в цикле проверяется выполнение условия; если оно истинно, выполняется оператор. Цикл продолжается до тех пор, пока условие не перестанет выполняться.

Рассмотрим процедуру SUM_0_N, которая подсчитывает сумму всех чисел от 0 до числа, определяемого входным параметром N. Вычисление суммы реализовано в цикле с использованием оператора WHILE...DO.

```
CREATE PROCEDURE SUM_0_N (N INTEGER)
RETURNS(S INTEGER)
AS
DECLARE VARIABLE TMP INTEGER;
BEGIN
S = 0; TMP = 1;
WHILE  (TMP <= N) DO
 BEGIN
  S = S + TMP; TMP = TMP + 1;
 END
SUSPEND;
END
```

Оператор EXIT

Оператор EXIT инициирует прекращение выполнения процедуры и выход в вызывающее приложение.

Процедура MAX_VALUE возвращает максимум из двух чисел, передаваемых как входные параметры; в случае, если одно из чисел имеет значение NULL, процедура завершается (в этом случае выходной параметр содержит значение NULL):

```
CREATE PROCEDURE MAX_VALUE(A INTEGER, B INTEGER)
RETURNS(M_V INTEGER)
AS
BEGIN
IF (:A IS NULL OR :B IS NULL) THEN EXIT;
IF (:A > :B) THEN M_V = :A; ELSE M_V = :B;
SUSPEND;
END
```

Оператор EXECUTE PROCEDURE

Оператор

```
EXECUTE PROCEDURE имя [параметр [, параметр . [RETURNING_VALUES
параметр [, параметр ...]];
```

вызывает другую хранимую процедуру. При этом после имени вызываемой процедуры перечисляются входные параметры, если они есть, а после RETURNING_VALUES - выходные параметры.

Перепишем приведенную выше процедуру FIRMA_MODEL_COUNT таким образом, чтобы из ее тела вызывалась другая процедура,

FIND_MAX_WEIGHT, возвращающая максимальный вес модели сотового телефона для заданной фирмы:

```
CREATE PROCEDURE FIRMA_MODEL_COUNT_PLUS
RETURNS (
    FIRM_NAME VARCHAR(15),
    MODEL_COUNT INTEGER,
    MAX_WEIGHT INTEGER)
AS
BEGIN
  FOR SELECT F.NAME AS "FIRMA",
      COUNT(M.NAME) AS "MODEL_COUNT"
      FROM FIRMA F, MODEL M
      WHERE F.N=M.N_FIRMA
      GROUP BY 1
      INTO :FIRM_NAME, :MODEL_COUNT
  DO
  BEGIN
  EXECUTE PROCEDURE
  FIND_MAX_WEIGHT(:FIRM_NAME)
  RETURNING_VALUES :MAX_WEIGHT;
  SUSPEND;
  END
END
```

Оператор POST _EVENT

Оператор

```
POST_EVENT "Имя_события";
```

применяется для посылки сервером клиентским приложениям сообщения о наступлении какой-либо ситуации, связанной с именем события.

Приложение должно зарегистрироваться на сервере для получения уведомления о наступлении событий и указать список интересующих приложение событий.

Пример процедуры рассылки сообщений представлен ниже.

```
CREATE PROCEDURE SEND_EVENT (
    EV_NAME VARCHAR(60))
AS
BEGIN
 POST_EVENT :EV_NAME;
 SUSPEND;
END
```

Изменение и удаление хранимых процедур

Изменение хранимой процедуры производится оператором

```
ALTER PROCEDURE ИмяПроцедуры
[ (входной_параметр тип_данных
[, входной_параметр тип_данных ... ])]
[RETURNS
(входной_параметр тип_данных
[ , входной_параметр тип_данных ...])] AS
<тело процедуры>;
```

Принципы построения оператора аналогичны изложенным выше принципам построения оператора CREATE PROCEDURE. После выполнения оператора ALTER PROCEDURE предыдущее определение процедуры заменяется на новое определение параметров, переменных и тела процедуры.

Для удаления хранимой процедуры из базы данных используется оператор DROP PROCEDURE ИмяПроцедуры;

2.10 Функции, определяемые пользователем (UDF)

Понятие функции, определяемой пользователем

Состав встроенных функций FireBird весьма небогат – в него входят функции вычисления агрегированных значений (MIN, MAX, SUM, AVG), функция преобразования букв UPPER и функция приведения типа CAST.

Часто разработчики нуждаются в дополнительных функциях, которые можно было бы использовать также, как и стандартные встроенные функции. Это могут быть функции вычисления модуля от вещественного значения, определения длины строки, усечения хвостовых или ведущих пробелов в символьных значениях, извлечение из даты значения месяца, года и т.д.

Специально для таких целей в FireBird существует аппарат функций, определяемых пользователем (User Defined Function, UDF).

Разработка UDF может осуществляться на любом алгоритмическом языке, позволяющем создать DLL (dynamic link library, динамически загружаемые библиотеки). Каждая UDF оформляется в виде функции, входящей в состав DLL. Таким образом, одна динамически загружаемая библиотека состоит минимум из одной функции.

После того как DLL разработана, она либо перемещается в подкаталог UDF каталога размещения FireBird, либо располагается в ином каталоге, путь к которому известен в операционной системе.

Каждая функция определяется оператором DECLARE EXTERNAL FUNCTION. Этот оператор устанавливает связь между функцией из DLL и ее описанием в базе данных. После этого функция может использоваться в SQL-операторах наряду со стандартными функциями FireBird.

Преимущества UDF:

– динамически загружаемая библиотека и определенные в ней функции могут использоваться более чем одной базой данных и более чем одним при-

ложением; таким образом осуществляется повторное использование однажды написанного кода;

– пользователь может реализовать в UDF достаточно сложные алгоритмы.

Объявление UDF в базе данных

Для объявления функции, определенной пользователем, в базе данных FireBird, необходимо выполнить оператор:

```
DECLARE EXTERNAL FUNCTION ИмяФункции
[<Тип данных> | CSTRING (число)[,<Тип данных> | CSTRING (число)…]]
RETURNS {<Тип данных>[BY VALUE] | CSTRING (число)}
ENTRY_POINT «<Имя функции в DLL>»
MODULE_NAME «<Имя DLL>»;
```

ИмяФункции – имя, под которым функция будет известна в базе данных. Это имя может отличаться от имени UDF в DLL. После имени функции следует список типов входных параметров функции. Это либо тип данных, разрешенный в FireBird, либо CSTRING (число для строковых значений). Число определяет размер строкового значения в символах. Если число меньше действительного размера строки, строка при передачи в UDF усекается.

После слова RETURNS указывается тип возвращаемого параметра функции. Это либо тип данных, разрешенный в FireBird, либо CSTRING (число) для строковых значений. Слова BY VALUE означают, что результат функции возвращается по значению, а не по ссылке.

После ENTRY_POINT в кавычках указывается имя функции в DLL, а после слов MODULE_NAME – имя модуля DLL (без разрешения).

Удалить из базы данных объявление функции, определенной пользователем, можно при помощи оператора

```
DROP EXTERNAL FUNCTION ИмяФункции;
```

2.11 Работа с триггерами

Триггер - это процедура базы данных, автоматически вызываемая SQL-сервером при обновлении, удалении или добавлении новой записи в таблицах базы данных. Непосредственно из программы к триггерам обратиться нельзя. Нельзя и передавать им входные параметры и получать от них значения выходных параметров. Триггеры всегда реализуют действие.

По событию изменения таблиц базы данных триггеры различаются на вызываемые при:

– добавлении новой записи;
– изменении существующей записи;
– удалении записи.

По отношению к событию, производящему их вызов, триггеры различаются на:

– выполняемые до наступления события;
– выполняемые после наступления события.

Преимущества использования триггеров приведены ниже.

– Автоматическое обеспечение каскадных воздействий в дочерних таблицах при изменении, удалении записи в родительской таблице выполняется на сервере. Пользователю нет необходимости заботиться о программной реализации каскадных воздействий. Поскольку каскадные воздействия выполняет сервер, нет необходимости пересылать изменения в таблицах базы данных из приложения на сервер, что снижает загрузку сети.
– Изменения в триггерах не влекут необходимости изменения программного кода в клиентских приложениях и не требуют распространения новых версий клиентских приложений.

При откате транзакции откатываются также и все изменения, внесенные в базу данных триггерами.

Создание триггеров

Триггер создается оператором

```
CREATE TRIGGER ИмяТриггера FOR ИмяТаблицы [ACTIVE | INACTIVE]
{BEFORE | AFTER} {DELETE | INSERT | UPDATE} [POSITION номер] AS
<тело триггера>
```

Для определения тела триггера используется процедурный язык. В него добавляется возможность доступа к старому и новому значениям столбцов изменяемой записи OLD и NEW - возможность, недоступная при определении тела хранимых процедур.

Структура тела триггера:

```
[<объявление локальных переменных>]
BEGIN
<оператор>
END
```

Определение заголовка триггера

Заголовок триггера имеет формат

```
CREATE TRIGGER ИмяТриггера FOR ИмяТаблицы
[ACTIVE | INACTIVE]
{BEFORE | AFTER}
{DELETE | INSERT | UPDATE}
[POSITION номер]
```

– ACTIVE | INACTIVE - указывает, активен триггер или нет. Можно определить триггер «прозапас», установив для него INACTIVE. В дальнейшем можно переопределить триггер как активный. По умолчанию действует ACTIVE.

– BEFORE | AFTER - указывает, будет выполняться триггер до (BEFORE) или после (AFTER) запоминания изменений в базе данных.

– DELETE | INSERT | UPDATE - указывает операцию над таблицей базы данных, при выполнении которой срабатывает триггер.

– POSITION номер - указывает, каким по счету будет выполняться триггер в случае наличия группы триггеров, обладающих одинаковыми характеристиками операции и времени (до, после операции) вызова триггера. Значение номера задается числом в диапазоне 0..32767. Триггеры с меньшими номерами выполняются раньше.

Например, если определены триггеры

```
CREATE TRIGGER A FOR MODEL BEFORE INSERT POSITION 1 ...
CREATE TRIGGER C FOR MODEL BEFORE INSERT POSITION 0 ...
CREATE TRIGGER D FOR MODEL BEFORE INSERT POSITION 44 ...
CREATE TRIGGER B FOR MODEL AFTER INSERT POSITION 1 ...
CREATE TRIGGER E FOR MODEL AFTER INSERT POSITION 44 ...
```

для операции добавления новой записи в таблицу **MODEL** они будут выполнены в последовательности C, A, D, B, E.

Значения OLD и NEW

Значение OLD.ИмяСтолбца позволяет обратиться к состоянию столбца, имевшему место до внесения возможных изменений, а значение NEW.ИмяСтолбца - к состоянию столбца после внесения изменений.

В том случае, если значение в столбце не изменилось, OLD.ИмяСтолбца будет равно NEW.ИмяСтолбца.

Следующий триггер срабатывает при удалении модели сотового телефона из таблицы MODEL и удаляет связанные с этой моделью записи в таблице MODEL_FUNCTIONS:

```
CREATE trigger model_before_delete for model
active before delete position 0
AS
begin
  delete
  from model_functions mf
  where mf.n_model=old.n;
end
```

Если в родительской таблице удалена запись, должны быть удалены все связанные с ней записи в дочерней таблице. Такое воздействие на дочернюю таблицу носит название каскадного удаления.

С использованием триггеров может быть реализован журнал изменений информации таблиц баз данных. Т.е. на каждое событие, совершаемое над записями таблицы, срабатывает триггер, который фиксирует изменения в таблице и сохраняет информацию об изменениях в специальных таблицах.
Триггеры активно используются для реализации бизнес-правил. В частности, это может быть установка с помощью генераторов уникальных значений индексных полей, накапливание статистики в других таблицах и многое другое.

Изменение и удаление триггера

Изменить существующий триггер можно при помощи оператора:

```
ALTER TRIGGER ИмяТриггера FOR ИмяТаблицы
[ACTIVE | INACTIVE]
{BEFORE | AFTER)
{DELETE | INSERT | UPDATE}
[POSITION номер] AS <тело триггера>
```

После выполнения этого оператора все старые определения триггера заменяются на определения, указанные в операторе ALTER TRIGGER. Для удаления триггера следует воспользоваться оператором
DROP TRIGGER ИмяТриггера.

2.12 Использование генераторов

Часто в состав первичного или уникального ключа входят цифровые поля, значения которых должны быть уникальны, то есть не повторяться ни в какой другой записи таблицы. В одних случаях такое значение является семантически значимым и формируется пользователем по определенному алгоритму - например, номер лицевого счета в банке. В других случаях лучше предоставить выработку такого значения приложению или серверу баз данных.

Для локальных СУБД (например, Paradox) для указанной цели применяются автоинкрементные поля.

В FireBird отсутствует аппарат автоинкрементных столбцов. Вместо этого для установки уникальных значений столбцов можно использовать аппарат генераторов.

Генератором называется хранимый на сервере базы данных механизм, возвращающий уникальные значения, никогда не совпадающие со значениями, выданными тем же самым генератором в прошлом.

Для создания генератора используется оператор:

```
CREATE GENERATOR ИмяГенератора;
```

Для генератора необходимо установить стартовое значение при помощи оператора:

```
SET GENERATOR ИмяГенератора TO СтартовоеЗначение;
```

При этом **СтартовоеЗначение** должно быть целым числом.

Для получения уникального значения к генератору нужно обратиться с помощью функции

```
GEN_ID (ИмяГенератора, шаг);
```

Эта функция возвращает увеличенное на шаг предыдущее значение, выданное генератором (или увеличенное на шаг стартовое значение, если ранее обращений к генератору не было). Значение шага должно принадлежать диапазону $-2^{31}...+2^{31}-1$.

Не рекомендуется переустанавливать стартовое значение генератора или менять шаг при разных обращениях к GEN_ID. В противном случае генератор может выдать неуникальное значение, и как следствие будет возбуждено исключение при попытке запоминания новой записи в таблицу базы данных.

Присваивание ключевому столбцу уникального значения может быть реализовано с помощью триггера, вызываемого перед запоминанием новой записи в базе данных:

```
CREATE trigger model_bi for model
active before insert position 0
as
begin
  if (new.n is null) then
    new.n = gen_id(gen_model_N,1);
end
```

2.13 Транзакции

Откат изменений и целостность БД

Существуют механизмы отката изменений в базе данных в случае невыполнения условия успешного завершения всех операций в составе группы. Один из таких механизмов носит название обработка транзакций.

Понятие транзакции

Транзакция - это единичное или чаще групповое изменение базы данных, которое или выполняется полностью, или не выполняется вообще. Ре-

зультаты выполнения транзакции записываются в базу данных только в том случае, если вся транзакция завершилась успешно. Таким образом, транзакция переводит базу данных из одного целостного состояния в другое.

Уровни изоляции транзакций

При одновременной работе нескольких клиентов с одной и той же базой данных возникают проблемы одновременного изменения данных.

Пусть пользователь А получил данные из таблицы FIRMA и впоследствии изменил их. В это время с той же записью в таблице FIRMA работает пользователь В. Он также изменил данные в той же записи, что и А, и пытается подтвердить их. Пользователь С работает с таблицей FIRMA в режиме только для чтения. Сразу же возникает группа вопросов - позволять или не позволять В изменять запись, если А еще не подтвердил ее изменение? Позволять ли С видеть изменения, внесенные А и В? Может ли А видеть изменения, внесенные В, и наоборот?

Для разрешения указанных проблем на стороне клиента определены три уровня изоляции (разграничения) транзакций.

При уровне изоляции транзакций Dirty Read (в буквальном переводе - грязное чтение) конкурирующие транзакции видят изменения, внесенные, но не подтвержденные текущей транзакцией. Если текущая транзакция откатит сделанные изменения, другие транзакции будут располагать недостоверными данными. Этот уровень изоляции может привести к серьезным ошибкам и применяется редко - фактически он не изолирует конкурирующие транзакции.

При уровне Read Committed (чтение подтверждений) конкурирующие транзакции оперируют только подтвержденными изменениями. Если транзакция А не завершена и в этот момент стартует транзакция В, она получит данные в том состоянии, которое было на момент старта А. Однако, если транзакция А внесла изменения в запись и подтвердила их, а незавершившаяся транзакция В вновь прочитала эти данные, она увидит изменения.

Уровень изоляции транзакций Repeatable Read (повторяющееся чтение) заставляет конкурирующие транзакции оперировать с собственными (локальными) версиями одной и той же записи. Если транзакция А читает какую-то запись, она, как и при уровне Read Committed, получит последнюю подтвержденную ее версию. Однако, если другая транзакция изменит эту запись, а транзакция А вновь прочитает ее, она не увидит внесенных изменений, т. е. А видит всегда одну и ту же запись.

Изменения, которые вносят конкурирующие транзакции в одну и ту же запись, могут конфликтовать. В этом случае фактически изменит данные первая завершившаяся транзакция, а попытки подтвердить свои изменения второй и всеми другими незавершенными транзакциями, изменившими те же записи, будут отвергнуты.

Управление транзакциями на SQL-сервере

Firebird управляет транзакциями при помощи SQL-операторов SET TRANSACTION (начать транзакцию), COMMIT (подтвердить транзакцию) и ROLLBACK (откатить транзакцию).

Оператор SET TRANSACTION имеет формат:

```
SET TRANSACTION [READ WRITE | READ ONLY]
[WAIT | NO WAIT]
[[ISOLATION LEVEL] {SNAPSHOT [TABLE STABILITY]
| READ COMMITTED [[NO] RECORD_VERSIOM]}]
[RESERVING <список таблиц> [FOR [SHARED | PROTECTED][READ |
WRITE]], [< список таблиц > ...];
```

где

– READ WRITE READ ONLY устанавливает уровень доступа к данным (по умолчанию READ WRITE);

– WAIT NO WAIT определяет поведение при возникновении конфликта по обновлению записи данной транзакции с другой транзакцией, ранее сделавшей изменение в той же записи: WAIT (по умолчанию) побуждает данную транзакцию ожидать завершения конкурирующей транзакции; NO WAIT определяет аварийное завершение данной транзакции;

– ISOLATION LEVEL определяет уровни изоляции транзакций на сервере (по умолчанию SNAPSHOT, что соответствует уровню REPETABLE READ);

– READ COMMITTED разрешает стартующей транзакции читать только подтвержденные данные: NO RECORD_VERSION (по умолчанию) - читается последняя версия записи, даже если она не подтверждена другой транзакцией; при этом если указан параметр WAIT, стартующая транзакция будет ожидать подтверждения последней версии; RECORD_VERSION - читается последняя подтвержденная версия записи, даже если на сервере есть более поздняя, но не подтвержденная версия;

– RESERVING в рамках стартующей транзакции блокирует конкурирующим транзакциям доступ к таблицам, перечисленным в списке таблиц.

В последнем случае каждому элементу списка таблиц ставятся в соответствие параметры:

– PROTECTED READ - конкурирующие транзакции могут читать данные, но не могут изменять;

– PROTECTED WRITE - читать данные могут только транзакции с уровнями SNAPSHOT или READ COMMITTED, и никакая конкурирующая транзакция не может их изменять.

Следующий оператор стартует транзакцию без имени с уровнем изоляции READ COMMITTED, транзакция будет ожидать подтверждения обновления последней версии нужной записи:

```
SET TRANSACTION WAIT
ISOLATION LEVEL READ COMMITTED;
```

Следующий оператор стартует транзакцию T1 и блокирует таблицы TABLE1 и TABLE2 для чтения, а TABLE3 - для записи;

```
SET TRANSACTION NAME T1 WAIT ISOLATION LEVEL READ COMMITTED RE-
SERVED TABLE1, TABLE2 FOR PROTECTED READ, TABLES FOR PROTECTED
WRITE;
```

2.14 Вопросы для самопроверки

– Перечислите основные функциональные возможности СУБД FireBird?
– Какие основные типы данных СУБД FireBird Вы знаете?
– Что понимается под понятием «домен»?
– Что понимается под понятием «первичный ключ»?
– Что понимается под понятием «уникальный ключ»?
– Что понимается под понятием «внешний ключ»?
– Что такое «индекс» и для чего он используется?
– Опишите общий формат оператора SELECT?
– Опишите общий формат оператора INSERT?
– Опишите общий формат оператора DELETE?
– Опишите общий формат оператора UPDATE?
– Что такое просмотры данных? Перечислите возможности их использования?
– Что такое хранимая процедура? Какие задачи могут быть решены с ее применением?
– Что такое UDF-функция?
– Что такое триггер? Перечислите возможные варианты его использования?
– Что такое генератор?
– Что Вы понимаете под понятием «транзакция»?
– Какие уровни изоляции транзакций Вы знаете?

3 Практическая реализация

3.1 Установка и настройка программного обеспечения

Для работы необходимо установить на компьютер следующие программные продукты:

- сервер баз данных FireBird 2.0;
- клиент-оболочка баз данных IBExpert 2007;
- инструмент для анализа статистики баз данных.

Сервер баз данных FireBird можно бесплатно скачать с официального сайта разработчиков http://www.firebirdsql.org. Этот сервер является бесплатным аналогом базы данных InterBase фирмы Borland.

Для удобства работы с сервером базы данных необходимо использовать клиент-оболочку IBExpert. Ее можно скачать с официального сайта разработчиков http://www.ibexpert.com. IBExpert является бесплатно распространяемым программным средством на территории Российской Федерации.

IBAnalyst - это инструмент для анализа статистики баз данных InterBase или Firebird, поиска проблем в производительности базы данных, сопровождении, или работы приложений.

IBAnalyst решает две важные задачи:

- визуализирует статистику базы данных и сообщает об актуальных или возможных проблемах;
- предлагает проверенные советы по улучшению производительности базы данных, на основе автоматического анализа ее статистики.

Используя IBAnalyst можно буквально за несколько секунд увидеть все потенциальные и актуальные проблемы анализируемой базы данных: плохие индексы, фрагментированные таблицы, состояние транзакций и т.д., а также получить профессиональные советы о том, как решить эти проблемы.

Программное средство IBAnalyst можно скачать с официального сайта http://www.ibase.ru. IBAnalyst является бесплатным программным средством для коммерческого и некоммерческого использования на территории России и стран СНГ.

Чтобы установить сервер баз данных на операционной системе Windows, необходимо запустить файл Firebird-2.{версия}-Win32.exe и выбрать язык, который будет использован при установке. Выбираем «русский». После появления окно приветствия «Мастера установки сервера баз данных FireBird», нажмите кнопку «Далее» для продолжения установки. В следующем окне «Лицензионное соглашение» необходимо выбрать «Я принимаю условия соглашения» и нажать кнопку «Далее». В появившемся на экране окне описана важная информация, касающаяся различных версии FireBird, нажмите кнопку «Далее». В следующем окне будет предложено выбрать папку для установки FireBird. Оставляем предложенную папку, нажимаем кнопку «Далее». После этого появляется окно выбора компонентов для ус-

тановки (рисунок 3.1). Оставляем все по умолчанию, нажимаем кнопку «Далее».

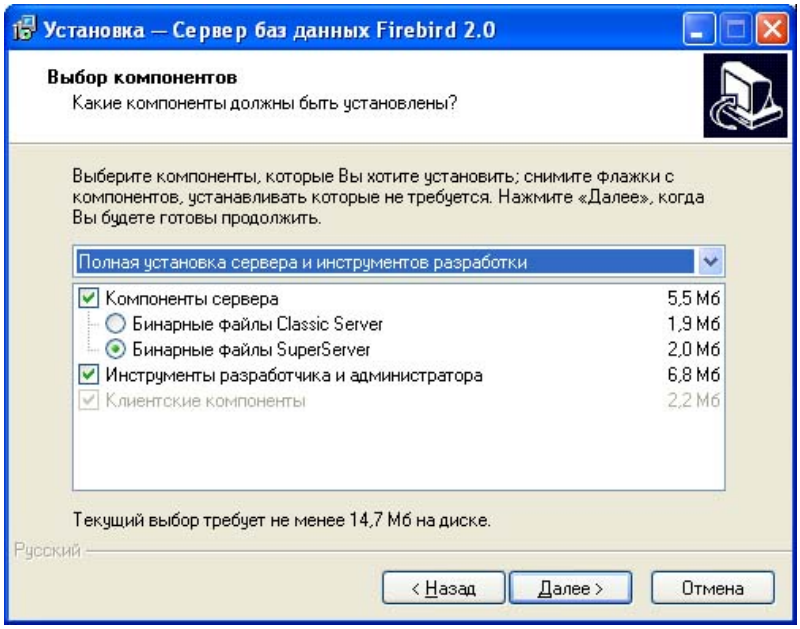

Рисунок 3.1 - Окно выбора компонентов для установки

В следующем окне предлагается выбрать название папки программы в меню «Пуск». Оставляем все по умолчанию, нажимаем кнопку «Далее». В появившемся окне необходимо выбрать дополнительные задачи, которые нужно выполнить при установке сервера баз данных FireBird (рисунок 3.2).

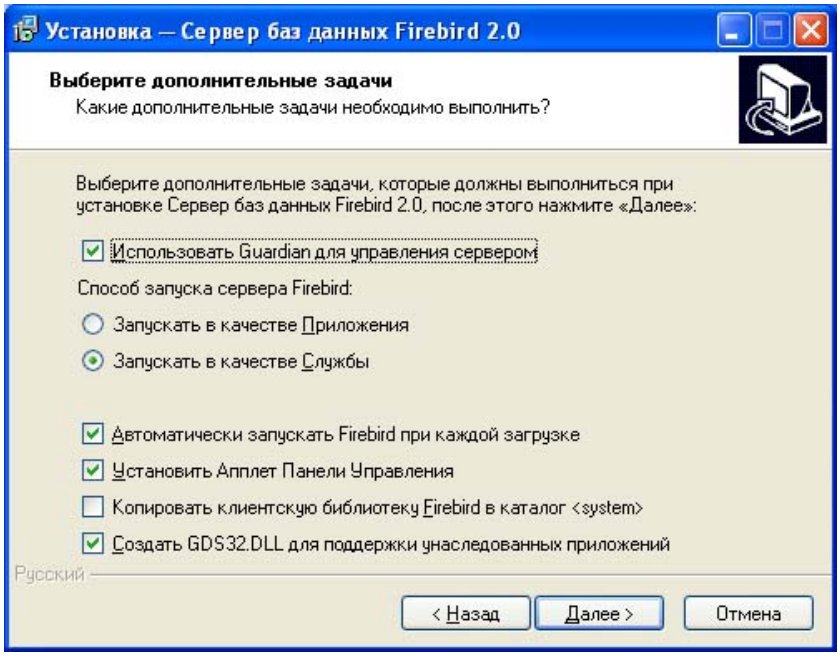

Рисунок 3.2 - Окно выбора дополнительных задач

Оставляем все настройки по умолчанию и нажимаем кнопку «Далее». В следующем появившемся окне необходимо нажать кнопку «Установить»,

после этого выполнится процесс установки. После завершения процесса установки будет выведено информационное окно, ознакомившись с его содержимым, необходимо нажать кнопку «Далее». На экране появится окно в котором предлагается осуществить запуск сервера баз данных непосредственно после инсталляции (рисунок 3.3). Нажмите кнопку «Завершить».

После выполнения описанных выше операций сервер баз данных FireBird 2.0 установлен и готов к работе.

Перейдем к процессу инсталляции и настройки утилиты IBExpert.

Для установки IBExpert необходимо запустить файл setup_trial.exe, на экране появится диалоговое окно, нажмите кнопку«Next». В следующем окне предлагается выбрать инсталлируемые компоненты. Оставляем все по умолчанию, нажимаем кнопку «Next». В следующем окне «Лицензионное соглашение» необходимо выбрать «I accept the agreement»(«Я принимаю условия соглашения») и нажать кнопку «Next». Далее в окне будет предложено выбрать папку для установки IBExpert. Оставляем предложенную папку, нажимаем кнопку «Next». В появившемся окне нажимаем кнопку «Install». Выполняется процесс инсталляции всех выбранных компонентов IBExpert, по завершению этой операции будет выдано окно окончания процесса инсталляции. В нем необходимо нажать кнопку «Finish».

Рисунок 3.3 - Завершение мастера установки сервера баз данных FireBird

Процесс инсталляции IBExpert закончен, переходим к его настройке.

Запустите IBExpert с помощью ярлыка, расположенного на рабочем столе, или через меню «Пуск». При первом запуске вам будут заданы вопросы, связанные с настройкой интерфейса IBExpert. Нажмите кнопку «Ok». Настройку интерфейса можно изменить в любой момент из среды IBExpert.

Первое, что необходимо сделать после запуска IBExpert, это настроить переменные окружения (рисунок 3.4). Для настройки необходимо в меню выбрать «Options» → «Environment Options…».

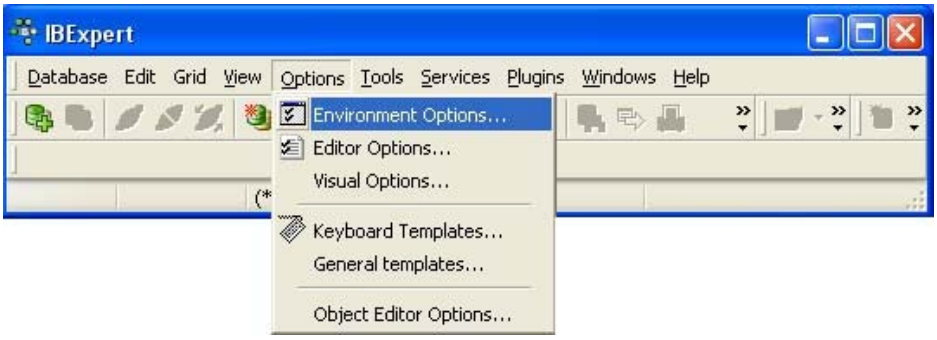

Рисунок 3.4 - Настройка переменных окружения IBExpert

В открывшемся окне (рисунок 3.5) необходимо выставить следующие значения переменных окружения:

- Interface Language – Russian;
- Default Server Version – Firebird 2.0;
- Default Character Set – WIN1251;
- Default Client Library – C:\Program Files\Firebird\Firebird_2_0\bin\fbclient.dll.

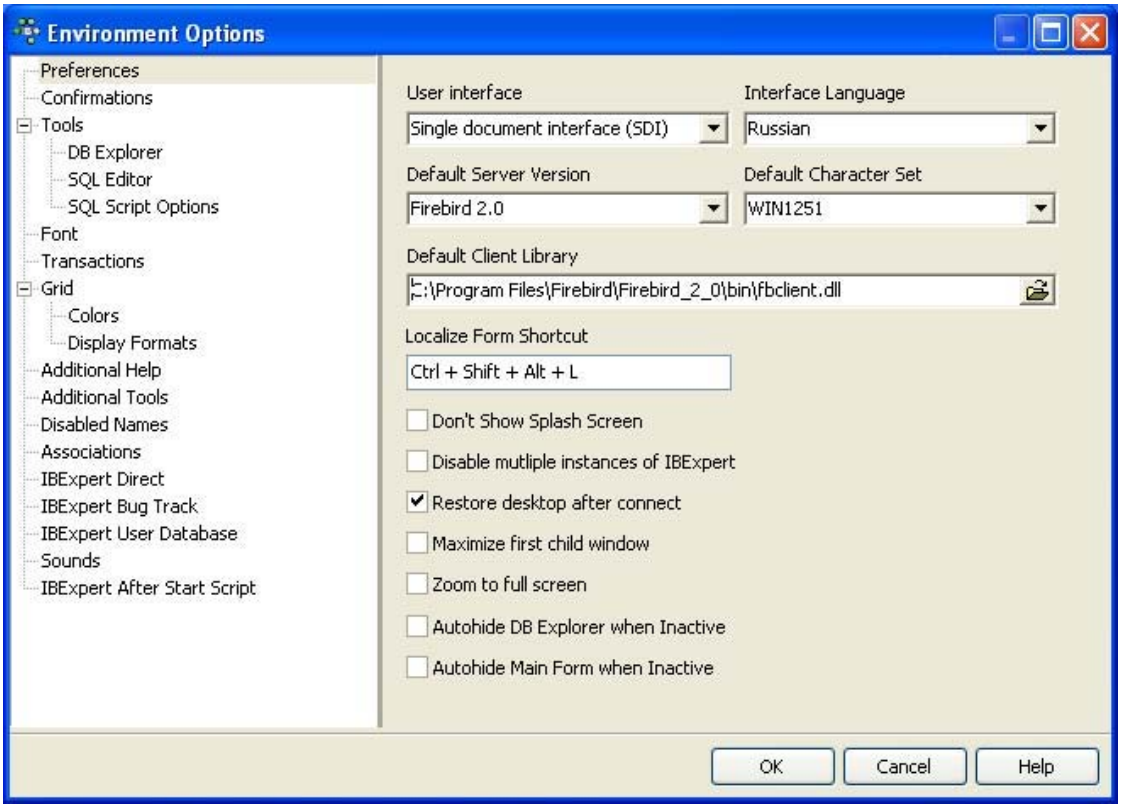

Рисунок 3.5 - Переменные окружения IBExpert

Для инсталляции IBAnalyst запускаем установочный файл «IBAnalyst_r.exe». Нажимаем в появившемся окне мастера установки (рисунок 3.6) кнопку «Далее».

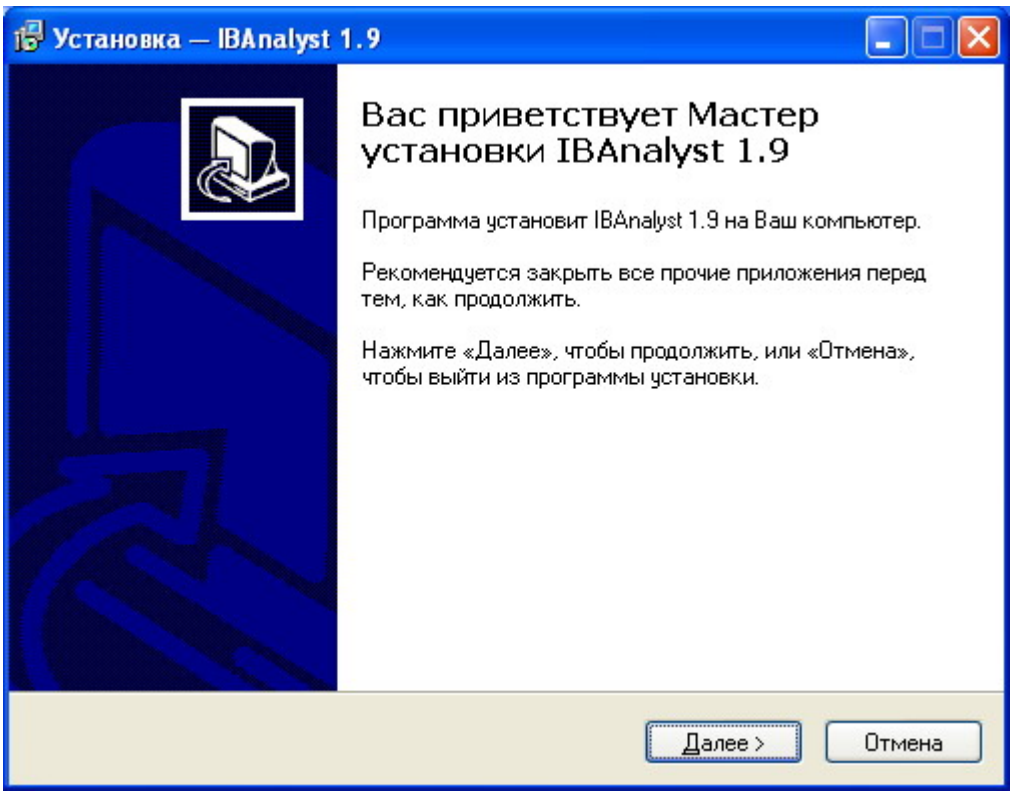

Рисунок 3.6 – Окно мастера установки IBAnalyst

В следующем окне «Лицензионное соглашение» необходимо выбрать «Я принимаю условия соглашения» и нажать кнопку «Далее». В следующем окне будет предложено выбрать папку для установки IBAnalyst. Оставляем предложенную папку, нажимаем кнопку «Далее». Далее на экране появляется диалоговое окно, в котором необходимо выбрать название папки для программ в меню «Пуск». Оставляем папку, предложенную по умолчанию, и нажимаем кнопку «Далее». Теперь необходимо подтвердить выбранные нами настройки. Для этого в появившемся окне нажимаем кнопку «Установить».

После завершения процесса установки на экране монитора выводится окно «Завершения Мастера установки» (рисунок 3.7). В этом окне можно, определить хотите ли вы ознакомится с нововведениями последней версии IBAnalyst, и посмотреть информацию файла «readme.txt». При необходимости выберите соответствующие позиции в диалоговом окне.

Запустим программное средство IBAnalyst. Для этого выполним следующие действия: меню «Пуск» → «Все программы» → «IBAnalyst» → «IBAnalyst». На экране появится окно IBAnalyst (рисунок 3.8).

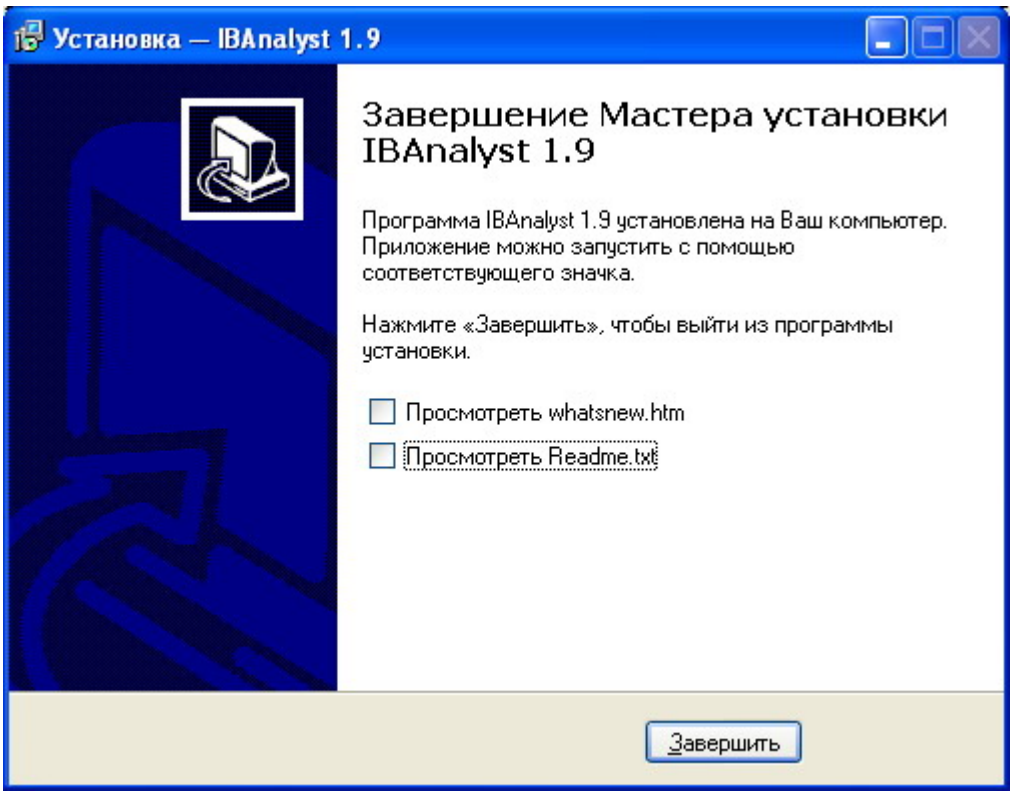

Рисунок 3.7 - Окно завершения Мастера установки

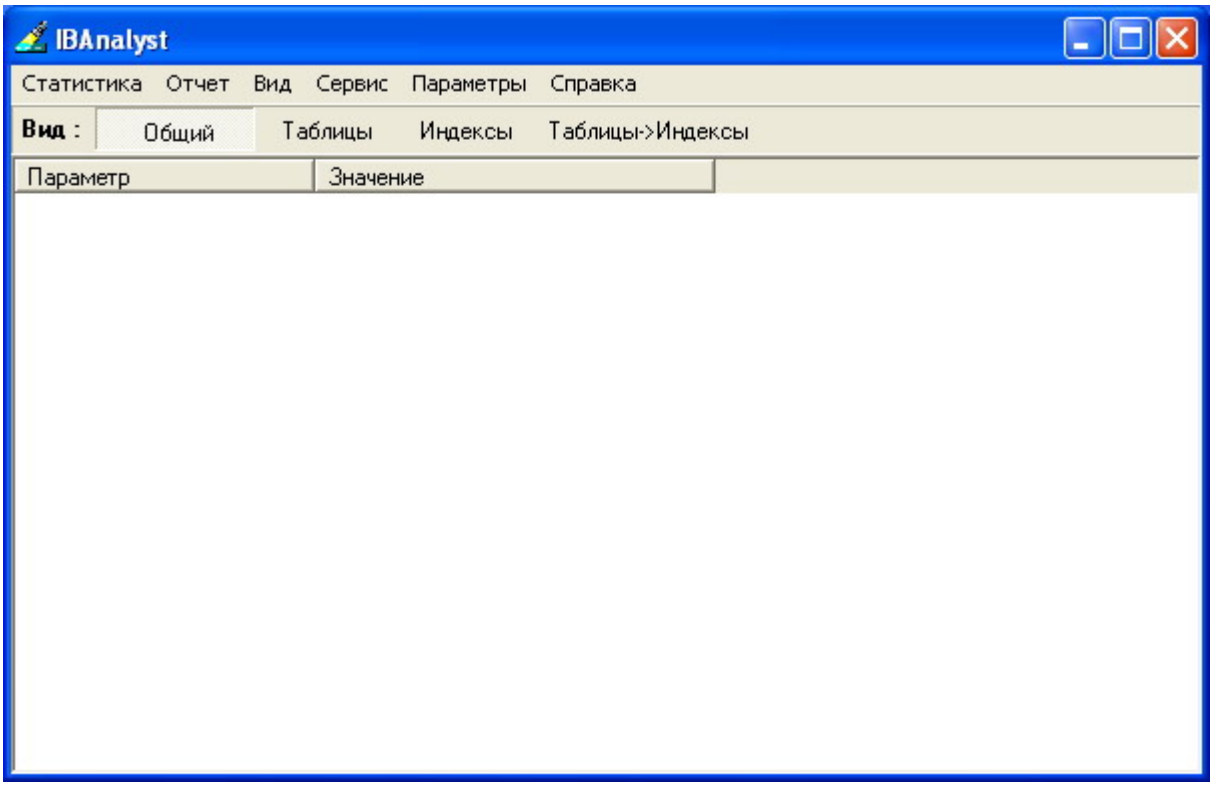

Рисунок 3.8 - Окно программы IBAnalyst

Для подключения к базе данных выполним через меню программы IBAnalyst «Статистика» → «Подключить через Services API» (рисунок 3.9).

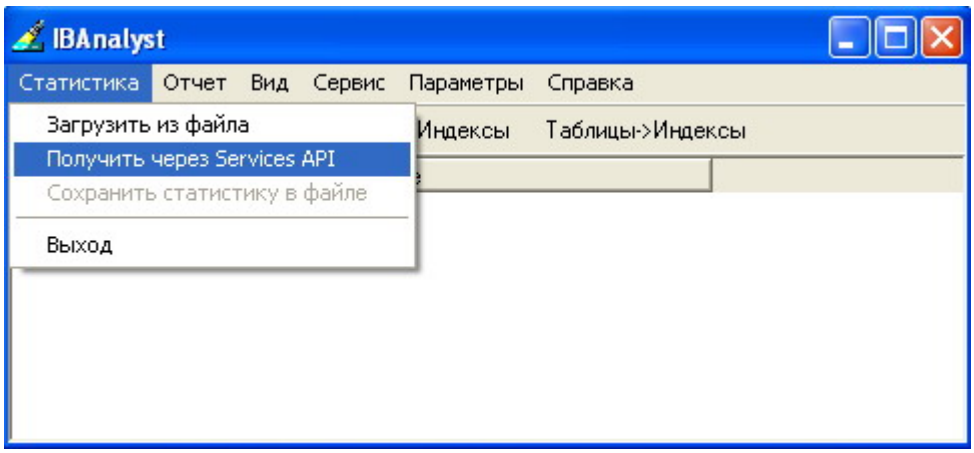

Рисунок 3.9 - Подключение IBAnalyst к базе данных

На экране появится диалоговое окно подключения к базе данных (рисунок 3.10).

В окне необходимо указать следующие параметры:

– Server Name - имя компьютера сервера. (т.к. база данных и IBAnalyst установлены на одном компьютере, то указываем локальное подключение localhost);

– Database Name – путь и имя файла базы данных;

– Protocol – протокол подключения к базе данных (TCP установлено по умолчанию);

– User – имя учетной записи администратора базы данных (по умолчанию SYSDBA);

– Password – пароль учетной записи администратора базы данных (по умолчанию masterkey).

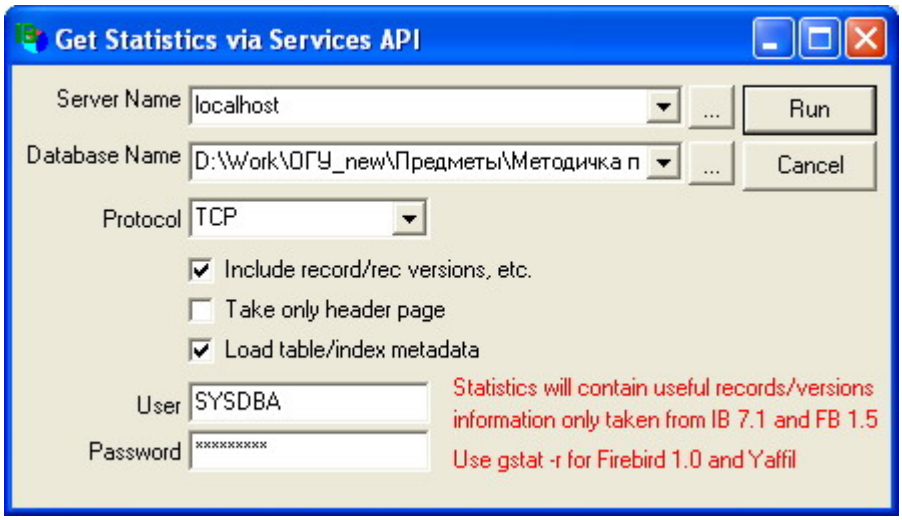

Рисунок 3.10 - Диалоговое окно подключения к базе данных

После указания всех параметров нажимаем на кнопку «Run». На экране появится окно информации о базе данных (рисунок 3.11).

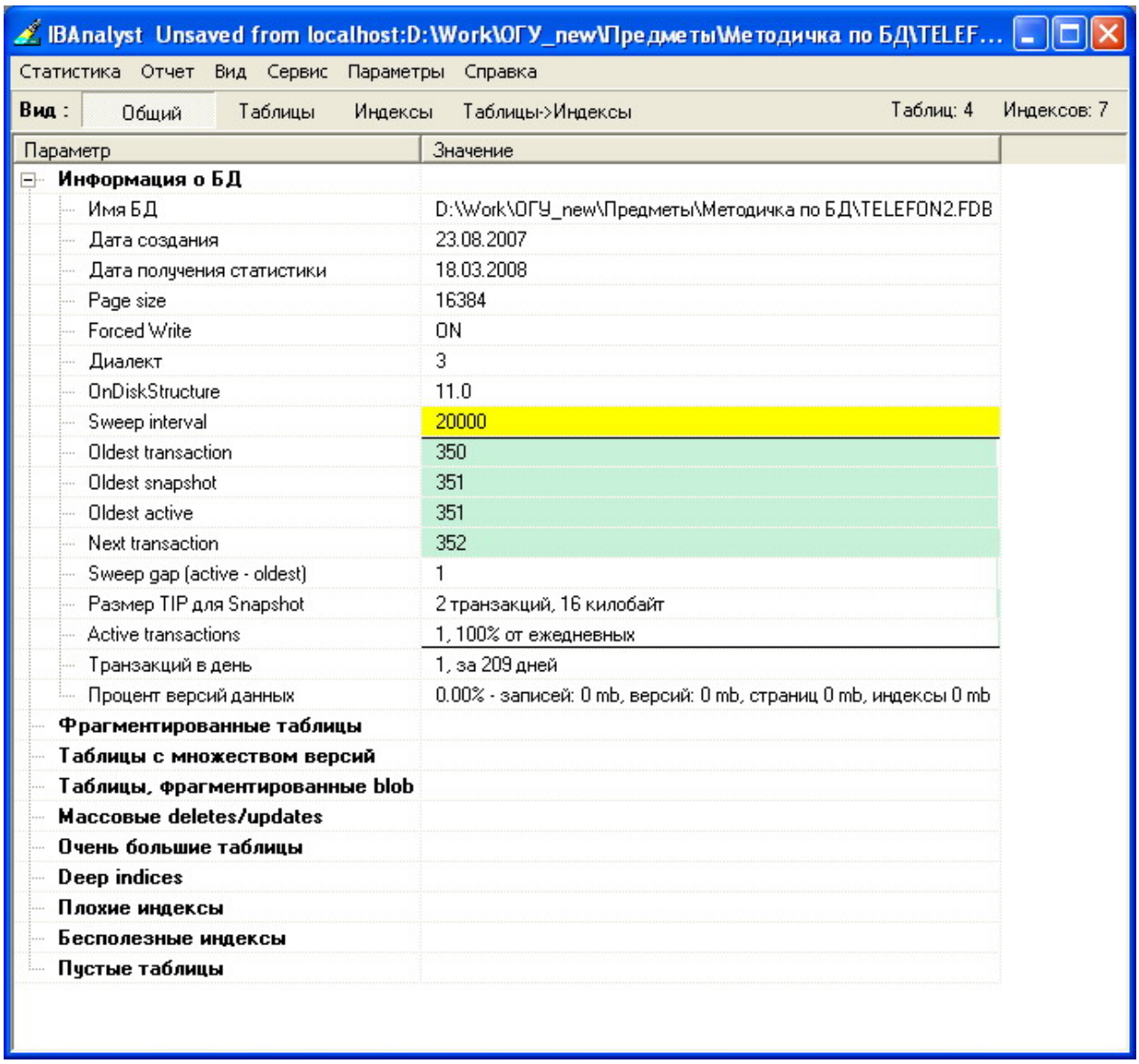

Рисунок 3.11 - Окно информации о базе данных

Теперь все необходимое программное обеспечение установлено, настроено и готово к работе.

3.2 Проектирование структуры базы данных в IBExpert

При разработке современных баз данных используются специальные CASE средства, позволяющие визуально проектировать структуру будущей базы данных. В среде IBExpert интегрировано CASE средство «Дизайнер БД». Используя возможности «Дизайнер БД», спроектируем базу данных по тематике «каталог сотовых телефонов».

База данных будет состоять из четырех таблиц. Таблица «Firma» содержит информацию о фирме производителе сотового телефона, например: Nokia, Samsung. В таблице «Model» будет хранить информацию о модели сотового телефона, например: 6233, N73. Таблица «Functions» используется для

хранения общих функций сотовых телефонов, например: mp3 на звонок, фонарик. И последняя таблица «Model_Functions» хранит соответствие между моделью телефона и его функциями. Er-диаграмма, проектируемой базы данных, изображена на рисунке 3.12.

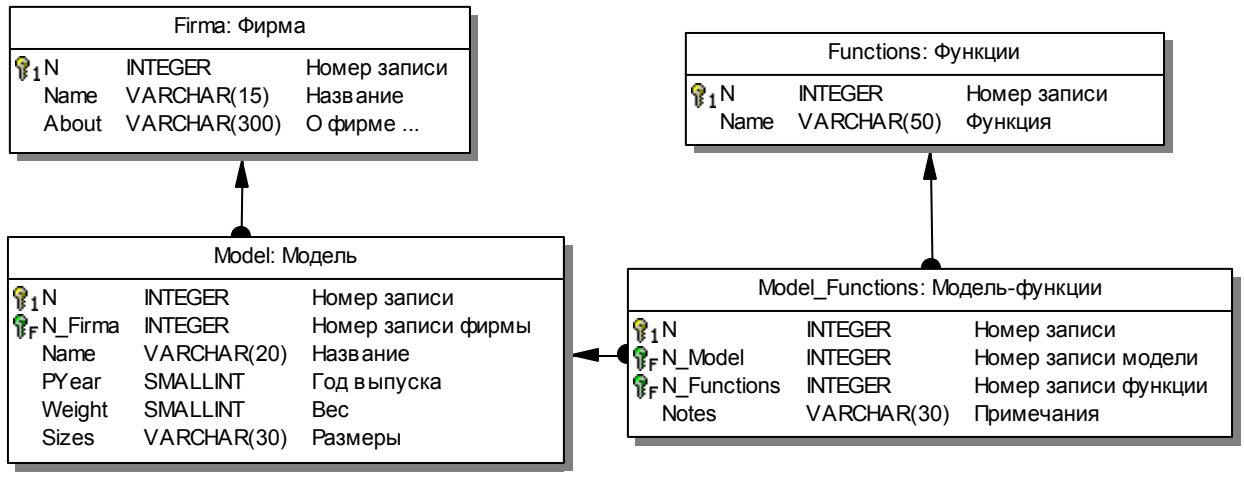

Рисунок 3.12 - ER-диаграмма базы данных «Сотовые телефоны»

Запуск «Дизайнера БД» осуществляется через меню IBExpert «Инструменты» → «Дизайнер БД». На рисунке 3.13 изображен «Дизайнер БД».

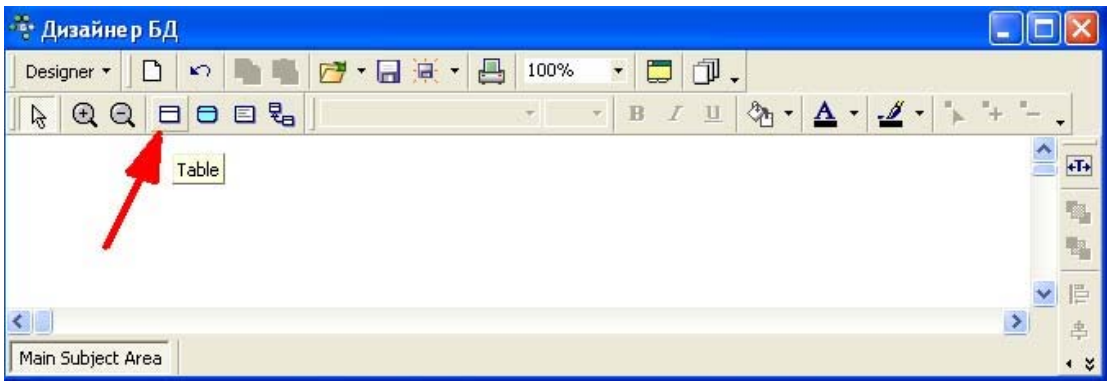

Рисунок 3.13 - «Дизайнер БД»

На рисунке 3.13 стрелочкой показана пиктограмма для создания таблицы. Этой пиктограммой мы будем пользоваться наиболее часто.

Создадим таблицу «Firma». Для этого необходимо нажать на пиктограмму «table», а затем нажать мышкой на рабочую область. На рабочей области появится изображение пустой таблицы с названием table1. После двойного нажатия на изображение таблицы появляется окно свойств таблицы (рисунок 3.14), в котором задаются название таблицы, поля и типы таблицы и т.д.

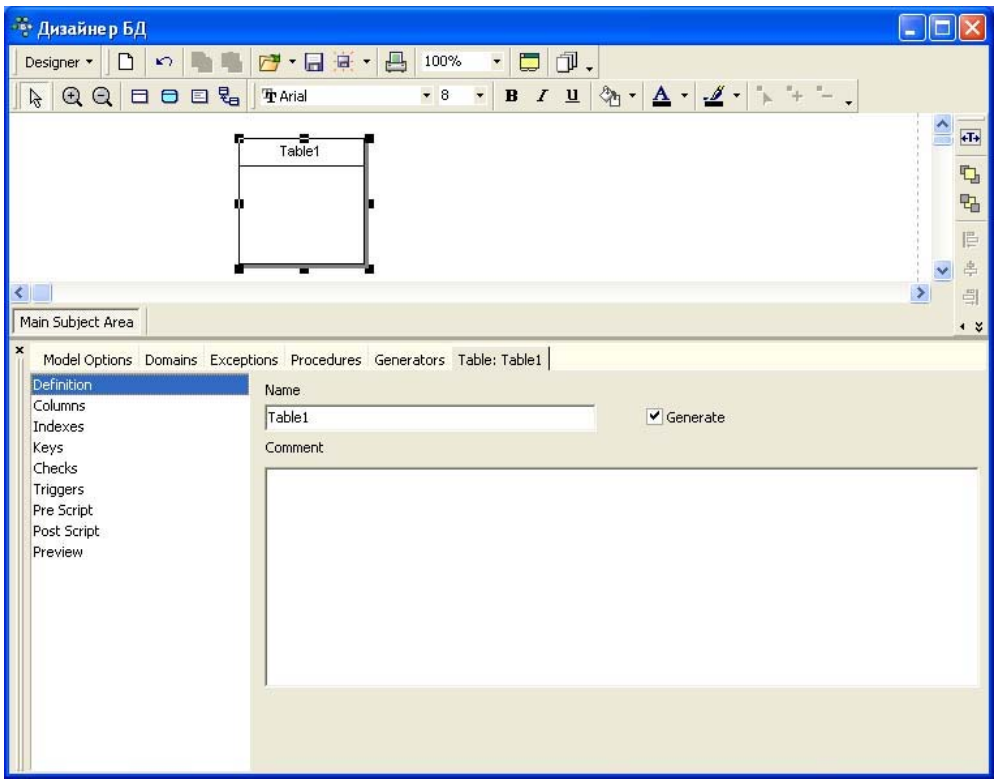

Рисунок 3.14 - Окно свойств таблицы

В поле «Name» задаем название таблицы – «Firma», в поле «Comment» вписываем комментарий к названию таблицы – «Фирма». Когда имя таблицы задано, создадим поля таблицы. Для этого необходимо перейти к диалогу создания полей, щелкнув мышкой на «Columns». На экране отобразится окно, которое необходимо заполнить, как показано на рисунке 3.15.

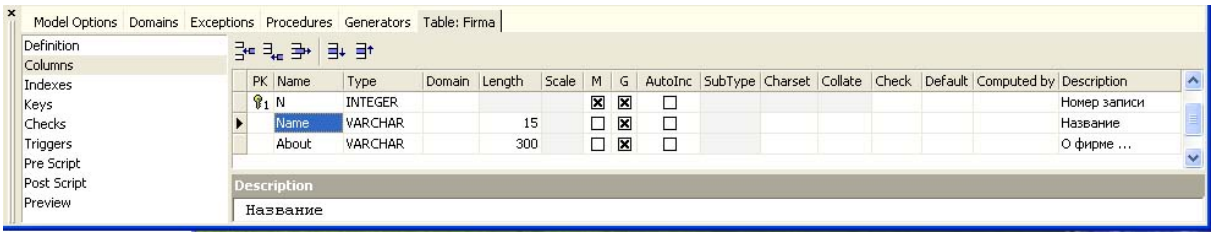

Рисунок 3.15 - Диалог создания полей таблицы

Первое поле таблицы «Firma» - «Номер записи». Это поле содержит уникальное значение и является ключом таблицы (Primary key). Для того, чтобы определить поле «N» как ключ таблицы, необходимо выполнить двойное нажатие мышки в поле «PK». В результате проделанной операции в поле появится изображение ключа. В поле «Description» для каждого поля указывается комментарий.

Для полного отображения всех свойств таблицы необходимо настроить раздел «Model Options» (рисунок 3.16).

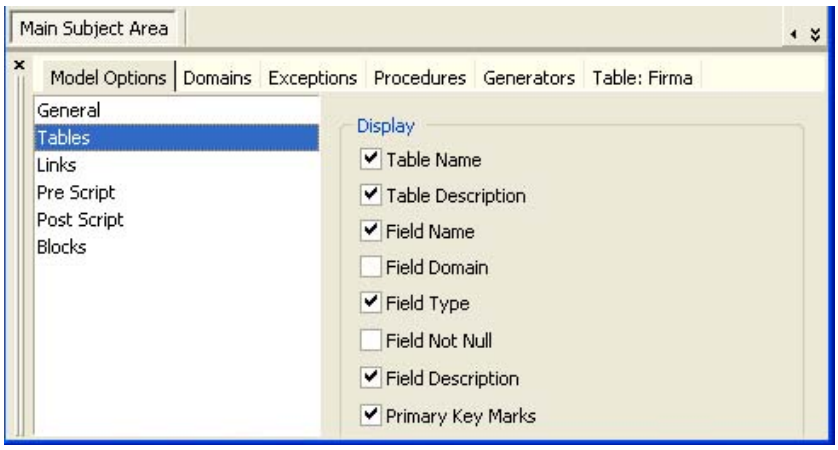

Рисунок 3.16 - Настройка «Model Options»

Используя алгоритм, описанный выше, создадим таблицу «Model» в дизайнере баз данных. Таблица «Model» является подчиненной таблице «Firma». Для реализации этой связи в таблице «Model» создается специальное поле «N_Firma» и внешний ключ (Foreign Key) к таблице «Firma». Эта операция осуществляется в диалоговом окне «Keys» таблицы «Model». Добавляется новый ключ «FK_Firma_Model» с параметрами, изображенными на рисунке 3.17.

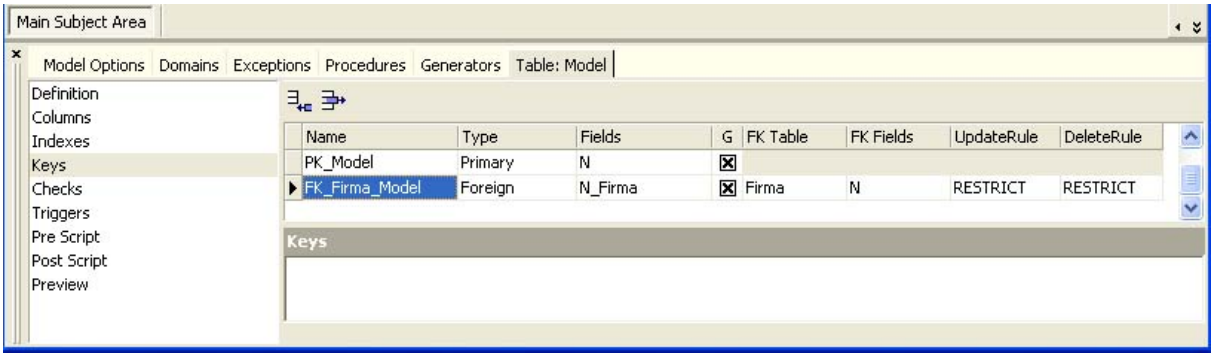

Рисунок 3.17 - Добавление ключа «FK_Firma_Model»

В результате проделанной операции между таблицами «Firma» и «Model» образовалась связь один ко многим. Т.е. одна запись таблицы «Firma» может быть связана с несколькими записями в таблице «Model». Так например, модели телефонов «6233» и «N73» произведены фирмой «Nokia». В дизайнер баз данных отображаются две связанные таблицы, как это показано на рисунке 3.18.

По аналогичному алгоритму создаются таблицы «Functions», «Model_Functions» и связи «FK_Function_MF»,«FK_Model_MF» для таблиц «Functions»-«Model_Functions» и «Model»-«Model_Functions» соответственно.

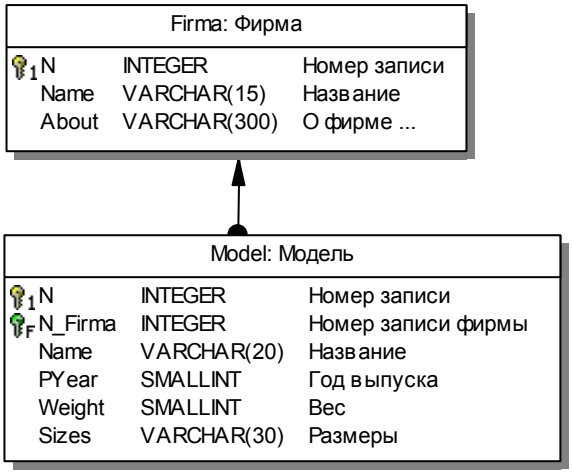

Рисунок 3.18 - Связанные таблицы «Firma» и «Model»

Структура проектируемой базы данных сформирована и теперь ее можно сохранить в файл (рисунок 3.19). В дизайнере баз данных существует возможность экспорта схемы базы данных в emf-файл (рисунок 3.19). Эта функция пригодится при создании документации на базу данных.

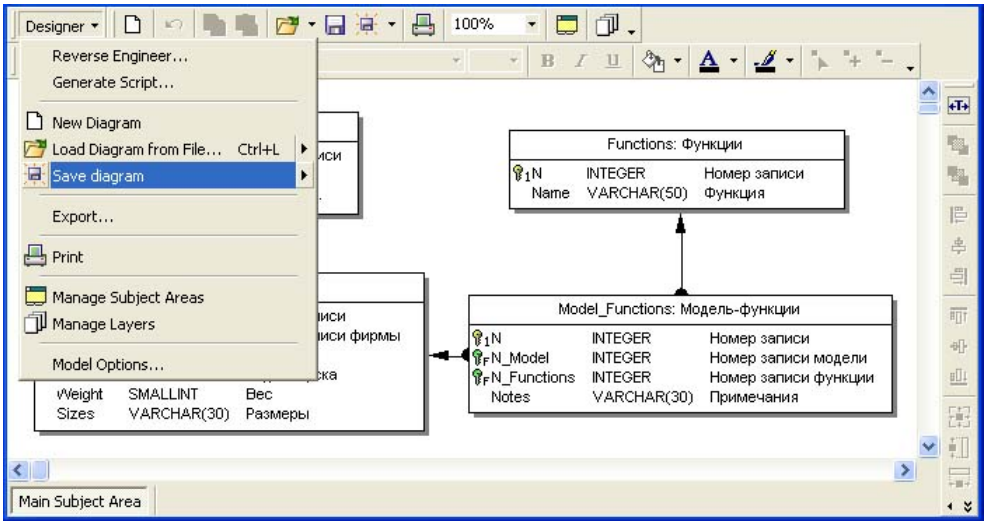

Рисунок 3.19 - Возможности сохранения и экспорта

После сохранения дизайнер баз данных необходимо закрыть.

3.3 Создание базы данных

В качестве примера рассмотрим создание базы данных с использованием IBExpert, для этого необходимо выполнить следующие действия. Через меню IBExpert выполните «База данных» → «Создать базу …». Появившееся окно необходимо заполнить, как показано на рисунке 3.20.

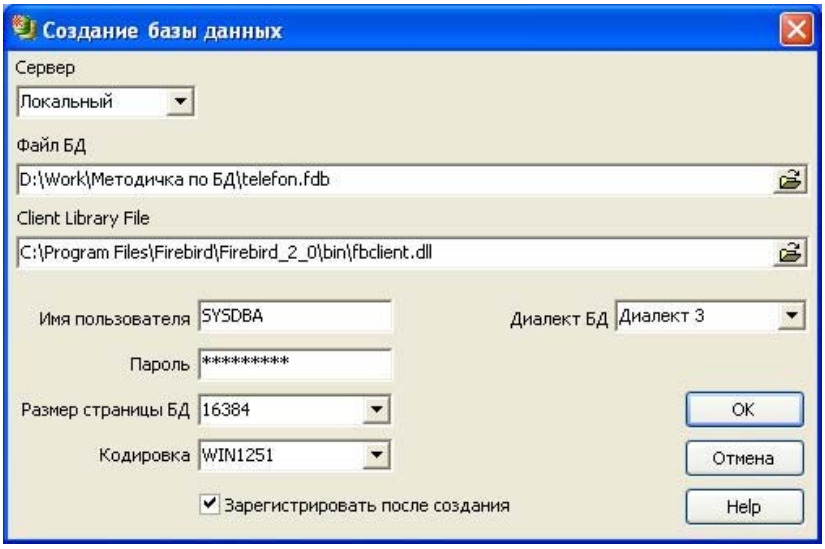

Рисунок 3.20 - Диалог создания базы данных

В окне «Создание базы данных» необходимо определить следующие параметры:

– сервер – «Локальный»;

– файл БД – указать путь и название создаваемой базы данных;

– имя пользователя - «SYSDBA»(«SYSDBA» - учетная запись администратора базы данных);

– пароль – «masterkey»(«masterkey» - пароль для учетной записи администратора по умолчанию).

Нажимаем кнопку «OK». На экране появится окно регистрации базы данных (рисунок 3.21).

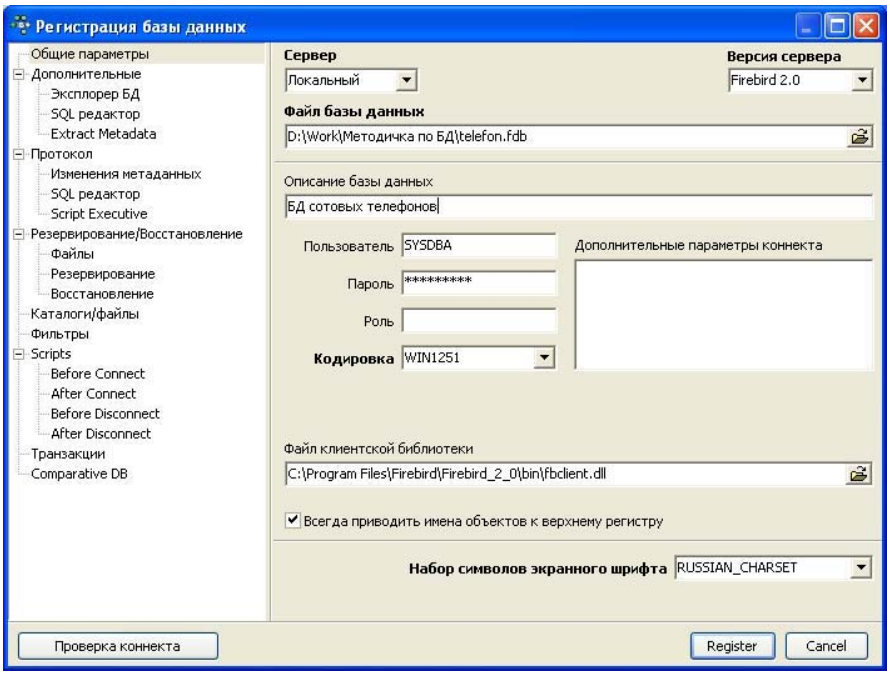

Рисунок 3.21 - Окно регистрации базы данных

В этом окне желательно заполнить поле «Описание базы данных». После этого нажимаем кнопку «Register». Теперь в окне «Database Explorer» появится база данных с названием «БД сотовых телефонов».

Созданная база данных пустая. Ее необходимо заполнить связанными таблицами, которые разработаны в дизайнере баз данных. Для этого запускаем дизайнер баз данных и загружаем разработанную ранее структуру «Designer» → «Load Diagram from File». Затем выполняем генерацию скрипта «Designer» → «Generate Script». Появляется окно генератора скриптов (рисунок 3.22).

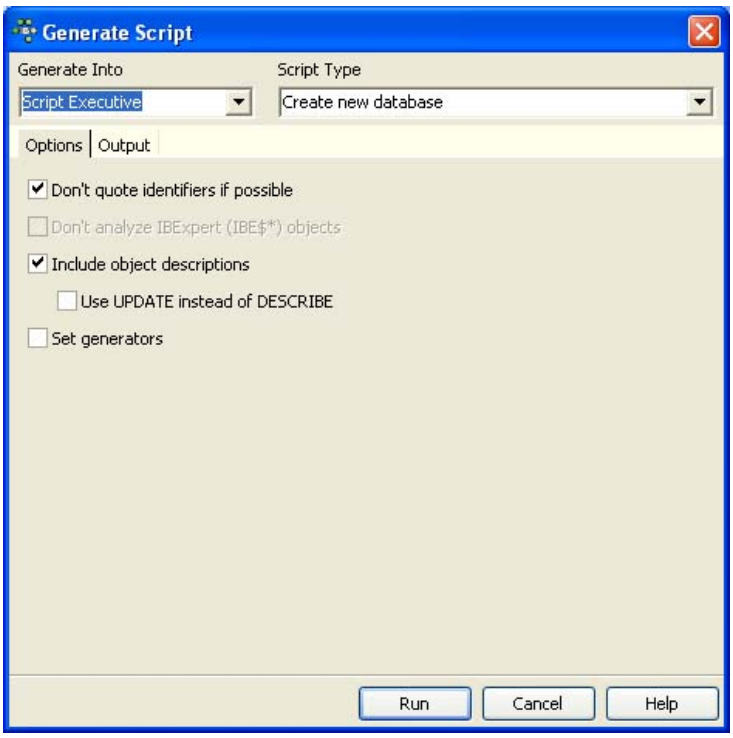

Рисунок 3.22 - Окно генератора скриптов

В этом окне необходимо в поле «Generate Info» выставить значение «Script Executive», т.е. скрипт будет автоматически сформирован в программе выполнения скриптов («Script Executive»). Нажимаем кнопку «Run».

Далее в появившемся окне необходимо выбрать в качестве активной созданную нами ранее базу данных (рисунок 3.23).

После выбора активной базы данных необходимо выбрать появившийся новый пункт «использовать текущее подключение» (рисунок 3.24). Далее нужно запустить скрипт на выполнение, нажав на кнопку, отмеченную на рисунке 3.24 стрелкой.

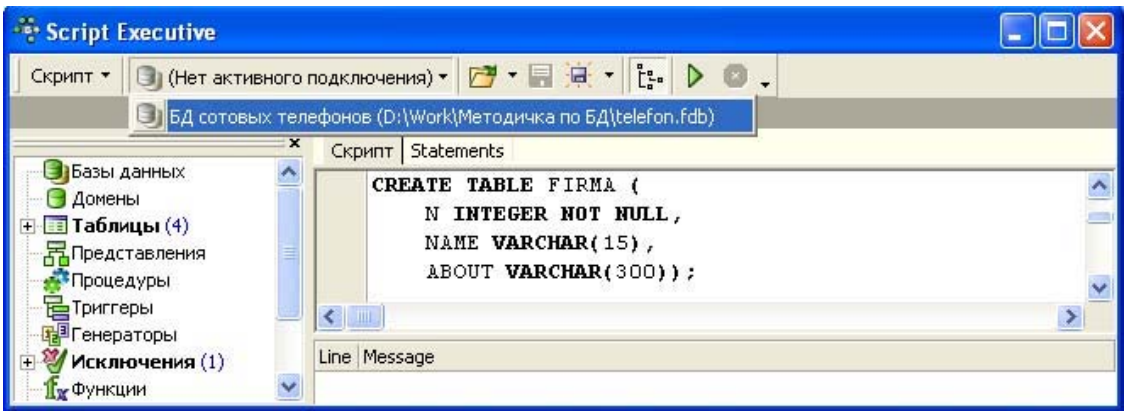

Рисунок 3.23 - Выбор активной базы данных

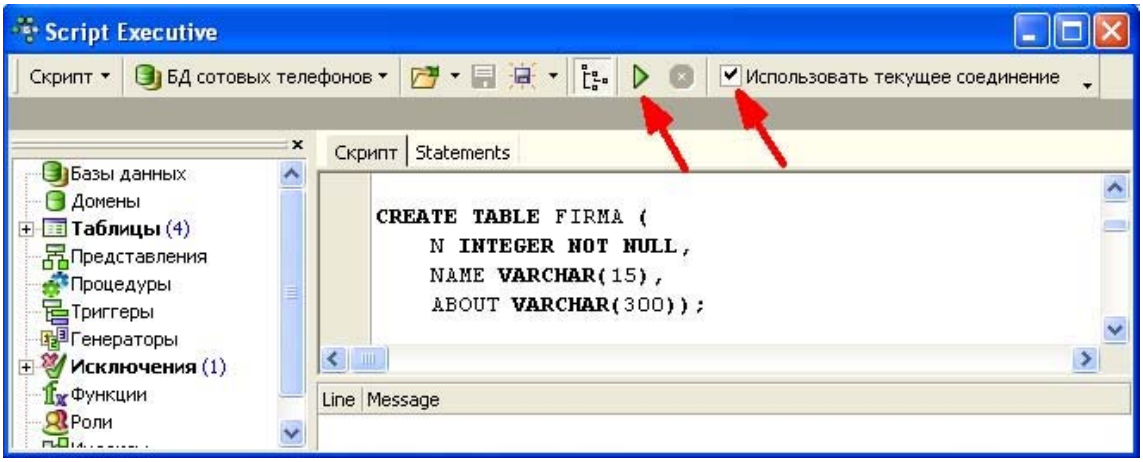

Рисунок 3.24 - Запуск скрипта на выполнение

После выполнения скрипта в базе данных появятся таблицы.

В каждой таблице присутствует поле с именем «N» - номер записи. Используя среду IBExpert, реализуем автоматическое заполнение этих полей. Для этого выберем таблицу в «Database Explorer» и включим режим редактирования структуры таблицы. На рисунке 3.25 кнопка включения режима редактирования показана стрелкой. Для поля «N» необходимо включить свойство «AutoInc». Это свойство показано на рисунке 3.25 стрелкой.

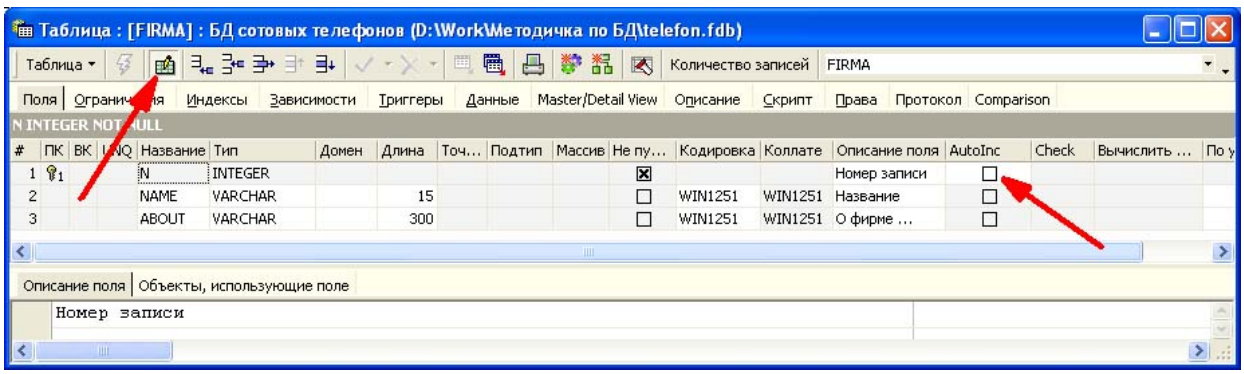

Рисунок 3.25 - Редактирование структуры таблицы «Firma»

В появившемся окне нужно отметить галочками поля «создать генератор», «создать триггер» (рисунок 3.26). Можно также, по желанию, скорректировать имя генератора.

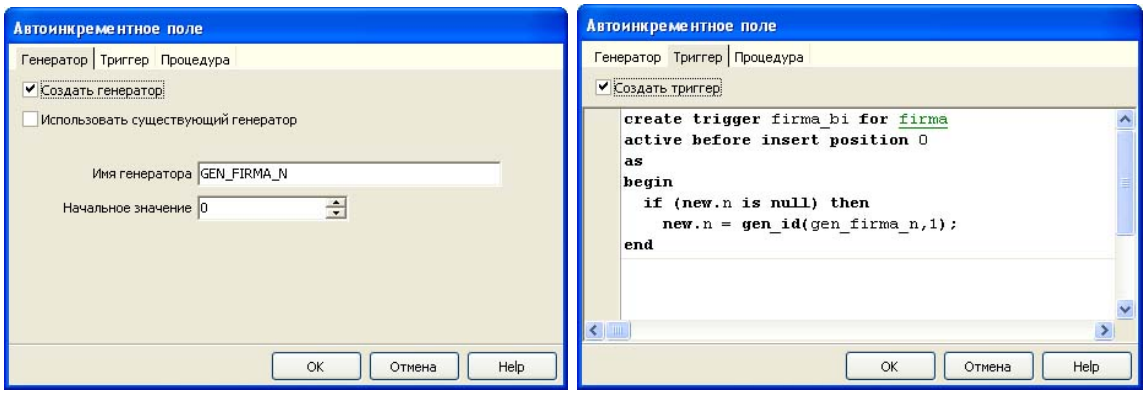

Рисунок 3.26 - Создание автоинкрементного поля

Далее нажимаем кнопку «OK». Выполняем компиляцию, используя комбинацию клавиш «Ctrl+F9», и нажимаем в появившемся окне кнопку «Commit».

После выполненных операций поле «N» таблицы «Firma» будет заполняться автоматически.

Аналогичные действия необходимо произвести над остальными таблицами.

Теперь база данные сформирована, и можно приступить к ее заполнению.

3.4 Заполнение базы данных

Заполнение базы данных записями из среды IBExpert может быть осуществлено тремя различными способами. Пополнять информацией таблицы можно через «Database Explorer», выбрав в пополняемой таблице закладку данные, как это показано на рисунке 3.27.

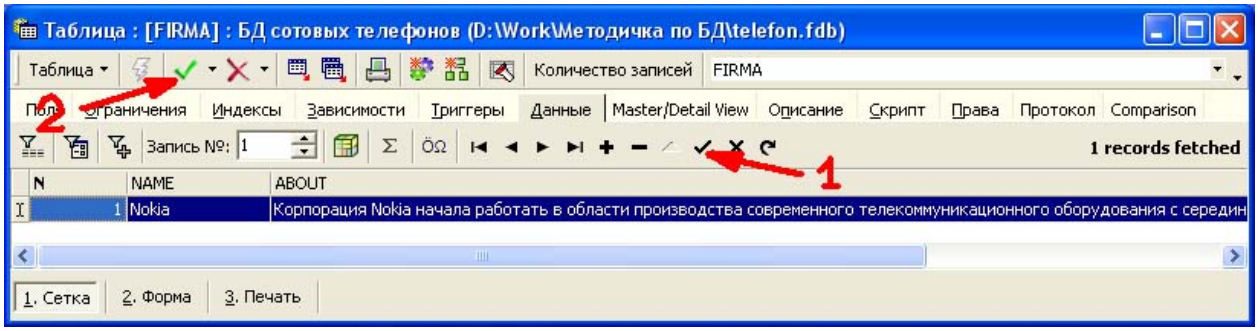

Рисунок 3.27 - Добавление новой записи в таблицу «Firma»

Заполняются поля «NAME» и «ABOUT», поле «N» заполнится автоматически после нажатия на кнопку, которая показана на рисунке 3.28

стрелкой № 1. Для подтверждения транзакции требуется нажать на кнопку, отмеченную на рисунке 3.28 стрелкой №2. После выполненных действий в таблице «Firma» сохранится запись о фирме «Nokia».

Два оставшихся метода внесения записей в таблицу основываются на SQL-операторе insert. SQL-операторы в среде IBExpert выполняются в SQL-редакторе. Для его запуска необходимо нажать на кнопку, указанную стрелкой на рисунке 3.28.

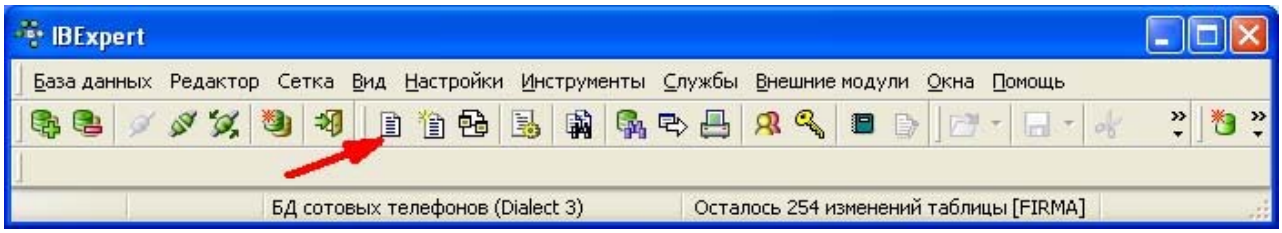

Рисунок 3.28 - Кнопка запуска SQL-редактора

Затем вводится и выполнятся текст SQL-оператора insert. Выполнение осуществляется путем нажатия на кнопку, показанную на рисунке 3.29 стрелкой №1. Подтверждение транзакции выполняется кнопкой, показанной на рисунке 3.29 стрелкой №2.

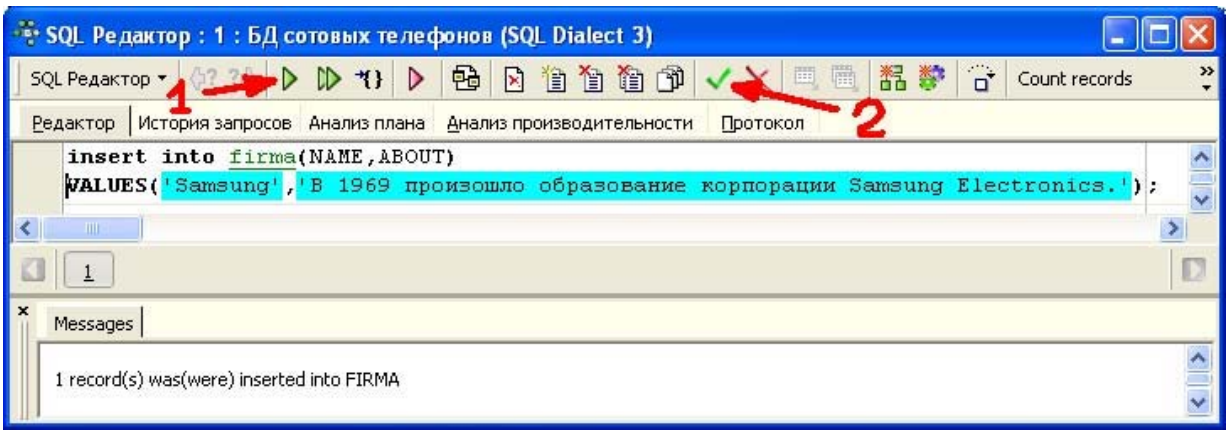

Рисунок 3.29 - Вставка записи в таблицу «Firma»

В таблице «Firma» теперь хранятся записи о двух фирмах.

При необходимости выполнения нескольких SQL-операторов insert используется редактор скриптов («Script Executive»). Работа с ним осуществляется аналогично. Основные принципы его использования были рассмотрены выше в главе 3.3.

При использовании второго варианта внесения новых записей в таблицу удобно пользоваться параметрами. Для этого необходимо ввести вместо значений полей в оператор insert названия параметров. Название параметра начинается с символа «:». Пример использования параметров изображен на рисунке 3.30.

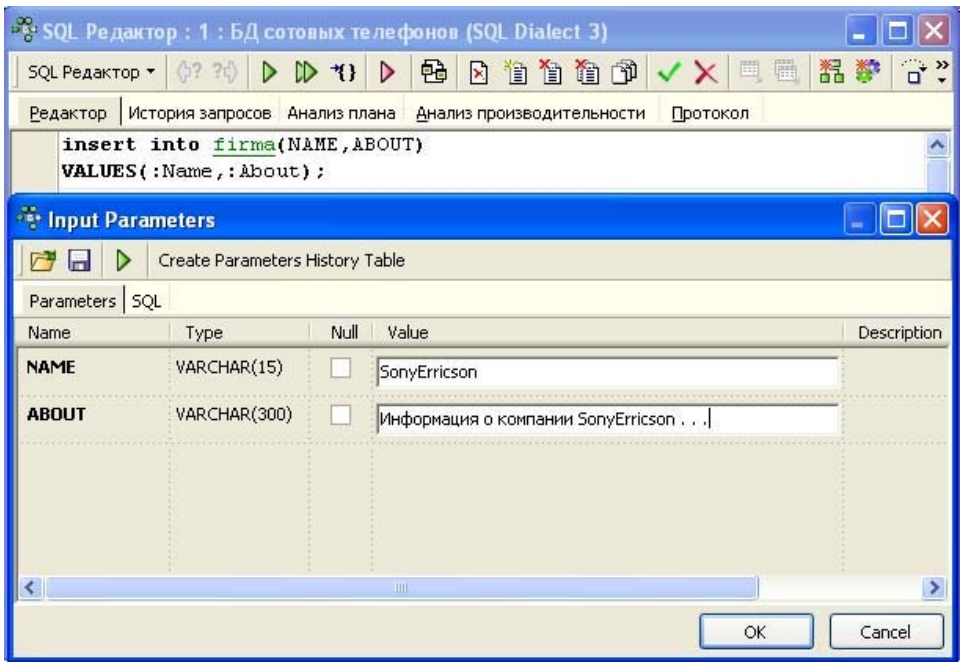

Рисунок 3.30 - Использование параметров в SQL-запросе

Рассмотрим пример заполнения таблицы «Model». Добавим в таблицу «Model» телефон 6233, произведенный фирмой Nokia. Для этого необходимо, используя «Database Explorer», выбрать таблицу «Model» и перейти на закладку данные. Затем включаем режим вставки записей и заполняем все поля за исключением «N» и «N_Firma». Поле «N» заполнится автоматически, а значение поля «N_Firma» надо выбрать из списка значений поля «N» таблицы «Firma»(рисунок 3.31).

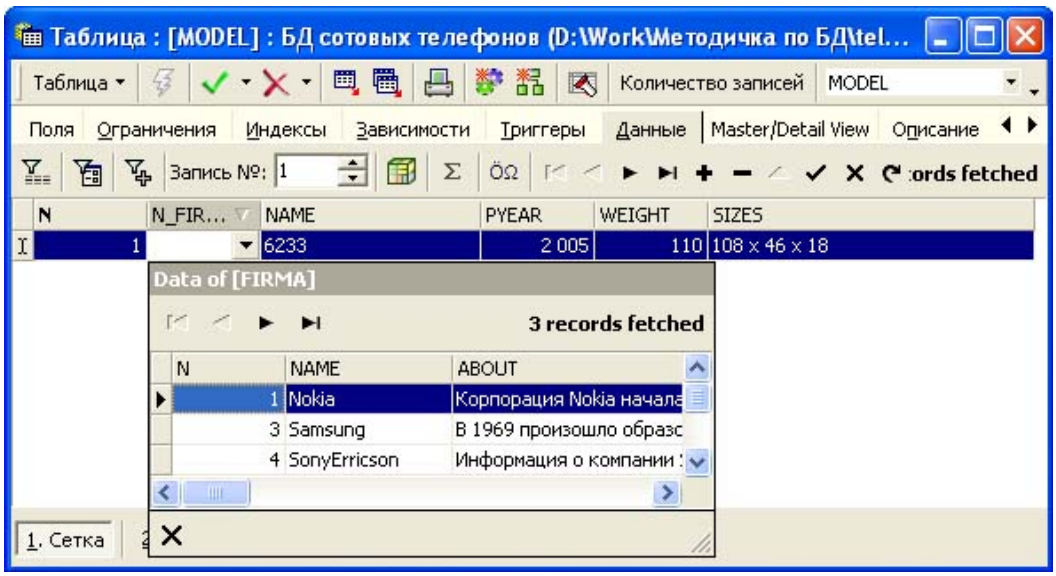

Рисунок 3.31 - Добавление записи в таблицу «Model»

После заполнения всех полей сохраняем изменения и выполняем подтверждение транзакции.

Аналогично добавим в таблицу «Model» модели телефонов из таблицы 3.1.

Таблица 3.1 - Модели телефонов

Фирма производитель	Модель	Год выпуска	Вес	Размеры
Nokia	5500 Sport	2006	103	107 x 45 x 18
Samsung	SGH-E900	2006	93	93 x 45 x 16.5
Samsung	SGH-E570	2006	81	86 x 45 x 24
SonyErricson	T290	2004	73	101 x 44 x 19

В таблицу «Functions» вставляем записи, описывающие функции телефонов. Информация для таблицы «Functions» представлена в таблице 3.2.

Таблица 3.2 - Функции телефонов

Функция	Модели
Камера 1.3 МП	Samsung SGH-E570
Камера 2 МП	Samsung SGH-E900, Nokia 6233, Nokia 5500 Sport
Пылевлагозащищенный корпус	Nokia 5500 Sport
Будильник	Все модели

В таблицы приведен неполный перечень функций рассматриваемых телефонов. Перечислены только функции, необходимые для примеров, которые будут рассмотрены позднее.

Пример заполнения таблицы «Model_Functions» представлен на рисунке 3.32.

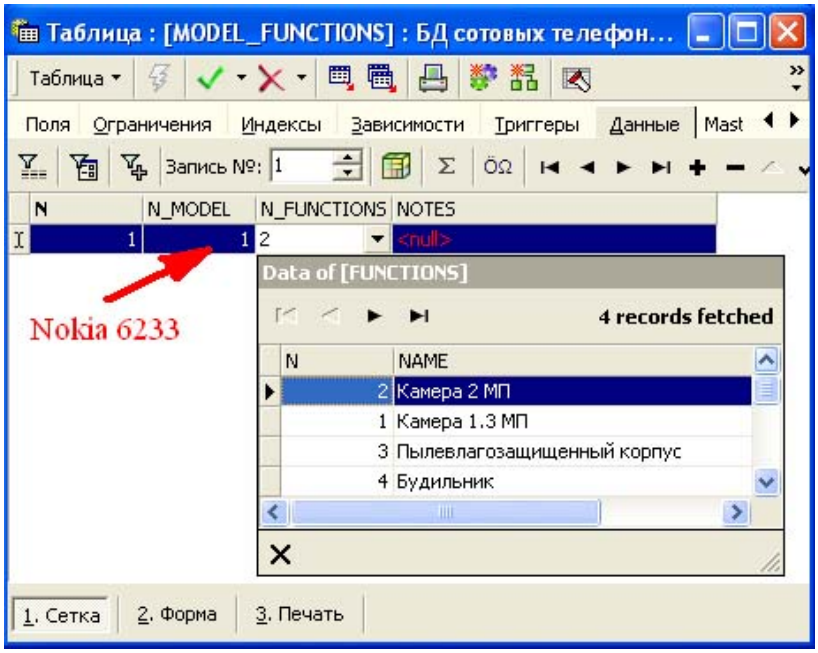

Рисунок 3.32 - Пример заполнения таблицы «Model_Functions»

3.5 Работа с данными

3.5.1 Выполнение запросов к базе данных

Теперь, когда таблицы базы данных заполнены записями, рассмотрим примеры запросов. Для выполнения запросов используем SQL-редактор.

Вывести список моделей телефонов, вес которых не превышает 100 г.

Ниже приведен текст запроса.

```
select f.name,m.name
from firma f, model m
where f.n=m.n_firma and m.weight<=100;
```

Результат выполнения запроса представлен на рисунке 3.33.

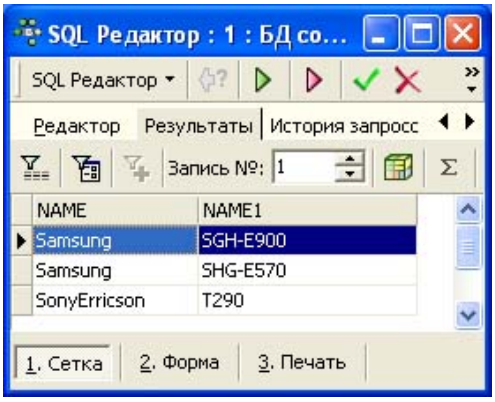

Рисунок 3.33 - Результат выполнения запроса

При выборе сотового телефона покупатель выбирает модель телефона по ряду функций, которые его интересуют. Если покупатель хочет выбрать сотовый телефон в пылевлагозащищенном корпусе с камерой 2МП, то поиск телефонов с такими характеристиками выполняется следующим запросом.

```
select f.name as "FIRMA",
       m.name as "MODEL"
from firma f,  model m, model_functions mf
where f.n=m.n_firma and m.n=mf.n_model
      and mf.n_functions in (2,3)
group by 1,2
Having count(mf.n_functions) = 2;
```

В запросе функция пылевлагозащищенного корпуса – «3», камера 2МП – «2».

В результате выполнения запроса будут выведены телефоны, удовлетворяющие условиям поиска (рисунок 3.34). Из моделей, внесенных в базу данных, под эти критерии попадает модель 5500 Sport фирмы Nokia.

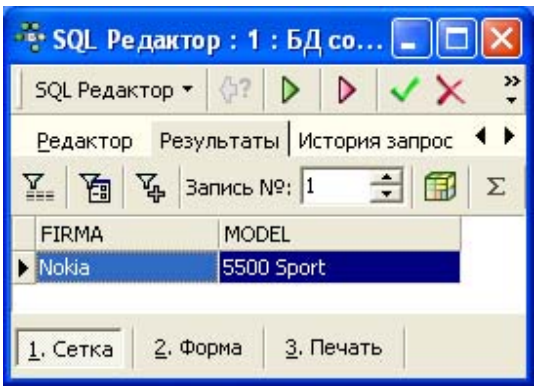

Рисунок 3.34 - Результат выполнения запроса

3.5.2 Использование триггеров

Рассмотрим один из возможных варрантов использования триггеров на примере удаления записей из связанных таблиц.

Удаление осуществляется с помощью команды DELETE либо через среду IBExpert. СУБД FireBird позволяет удаление записи, на которую не ссылается ни одна запись из подчиненных таблиц. Ситуация осложняется; если мы пытаемся удалить запись, с которой есть связанные записи в подчиненных таблицах. Например, при попытке удалить модель T290 СУБД выдаст следующую ошибку (рисунок 3.35).

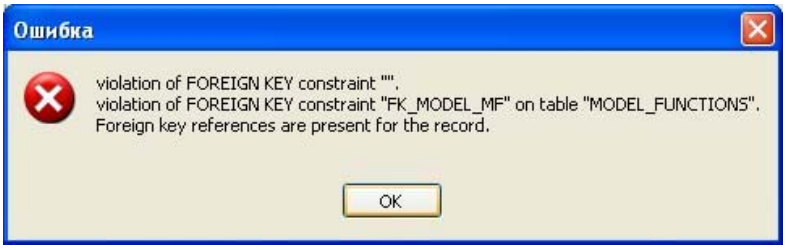

Рисунок 3.35 - Ошибка при удалении связанной записи

Ошибка возникает из-за того, что в таблице «Model_Functions» есть записи, связанные с записью T290 в таблице «Model». В решении этой проблемы нам помогут триггеры. Триггер - это предварительно определённое действие или последовательность действий, автоматически осуществляемых при выполнении операции обновления, добавления или удаления данных. В нашем случае нам необходим триггер, который будет срабатывать перед удалением записи из таблицы «Model». Перед удалением триггер будет автоматически удалять все записи из таблицы «Model_Functions», ссылающиеся на удаляемую запись.

Для добавления триггера необходимо выбрать таблицу «Model» в «Database Explorer» и перейти на вкладку «Триггеры». Как мы видим, к этой таблице уже привязан один триггер на действие «Перед вставкой»(это триггер, осуществляющий автоматическое заполнения поля «N»). Нажимаем пра-

вой кнопкой на надписи «Перед удалением» и выбираем «новый триггер» (рисунок 3.36).

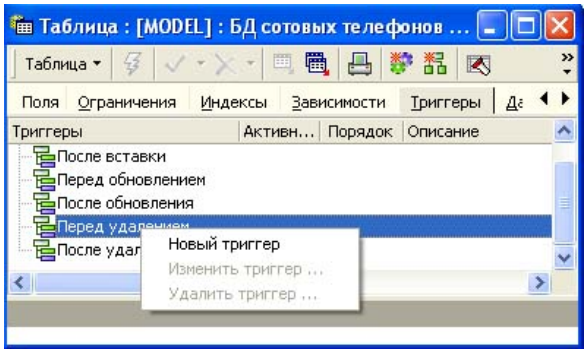

Рисунок 3.36 - Создание нового триггера

Вводим информацию в диалоговое окно, как показано на рисунке 3.37. В поле ввода «Порядок» указывается порядок выполнения триггеров. На одно событие возможна установка нескольких триггеров. В этом случае можно определять порядок их выполнения. При «0» порядке триггер будет выполнен первым.

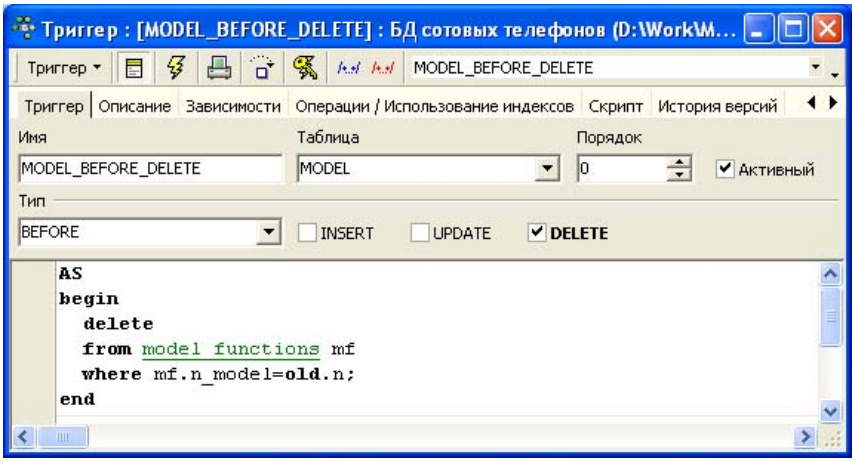

Рисунок 3.37 - Триггер «MODEL_BEFORE_DELETE»

Компилируем триггер. Теперь можно повторить попытку удаления модели T290.

Триггеры имеют очень широкий спектр применения, в нашем примере было рассмотрено, как с помощью триггеров осуществляется каскадное удаление записей.

Рассмотрим следующий пример. Необходимо скорректировать поле «Notes» таблицы «Model_Functions», т.е. заменить «по дням недели» на «продвинутый будильник». Ниже приведена команда **update**, реализующая требуемую замену.

```
update model_functions mf
 set mf.notes = 'продвинутый будильник'
```

```
where mf.notes like '%дням%';
```

3.5.3 Использование хранимых процедур

При обработке информации базы данных удобно использовать хранимые процедуры. FireBird предоставляет возможность создавать хранимые процедуры, возвращающие записи подобно таблицам. И вызывать их впоследствии через select ... from <имя_процедуры> соответствующим образом. Рассмотрим пример. Предположим необходимо вывести количество моделей сотовых телефонов по каждой из фирм. Такую информацию из базы данных можно получить следующим запросом.

```
select f.name as "FIRMA",
       count(m.name) as "MODEL_COUNT"
from firma f,  model m
where f.n=m.n_firma
group by 1;
```

Оформим этот запрос виде процедуры FIRMA_MODEL_COUNT, которая будет выводить информацию по количеству сотовых телефонов. Для этого необходимо в окне «DataBase Explorer» среды IBExpert правой кнопкой мыши выбрать пункт «Процедуры» как это показано на рисунке 3.38.

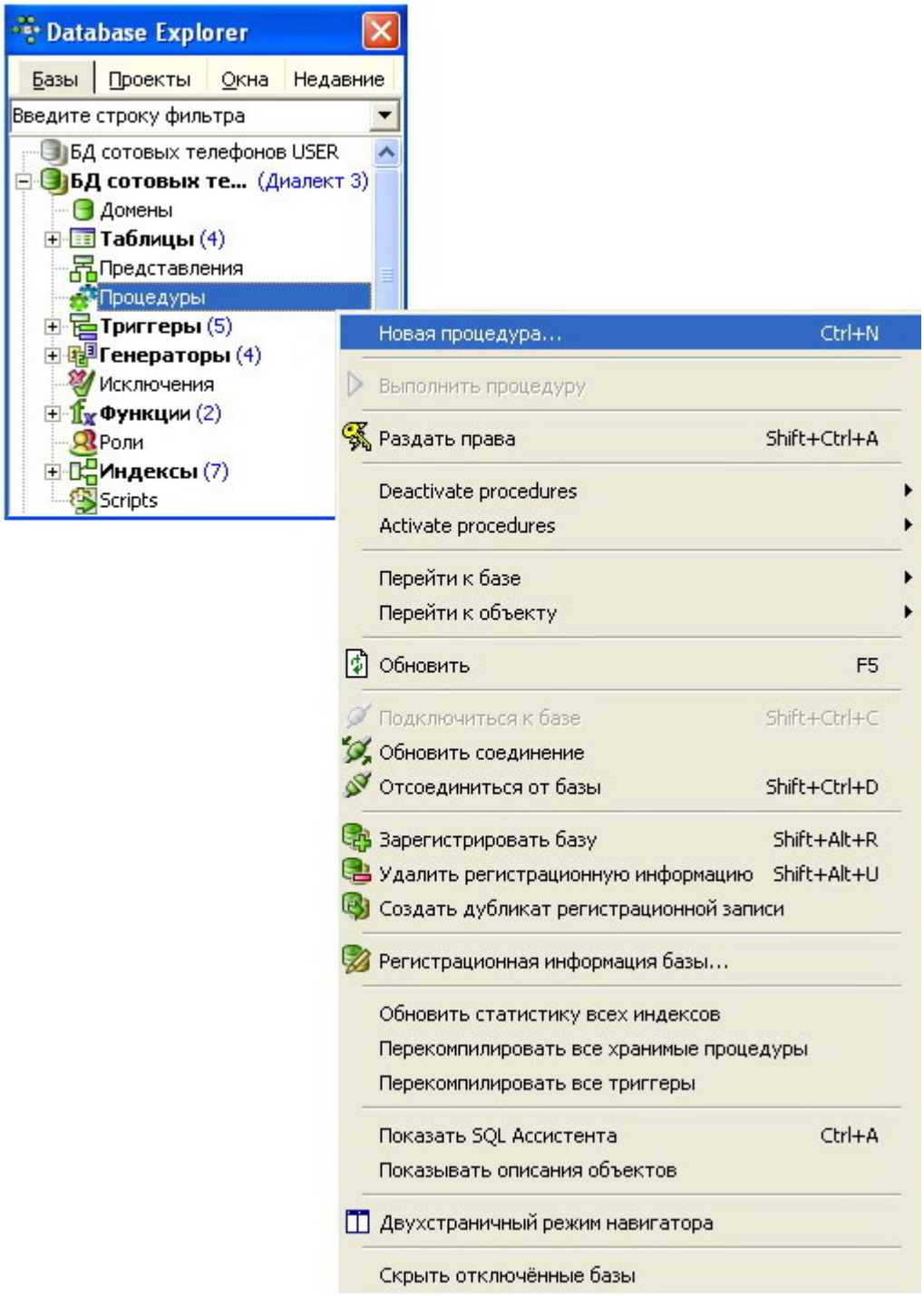

Рисунок 3.38 - Запуск диалога создания хранимой процедуры

В появившемся меню выбираем пункт «Новая процедура». На экране появится окно диалога создания хранимой процедуры (рисунок 3.39).

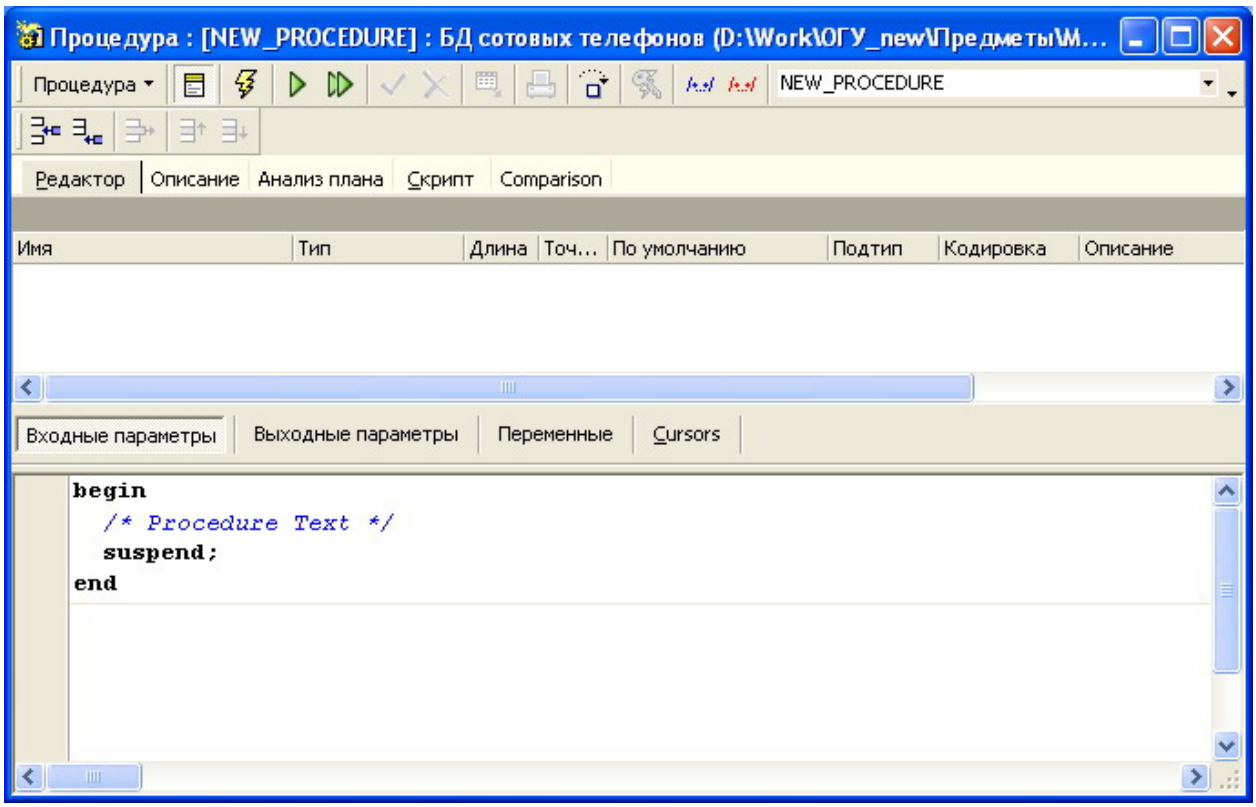

Рисунок 3.39 - Диалог создания хранимой процедуры

Необходимо указать название создаваемой хранимой процедуры и определить ее переменные. Название создаваемой процедуры определяется в верхнем правом поле ввода диалогового окна. На рисунке 3.40 это поле выделено стрелкой. В нашем примере у хранимой процедуры присутствуют только выходные параметры. Для определения выходных параметров хранимой процедуры нажимаем на кнопку «Выходные параметры». Затем, используя кнопку, выделенную на рисунке 3.40 второй стрелкой, добавляем выходные параметры:

- название фирмы (FIRM_NAME) типа VARCHAR(15);
- количество моделей сотовых телефонов (MODEL_COUNT) типа INTEGER.

Текст хранимой процедуры вводится в поле диалогового окна (рисунок 3.40). В тексте хранимой процедуры используется оператор for select … do. Этот оператор выполняется для каждой записи, получаемой запросом.

Рассмотрим работу процедуры подробнее. При выполнении хранимой процедуры происходит следующее: выбирается фирма, по выбранной фирме выполняется подсчет количества моделей сотовых телефонов и результат передается в выходные переменные into :FIRM_NAME, :MODEL_COUNT. Затем выполняется оператор suspend, который выдает значение выходных переменных хранимой процедуры.

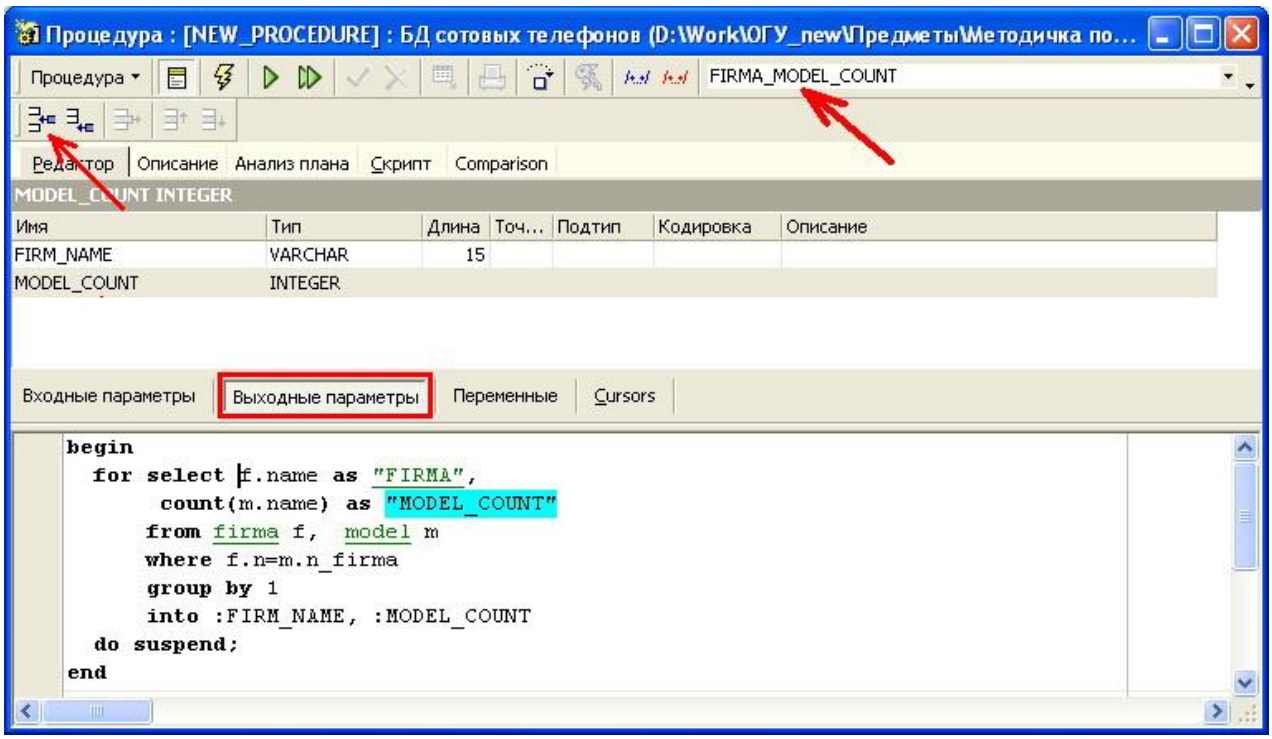

Рисунок 3.40 - Создание хранимой процедуры FIRMA_MODEL_COUNT

Выполним созданную хранимую процедуру. IBExpert выведет на экран информационное сообщение об изменении названия хранимой процедуры (рисунок 3.41). Нажимаем кнопку «Да».

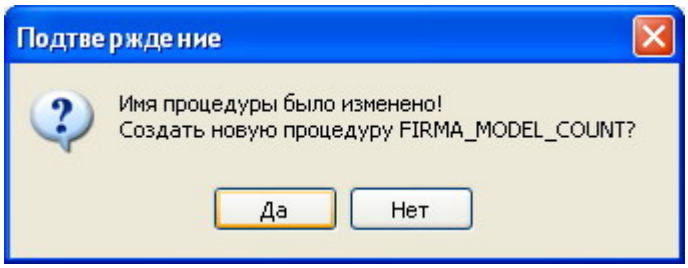

Рисунок 3.41 - Информационное сообщение среды IBExpert

Далее на экране появится диалоговое окно отчета выполнения операции (рисунок 3.42). Если создаваемая хранимая процедура сформирована без ошибок, то в окне отчеты о выполнении операции в графе «Результат» будет написано «Выполнено». Нажимаем кнопку «Commit» для сохранения разработанной хранимой процедуры в базе данных. После сохранения происходит выполнение хранимой процедуры. Так как в нашем примере у хранимой процедуры отсутствуют входные параметры, на экране появляется окно, содержащее результата выполнения хранимой процедуры (рисунок 3.43). Если у процедуры присутствуют входные параметры, то перед ее запуском пользователю выдается окно, в котором необходимо задать все входящие параметры процедуры.

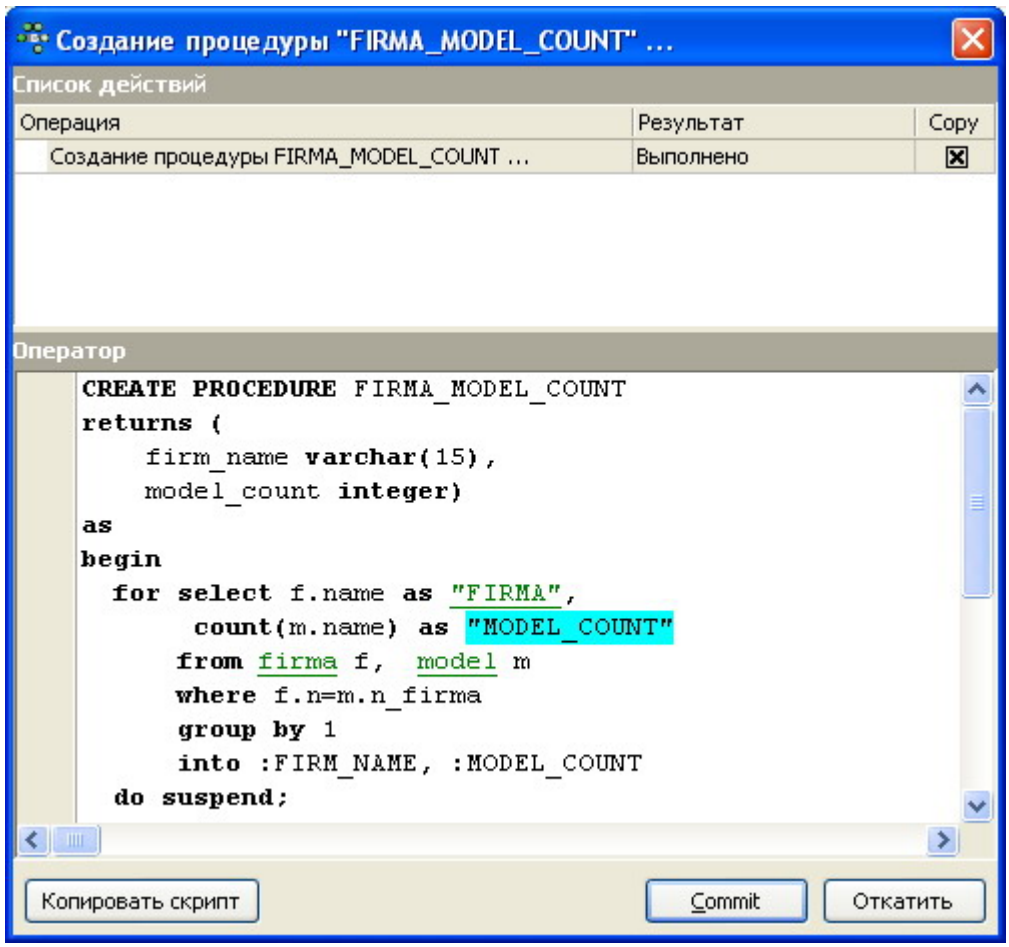

Рисунок 3.42 - Отчет о выполнении операции

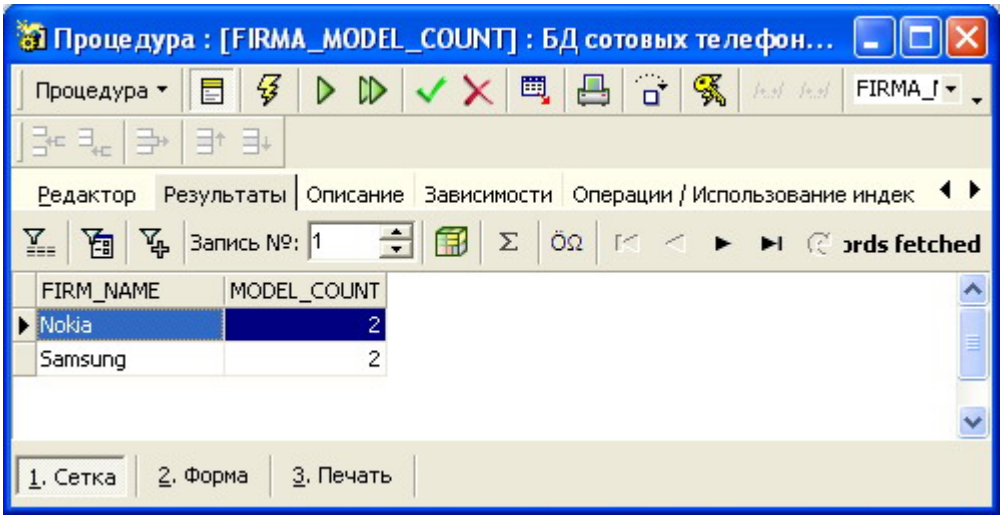

Рисунок 3.43 - Результат выполнения процедуры FIRMA_MODEL_COUNT

Продемонстрируем возможность выполнения select из хранимой процедуры. Для этого добавим в базу данных еще одну модель сотового телефона фирмы Nokia. Тогда в базе данных будет хранится информация о трех моделях сотовых телефонов фирмы Nokia и двух моделях сотовых телефонов Samsung.

Сформируем запрос к хранимой процедуре, который выведет названия фирм производителей сотовых телефонов, у которых в базе данных хранится информации больше чем о двух моделях сотовых телефонов.

```
select firma_name
from firma_model_count
where model_count > 2;
```

Результат выполнения запроса представлен на рисунке 3.44.

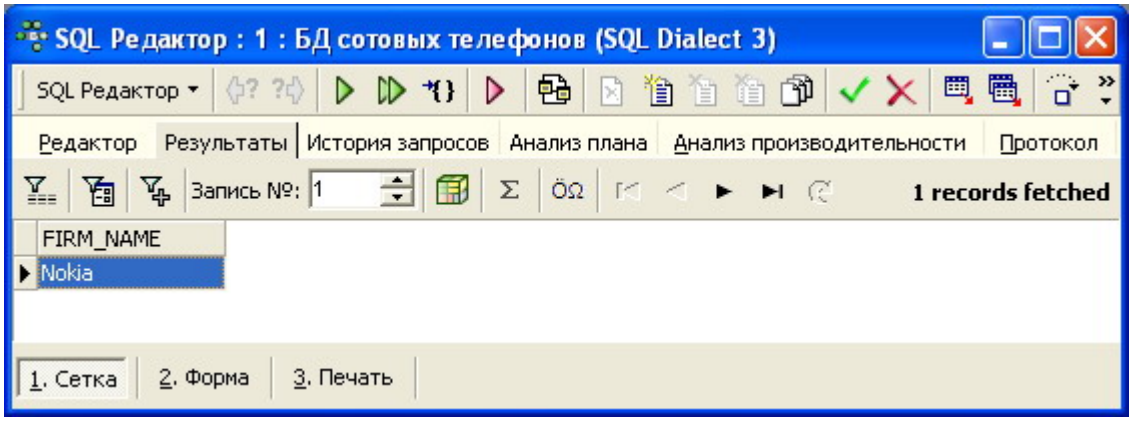

Рисунок 3.44 – Результат выполнения запроса

3.6 Пользователи и права доступа к ресурсам базы данных

В базе данных по умолчанию существует учетная запись администратора. Имя учетной записи SYSDBA; пароль masterkey.

Для создания новых учетных записей и корректировки уже созданных в среде IBExpert используется «Менеджер пользователей». Для задания и изменения прав доступа этих записей используется «Менеджер прав». На рисунке 3.45 выделены кнопки быстрого запуска «Менеджер пользователей» и «Менеджер прав».

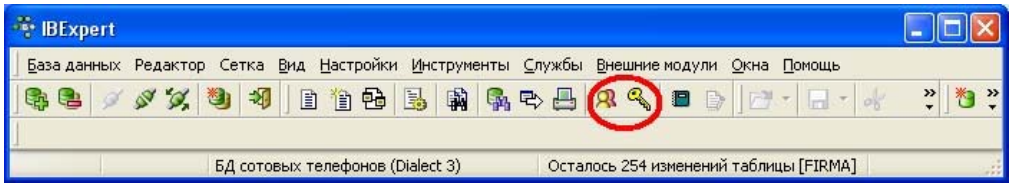

Рисунок 3.45 - Кнопки запуска «Менеджер пользователей» и «Менеджер прав»

Добавим нового пользователя в базу данных. Для этого запустим «Менеджер пользователей» (рисунок 3.46).

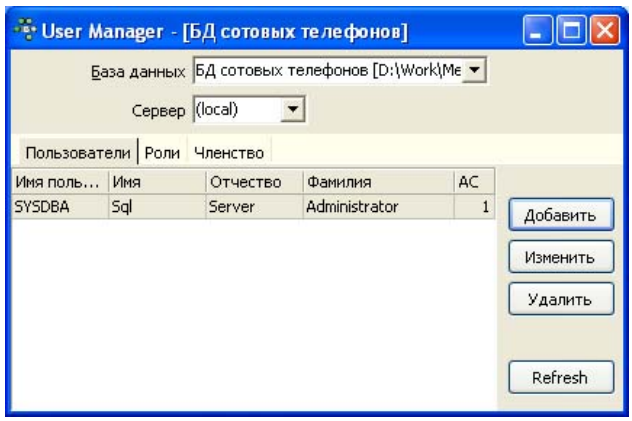

Рисунок 3.46 - Окно «Менеджера пользователей»

Нажимаем кнопку «Добавить». На экране появится диалоговое окно создания нового пользователя. Его необходимо заполнить, как показано на рисунке 3.47.

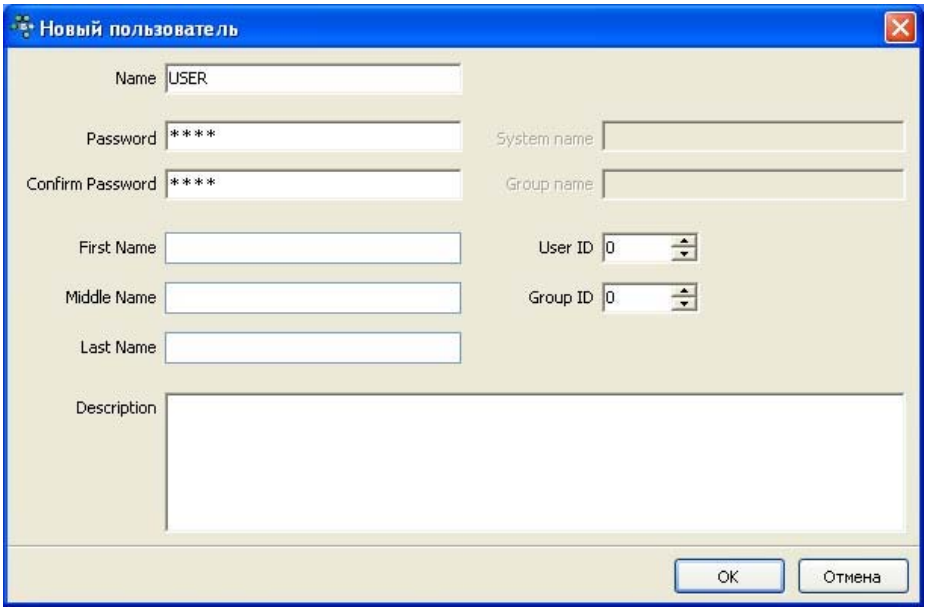

Рисунок 3.47 - Создание нового пользователя

Имя пользователя – USER, пароль - USER. После ввода имени пользователя и пароля нажимаем кнопку «OK».

Теперь, когда новый пользователь создан, ему необходимо дать права. Пусть пользователю USER можно редактировать только таблицу «FUNCTIONS», а остальные таблицы базы данных он может только просматривать. Запускаем «Менеджер прав». Выбираем пользователя USER и с помощью кнопок управления задаем его права (рисунок 3.48).

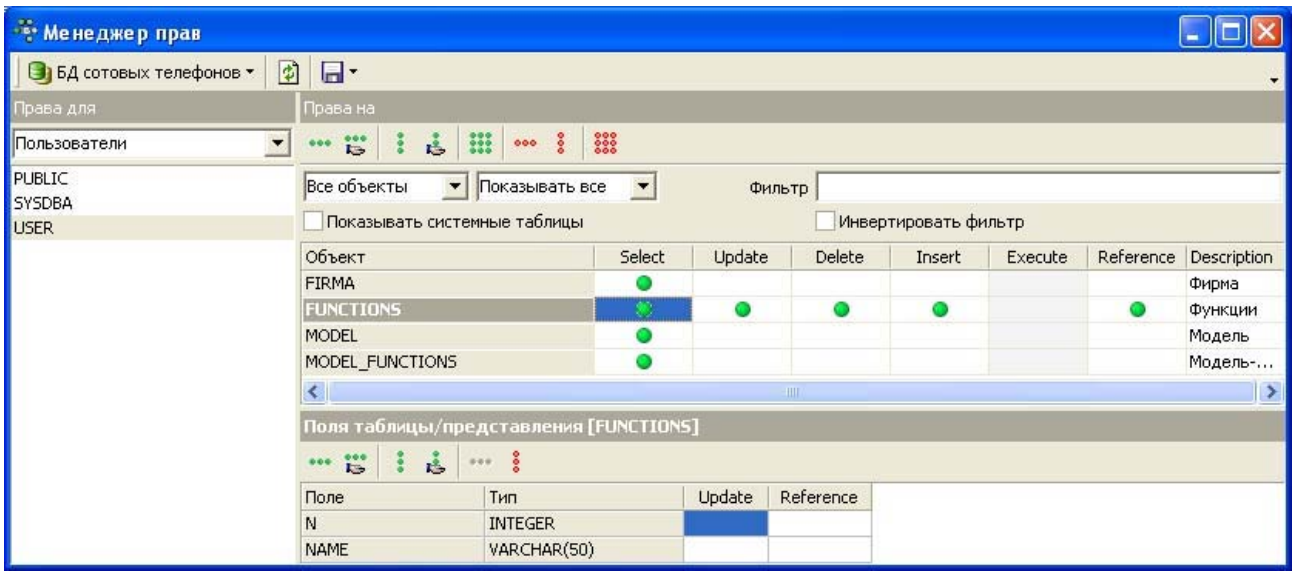

Рисунок 3.48 - Назначение прав доступа пользователю USER

Для пользователя можно определять права доступа как к таблицам и полям, так и к представлениям и процедурам. Существует два варианта выдачи прав обычный и с «GRANT OPTION». Во втором случае пользователь может передавать свои права другим пользователям.

Попробуем использовать созданную учетную запись USER. Для этого создадим новое подключение. Используя «Database Explorer», щелкнем правой кнопкой по названию базы данных и выберем пункт меню «Создать дубликат регистрационной записи», вызовем диалоговое окно регистрации базы данных (рисунок 3.20). Заменим описание базы данных на «БД сотовых телефонов USER» и сменим имя пользователя и пароль на USER. Нажимаем «OK». Затем подключаемся к базе данных. Попробуем добавить новую фирму в таблицу «Firma». Запись не добавляется, на экран выдается сообщение (рисунок 3.49).

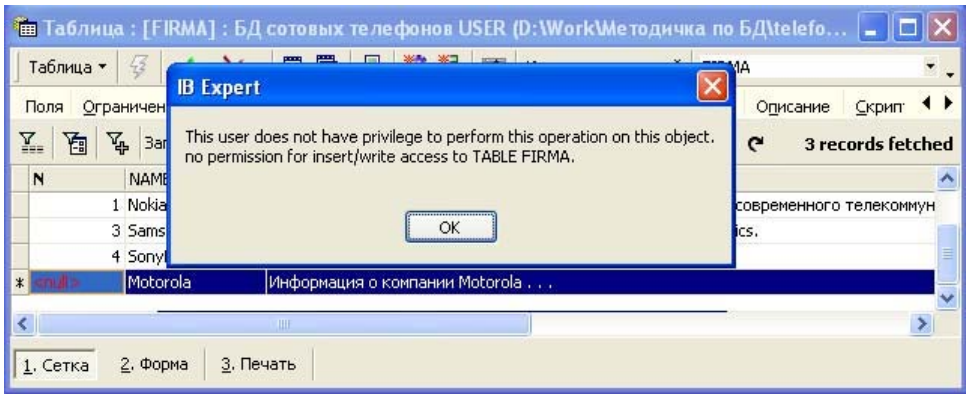

Рисунок 3.49 - Нарушение прав доступа

Попробуем добавить новую функцию «mp3 плеер» в таблицу «Functions». СУБД разрешает пользователю USER добавлять запись в таблицу «Functions» (рисунок 3.50).

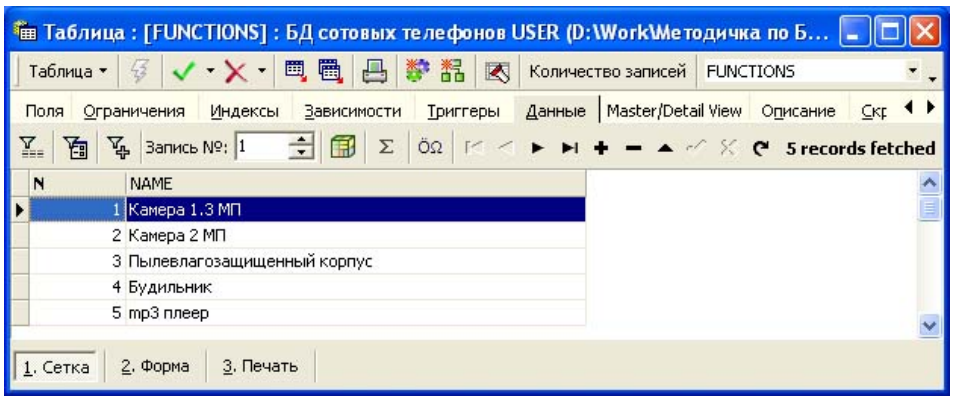

Рисунок 3.50 - Новую функцию «mp3 плеер» в таблицу «Functions»

Описанный алгоритм позволяет определять для пользователей необходимый уровень доступа к информации, хранящейся в базе данных.

3.7 Резервирование и восстановление базы данных

В процессе работы с базой данных необходимо регулярно осуществлять резервное копирование. Выполнять резервное копирование можно при работающих пользователях, однако надо помнить, что сохранены в резервной копии будут только те данные, которые существовали на момент старта процедуры резервного копирования. Запуск процесса резервного копирования из среды IBExpert осуществляется путем выбора в меню «Службы» → «Резервирование базы данных». На экране появится окно диалога резервирования базы данных (рисунок 3.51).

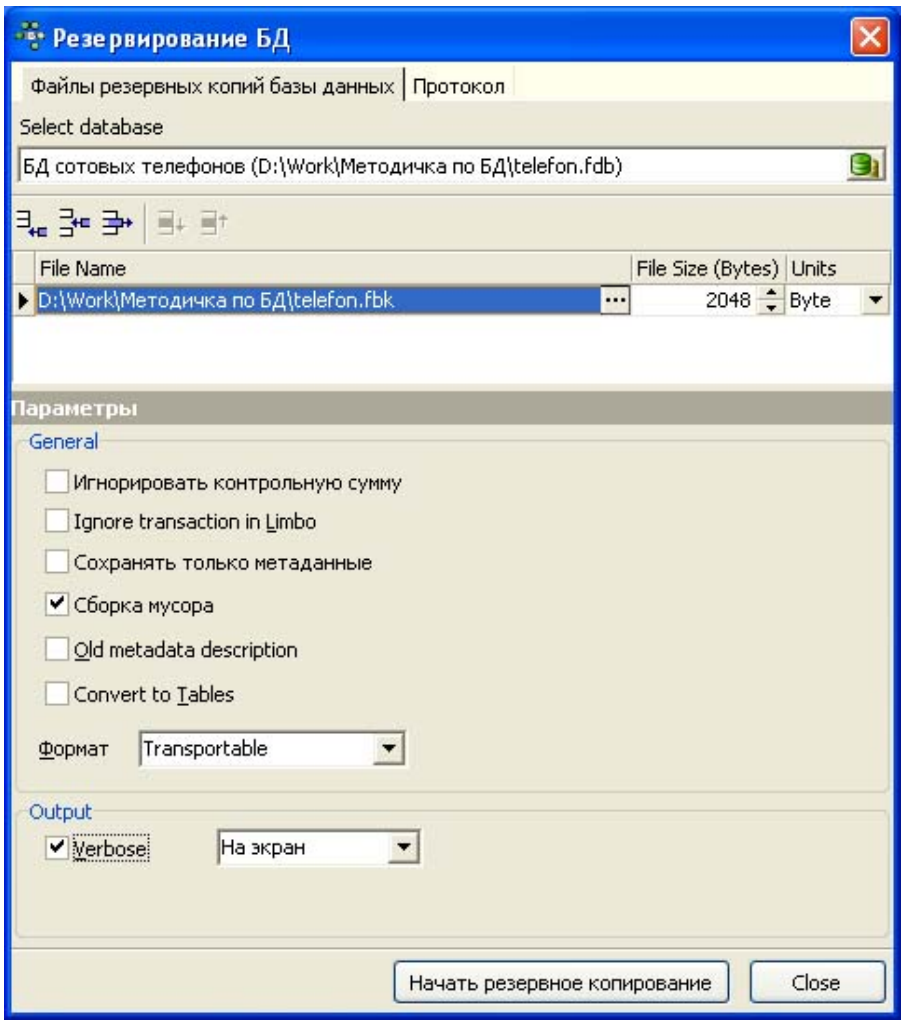

Рисунок 3.51 - Окно диалога резервирования базы данных

Запускаем процесс резервного копирования нажатием на кнопку «Начать резервное копирование». После завершения процесса резервного копирования на экран выводится результативное окно, содержащее статистику по процессу резервного копирования.

Восстановление базы данных из резервной копии рекомендуется делать в новую базу данных, а не поверх существующей базы данных.

Для восстановления необходимо выполнить следующие действия. Создать новую базу данных TELEFON2.FDB. Запустить диалог восстановления базы данных. Для этого через меню IBExpert необходимо выполнить «Службы» → «Восстановление базы данных». Выбрать файл резервной копии базы данных и запустить процесс восстановления, нажав на кнопку «Start Restore». Диалоговое окно восстановления базы данных изображено на рисунке 3.52.

Перед началом восстановления базы данных будет предложено ввести имя пользователя и пароль администратора базы данных SYSDBA/masterkey. После завершения процесса восстановления базы данных из резервной копии на экран выводится результативное окно, содержащее статистику по процессу восстановления базы данных из резервной копии.

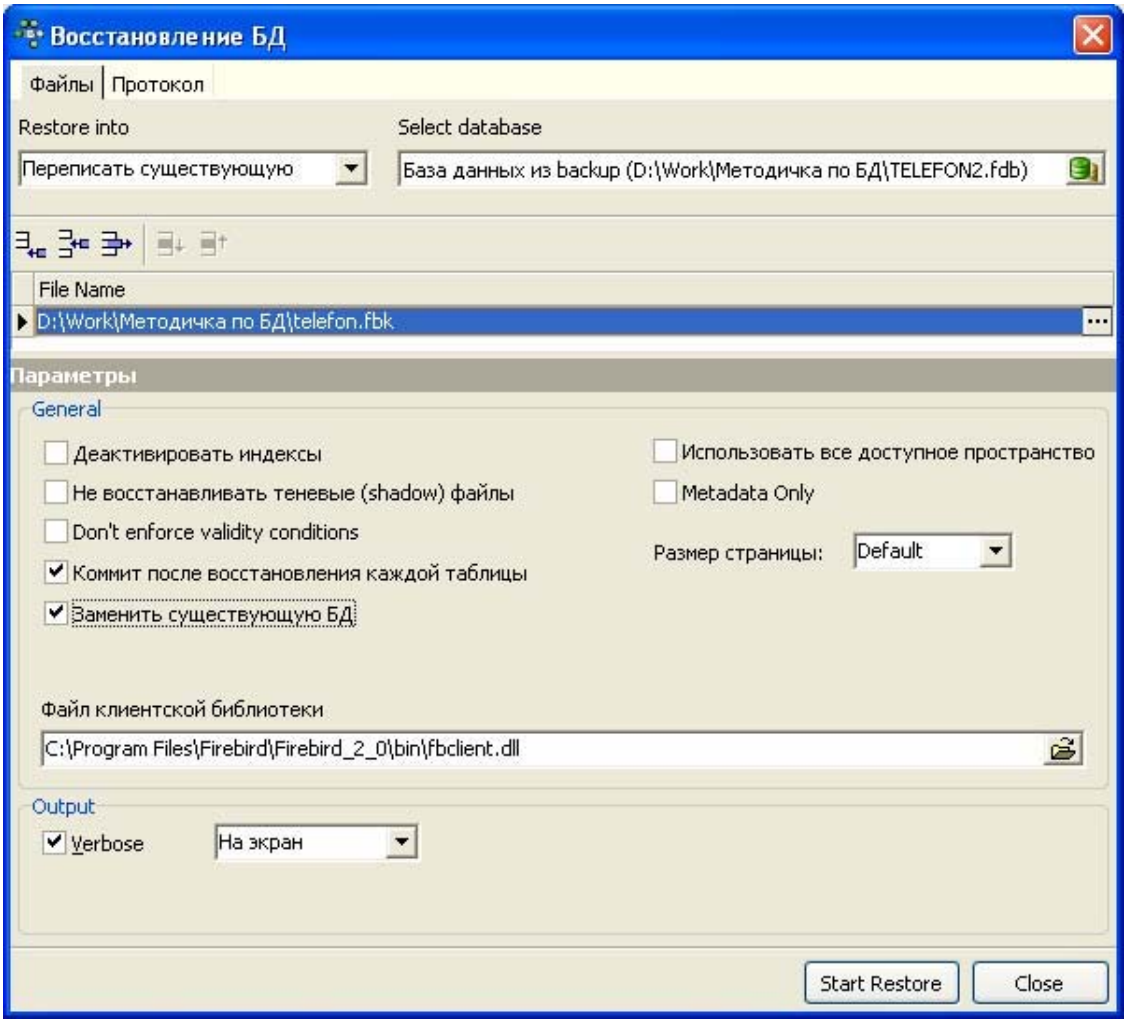

Рисунок 3.52 - Диалоговое окно восстановления базы данных

3.8 Сбор статистики работы с базы данных с применением IBAnalyst

Для сбора статистики работы базы данных и оптимизации ее работы необходимо получать статистику с наполненной и работающей базы данных. В связи с этим рассмотрим основные возможности анализа статистики с применением программного средства IBAnalyst на примерах, поставляемых с дистрибутивом IBAnalyst.

Рассмотрим некоторые основные состояния базы данных и соответствующие показатели этого состояния, полученные программным средством IBAnalyst.

Примеры статистики находятся в каталоге C:\Program Files\IBAnalyst\Examples. Для подключения файла статистики выполните следующшие действия меню программы IBAnalyst «Статистика» → «Загрузить из файла».

Нормальное состояние базы данных

IBAnalyst показывает не только дату создания базы данных, но и анализирует текущее время получения статистики через Services API, или время создания файла статистики.

Поэтому не рекомендуется редактировать полученные файлы статистики, так как IBAnalyst будет некорректно считать среднее число транзакций в день. В этом примере (рисунок 3.53) строка «Транзакций в день» показывает, что в день в этой базе данных совершается примерно 12500 транзакций, и база данных запушена в течение 8-ми дней с момента своего создания или восстановления из резервной копии.

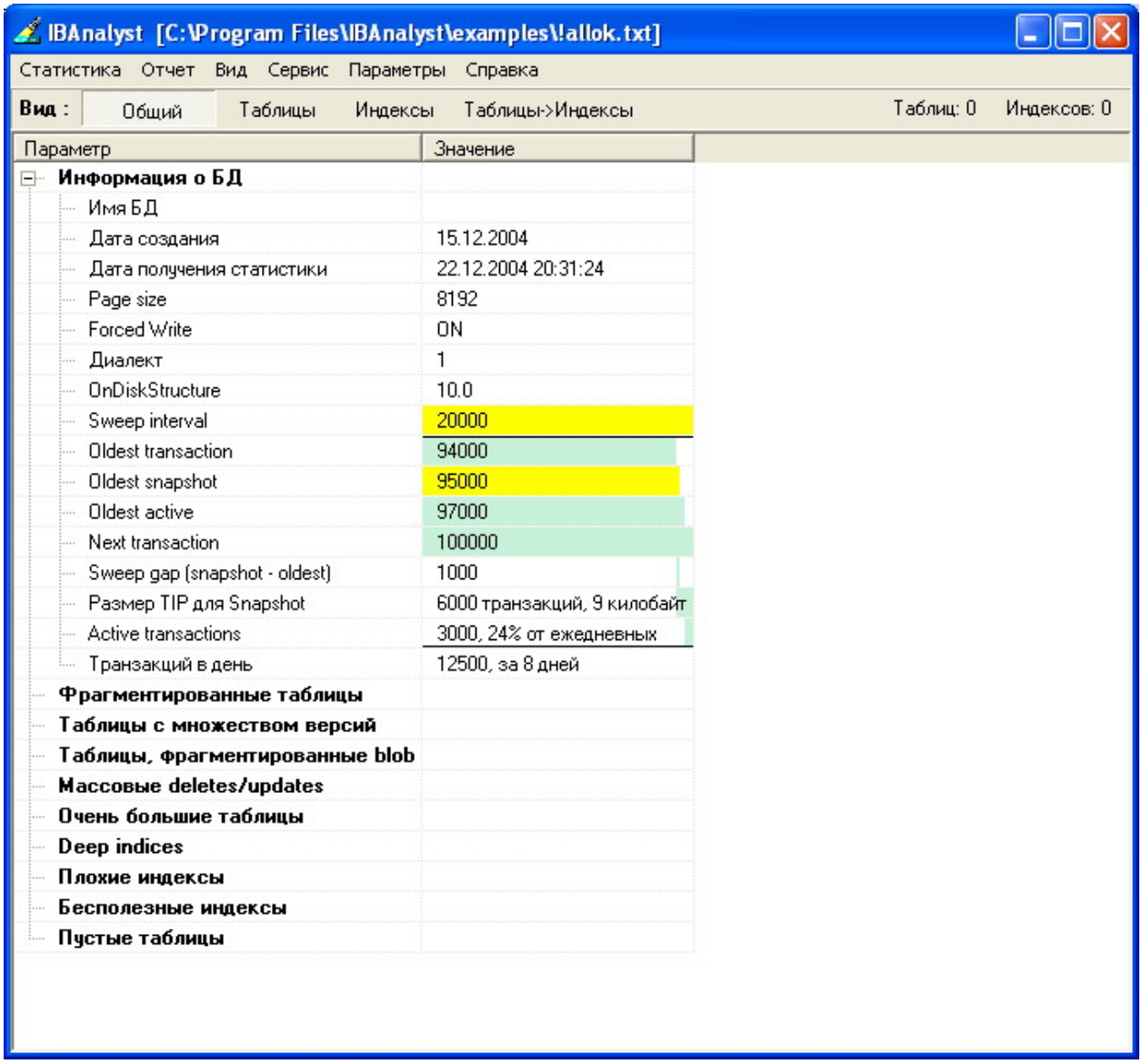

Рисунок 3.53 - Нормальное состояние базы данных

Здесь транзакции Oldest, Oldest snapshot, Oldest active и Next transactions находятся в отличном соотношении.

Перед анализом базы данных убедитесь, что вы получили статистику из базы данных, находящейся в работе. Большинство предупреждений по состоянию транзакций базируются на оценке промышленной загрузки и времени жизни базы данных. Если в базе данных низкая активность (меньше ~1000 транзакций в сутки, обычно в процессе разработки), или база данных длительное время не использовалась, рекомендации и предупреждения IBAnalyst могут оказаться бесполезными.

Много активных транзакций

В этом примере (рисунок 3.54) строка Active transactions выделена, потому что существует большая разница между Oldest active и Next transaction.

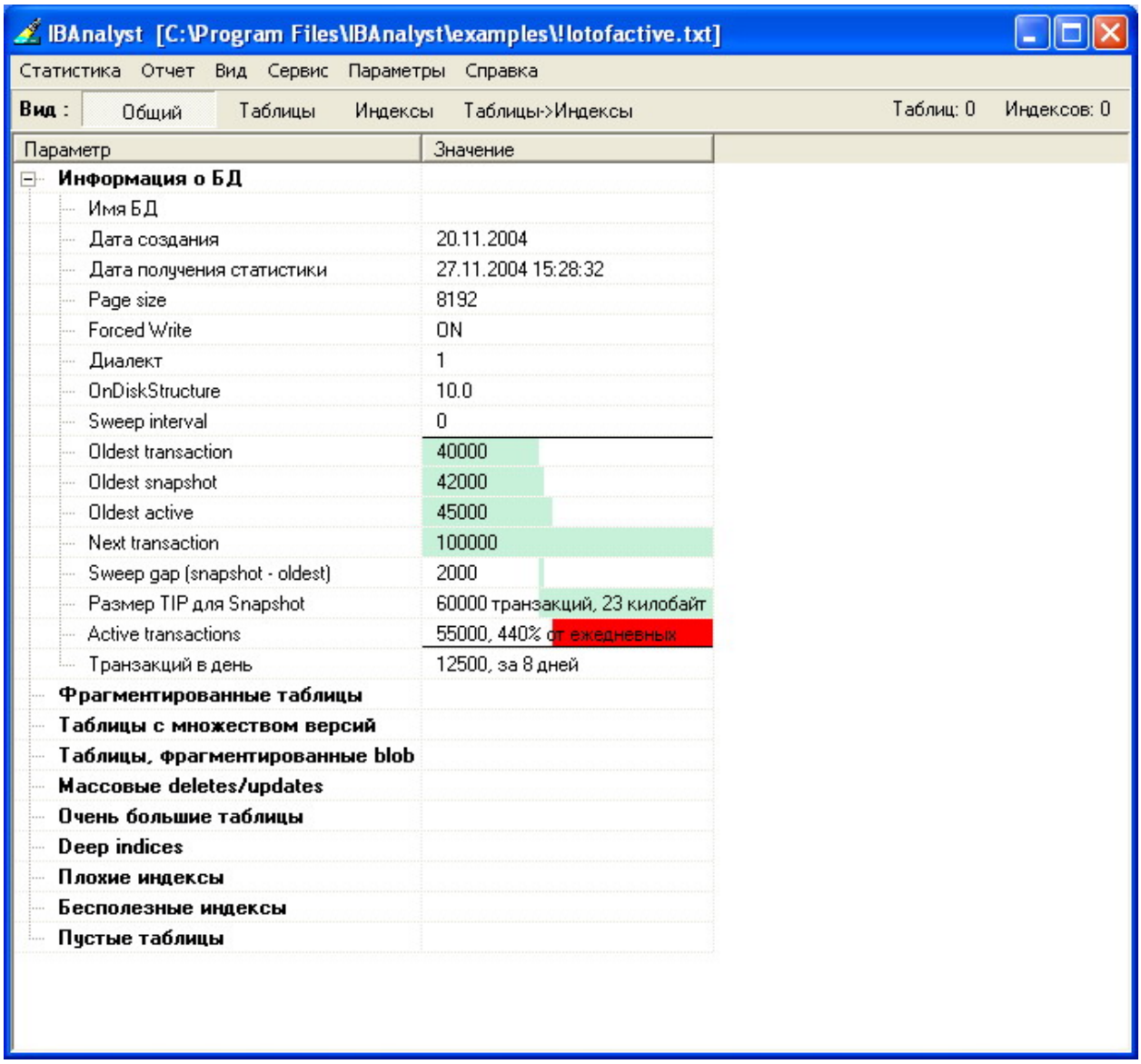

Рисунок 3.54 - Много активных транзакций

Это означает, что на момент сбора статистики некоторая транзакция была активной, в то время как после нее стартовало 55000 транзакций (которые могут быть в любом состоянии – активны, завершены по commit/rollback, или in limbo). Поскольку среднее число транзакций в день примерно ~12500, IBAnalyst выдает предупреждение, что некоторая транзакция находится в активном состоянии уже на протяжении 4,4 дней. Такое может происходить, если:

– некоторое приложение работает и держит открытой хотя бы одну транзакцию – например, пользователь не закрыл приложение, и какая-либо его форма очень долго открыта;

– приложения работают длительное время, и «теряют хэндлы» транзакций в коде – если ваш код (или используемые компоненты/библиотеки) создает транзакции динамически, то возможно где-то не происходит завершения такой транзакции по неизвестным причинам;

– приложения не работают с транзакциями явно (BDE), оставляя работу с транзакциями на усмотрение используемых компонент доступа. В результате длительность активных транзакций никак не контролируется приложением, и можно с большой уверенностью сказать, что большинство транзакций от Oldest Active до Next действительно являются активными;

– используемая библиотека компонент или драйвер работает с «транзакцией по умолчанию», которая не управляется. Поскольку контроля над такой транзакцией нет, она может длиться долго.

Oldest transaction Это номер транзакции, которая завершилась по rollback или была переведена сервером в состояние rollback из-за обрыва соединения, в котором эта транзакция работала. Зависание Oldest transaction буквально означает, что где то в базе данных есть версии записей, которые однозначно являются «мусором» и должны быть удалены. В то же время зависание Oldest transaction не говорит о том, что в базе данных накапливается «мусор» - если все записи в базе данных так или иначе прочитываются приложениями, то вероятнее всего накопленные мусорные версии будут удалены посредством кооперативной сборки «мусора» (фоновой или явной).

Sweeping

Sweep - это процесс, который просматривает все записи в базе данных и пытается убрать их «мусорные» версии, а также подвинуть «вверх» Oldest transaction. В базе данных Sweep interval по умолчанию равен 20000. Когда разница между транзакциями (таблица 3.3) становится больше Sweep interval, сервер автоматически стартует sweep. В результате вы можете наблюдать периодическое падение производительности базы данных.

Когда в статистике обнаруживается, что Sweep interval не равен 0, IBAnalyst обычно помечает эту строку (предупреждая о том, что автоматический sweep может стартовать в любой неожиданный момент) (рисунок 3.55).

В 60% промышленных приложений сталкиваются с проблемой автоматического sweep. Самый простой способ решения этой проблемы – устано-

вить sweep interval в 0, что, собственно, его выключает совсем. Но, если какое-либо приложение сделает много изменений, а затем rollback, то «зависнет» Oldest transaction, и не будет двигаться до тех пор, пока не будет запущен sweep. Поскольку сервер определяет состояния транзакций в интервале от Oldest до Next, эта дистанция будет расти, и производительность будет падать. Для предотвращения этого надо запускать sweep вручную.

Таблица 3.3 – Условия запуска процесса Sweep

Версия СУБД	Условие запуска процесса sweep
InterBase 7.1 и выше	(Oldest Active – Oldest) > Sweep interval
InterBase 4.x, 5.x, 6.x, Firebird, Yaffil	(Oldest Snapshot – Oldest) > Sweep interval

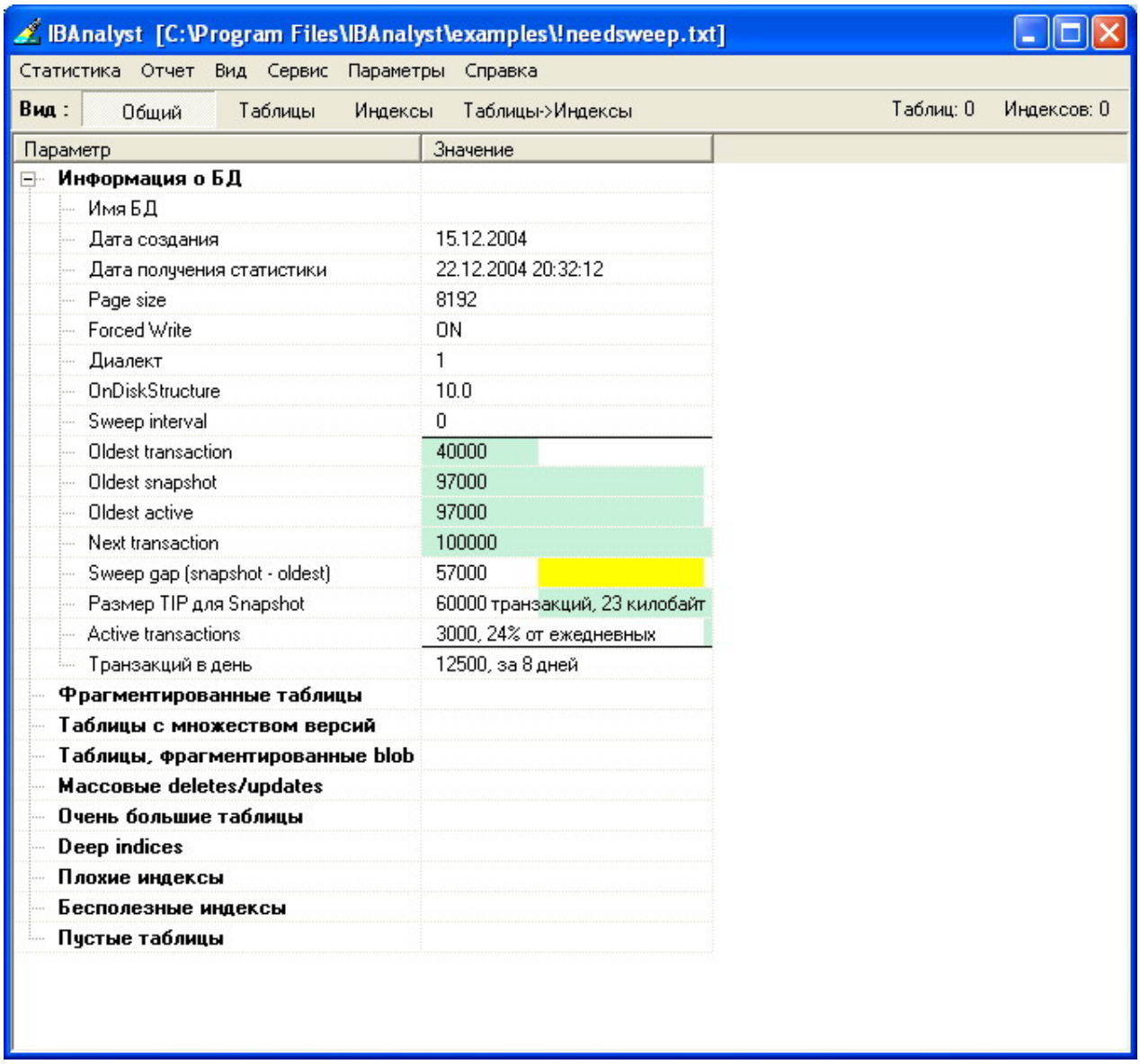

Рисунок 3.55 - Необходимость запуска процесса Sweep

Когда sweep не может выполнить свою работу

Есть несколько ситуаций, когда sweep не может продвинуть вперед Oldest transaction (рисунок 3.56).

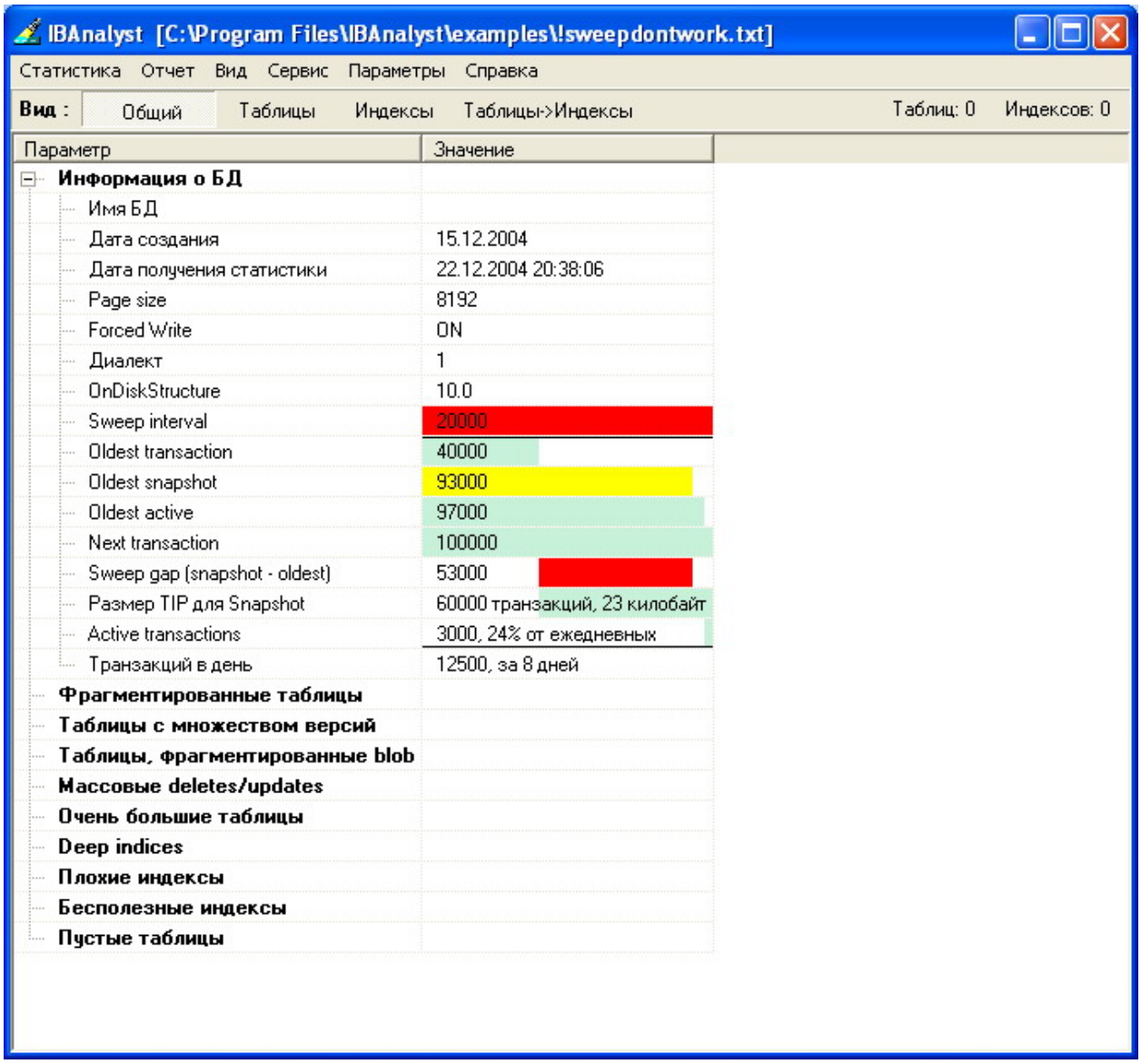

Рисунок 3.56 - Процесс sweep не может выполнить свою работу

Sweep пытается сделать свою работу, то есть проверить все записи на наличие «мусорных версий» и собрать их. Но sweep будет запускаться вновь и вновь, если были проблемы во время работы sweep: либо сервер был остановлен, либо произошла какая-то ошибка, например при обращении к поврежденной «мусорной» версии записи. Такое соотношение транзакций в вашей базе данных может быть, когда:

- статистика была собрана в момент работы sweep;
- sweep работает и пытается собрать мусор в таблице, которая часто обновляется. Sweep будет пытаться убрать мусор снова и снова, до тех пор пока update не прекратятся или станут выполняться реже;

– sweep «завис» на блокировках страниц, если с базой интенсивно работает много пользователей.

В общем, sweep имеет мало шансов завершиться в обозримый интервал времени при высокой загрузке базы данных. Обычно, когда производительность падает до такого состояния, что работать дальше невозможно, администратор базы данных перезагружает сервер, sweep стартует при первом коннекте, и пока остальные пользователи подключаются к БД, sweep успевает завершить свою работу.

Когда ReadCommitted блокирует Oldest Snapshot

Ситуация когда ReadCommitted блокирует Oldest Snapshot представлена на рисунке 3.57.

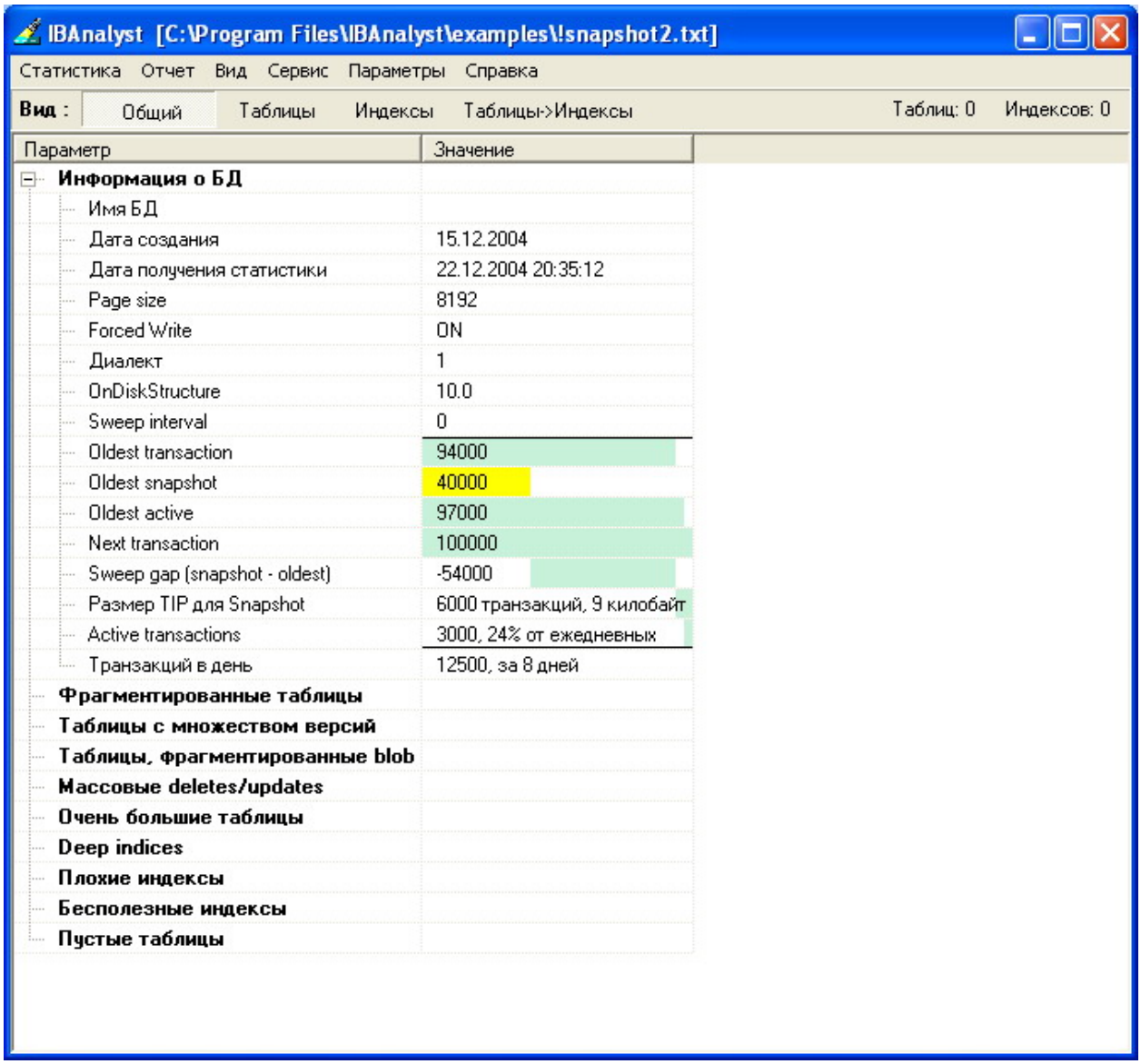

Рисунок 3.57 - ReadCommitted блокирует Oldest Snapshot

Обратите внимание, что Oldest transaction больше Oldest snapshot. И Sweep gap имеет отрицательное значение. Это может произойти в двух случаях. Первый случай, когда есть периодически стартуемые и завершаемые транзакции snapshot. На рисунке 3.57 показано наличие «долгоживущих» snapshot. Второй случай происходит на серверах, кроме IB 7.1, при работе с ReadCommitted транзакциями (или в комбинации read_committed и snapshot). Здесь транзакция ReadCommitted блокирует Oldest Snapshot точно таким же образом, как транзакции Snapshot. Это поведение является историческим, и оно является результатом недоделок при реализации Read Committed на базе Snapshot в Borland (изначально в InterBase были только snapshot-транзакции).

Такое состояние транзакций может быть с эмулировано так:

- стартуйте транзакцию 1, snapshot или read_committed;
- стартуйте/завершите много транзакций read_committed;
- стартуйте транзакцию 2, snapshot или read_committed;
- стартуйте/завершите много транзакций read_committed;
- сделайте commit транзакции 1.

В этот момент текущая активная транзакция 2 (snapshot или read_committed), стартовавшая после старта транзакции 1 (пункт 3), будет удерживать номер Oldest Snapshot (если у вас IB 7.1 и выше, то речь идет только о конкурирующих snapshot. Транзакции read_committed или read_committed+snapshot не будут создавать такой эффект). Так как не было откатов (rollback) больших транзакций, Oldest transaction продвигается вперед и становится больше, чем Oldest Snapshot.

Таким образом, если у вас есть приложения с длинными read_committed транзакциями, вы можете наблюдать именно эту картину. Здесь переработкой транзакций в приложениях сделать ничего нельзя (за исключением установки читающих транзакций в read read_committed). В этом случае sweep не будет автоматически запускаться (если sweep interval <> 0).

Такое поведение в отношении read_committed исправлено в Firebird 2.0.

4 Задания к лабораторным работам

Теперь, когда мы познакомились с основными практическими принципами работы с программным средством IBExpert и СУБД FireBird, можно переходить к выполнению практических заданий.

Каждый обучаемый выбирает тематику разрабатываемой базы данных. Тематика базы данных и краткое ее описание приведены ниже. Создаваемая обучаемым база данных должна состоять минимум из четырех взаимосвязанных таблиц. При проектировании структура базы данных может быть упрощена. Обучающиеся должны разработать свою базу данных по предлагаемой тематике. База данных по уровню детализации должна быть сопоставима с базой данных, рассмотренной в работе (база данных телефонов).

Лабораторная работа № 1

Цель: Получение навыков работы с CASE средством дизайнер БД (IBExpert).

Задание: Используя средства IBExpert'а спроектировать базу данных по вашей тематике. Результат работы оформить в виде отчета. Отчет должен содержать титульный лист, задание, ER -диаграмму, краткое описание базы данных, список использованных источников, нумерацию страниц и оглавление.

Лабораторная работа № 2

Цель: Получение навыков работы с генератором скриптов и SQL-редактором (IBExpert).

Задание: Создать и заполнить данными базу данных, разработанную в лабораторной работе № 1. Результат работы оформить в виде отчета. Отчет должен содержать титульный лист, задание, примеры заполнения базы данных, список использованных источников, нумерацию страниц и оглавление.

Лабораторная работа № 3

Цель: Изучение языка запросов SQL.

Задание: Для базы данных, созданной в лабораторной работе № 2, разработать пять различных запросов. В двух и более запросах должны быть задействованы все таблицы базы данных. Результат работы оформить в виде отчета. Отчет должен содержать титульный лист, задание, тексты запросов и результаты их выполнения, список использованных источников, нумерацию страниц и оглавление.

Лабораторная работа № 4

Цель: Приобретение навыков использования триггеров.

Задание: Для базы данных, созданной в лабораторной работе № 3, разработать триггер, удаляющий записи в подчиненных таблицах при удалении записи в основной таблице. Продемонстрировать каскадное обновление

информации в таблицах базы данных с использованием триггера. Результат работы оформить в виде отчета. Отчет должен содержать титульный лист, задание, скрипты триггеров, запросов и результаты их выполнения, список использованных источников, нумерацию страниц и оглавление.

Лабораторная работа № 5

Цель: Изучение внутреннего языка для написания хранимых процедур.

Задание: Для базы данных, созданной в лабораторной работе № 4, разработать набор хранимых процедур (не менее трех), демонстрирующих возможности использования хранимых процедур. Результат работы оформить в виде отчета. Отчет должен содержать титульный лист, задание, скрипты хранимых процедур, запросы и результаты их выполнения, список использованных источников, нумерацию страниц и оглавление.

Лабораторная работа № 6

Цель: Изучение возможностей расширения функциональности СУБД FireBird за счет использования UDF-функций.

Задание: Для базы данных, созданной в лабораторной работе № 5, разработать простейшую UDF-функцию на любом известном языке программирования и подключить ее к СУБД. Продемонстрировать возможности ее использования. Результат работы оформить в виде отчета. Отчет должен содержать титульный лист, задание, исходные тексты UDF-функции, примеры запросов с ее использованием и результаты их выполнения, список использованных источников, нумерацию страниц и оглавление.

Лабораторная работа № 7

Цель: Приобретение навыков работы с представлениями (View).

Задание: Для базы данных, созданной в лабораторной работе № 6, разработать три примера использования представления. Результат работы оформить в виде отчета. Отчет должен содержать титульный лист, задание, скрипты представлений, примеры запросов и результаты их выполнения, список использованных источников, нумерацию страниц и оглавление.

Лабораторная работа № 8

Цель: Изучения возможностей улучшения производительности базы данных.

Задание: Для базы данных, созданной в лабораторной работе № 7, создать необходимые индексы и продемонстрировать увеличение скорости выполнения запросов к базе данных. Отчет должен содержать титульный лист, задание, команды создания индексов, примеры запросов и результаты их выполнения, список использованных источников, нумерацию страниц и оглавление.

Лабораторная работа № 9

Цель: Изучение возможностей определения прав доступа к ресурсам базы данных.

Задание: Для разрабатываемой базы данных создать двух пользователей: администратора и пользователя. Администратор будет иметь полный доступ к базе данных. Пользователь может только просматривать данные. Продемонстрировать разницу при работе с базой данных, подключившись к ней под разными пользователями. Результат работы оформить в виде отчета. Отчет должен содержать титульный лист, задание, демонстрационные скриншоты, список использованных источников, нумерацию страниц и оглавление.

Лабораторная работа № 10

Цель: Познакомится с возможностями резервного копирования базы данных с применением среды IBExpert.

Задание: Сделать резервную копию базы данных. Изменить названия полей в двух таблицах базы данных и добавить несколько полей в остальные таблицы. Заполнить новые поля информацией. Создать дополнительного пользователя, имеющего полный доступ к половине таблиц и доступ на чтение к остальным таблицам базы данных. Продемонстрировать ограничения на доступ нового пользователя. Восстановить исходную базу данных из резервной копии. Результат работы оформить в виде отчета. Отчет должен содержать титульный лист, задание, демонстрационные скриншоты, список использованных источников, нумерацию страниц и оглавление.

Варианты лабораторных заданий

Вариант № 1

База данных «Студент». Студент обучается в группе. У каждого студента есть одно или несколько хобби («футбол», «шахматы», «танцы» и т.д.).

Вариант № 2

База данных «Автосалон». Автосалон занимается продажей автомобилей нескольких фирм. В автосалоне каждая фирма представлена одной или несколькими моделями. Для моделей существует список опций («фары ксенон», «кожаный салон», «руководители среднего звена» и т.д.).

Вариант № 3

База данных «Гостиница». В гостинице постояльцам предлагаются номера различных типов («одноместные», «двухместные» и т.д.). В номерах может присутствовать дополнительное оборудование/мебель («кондиционер оконный», «сплит-система», «кожаный диван» и т.д.) .

Вариант № 4

База данных лингвистических школ. Школы размещаются на территории города в различных районах («центральный район», «южный» и т.д.). В школе преподают один или несколько иностранных языков («английский», «немецкий» и т.д.).

Вариант № 5

База данных ювелирных украшений. Украшения могут быть различного вида («кольцо», «серьги» и т.д.). Каждое изделие состоит из одного или нескольких драгоценных материалов («желтое золото 586», «платина», «бриллиант» и т.д.).

Вариант № 6

База данных столовая. Каждое блюдо в столовой состоит из различных ингредиентов в определенной массовой пропорции («свинина – 300 гр.», «соль – 5 гр.» и т.д.). По каждому ингредиенту известна цена («свинина 1 кг - 200 руб.»). База данных должна реализовывать возможность составлять меню следующего вида: блюдо – цена. Необходимо учесть, что приготовление увеличивает цену блюда на 15% от стоимости его ингредиентов.

Вариант № 7

База данных риэлтерской компании. Компания занимается работой с различной недвижимостью («квартира», «гараж», «дом» и т.д.). Каждому объекту недвижимости можно поставить в соответствие набор характеристик. Для квартиры и дома – «количество комнат», «площадь» и т.д.

Вариант № 8

База данных по спортзалам. Каждый спортзал находится в конкретном районе города («центральный район», «южный» и т.д.). В спортзале проводятся различные занятия («танцы», «борьба», «бокс» и т.д.).

Вариант № 9

База данных по салонам красоты. Салоны красоты находятся в различных частях города («центральный район», «южный» и т.д.). В салоне красоты осуществляет определенный набор услуг («стрижка/укладка», «наращивание ногтей», «солярий» и т.д.).

Вариант № 10

База данных по кинофильмам. Кинофильмы могут быть нескольких типов («триллер», «комедия», «боевик» и т.д.). Один актер может сниматься в одном или нескольких фильмах.

Вариант № 11

База данных по услугам строительных фирм. Фирма предоставляет перечень услуг («ремонт фасадов», «ремонт помещений», «капитальное строительство» и т.д.). По каждой фирме существует список выполненных ими работ.

Вариант № 12

База данных рестораны. Рестораны размещены в различных районах города («центральный район», «южный» и т.д.). Каждый ресторан предлагает блюда одной или нескольких кухонь («китайская», «итальянская», «европейская» и т.д.).

Вариант № 13

База данных по школам танцев. Помещения школ располагаются в различных районах города («центральный район», «южный» и т.д.). В школе

преподают один или несколько видов танцев («вальс», «танец живота» и т.д.).

Вариант № 14

База данных преподавателей. Преподаватель может иметь ученое звание («кандидат технических наук», «доктор экономических наук»). Сформирован набор дисциплин. Преподаватель может вести одну или несколько дисциплин («математика», «численные методы», «базы данных» и т.д.).

Вариант № 15

База данных услуг автомастерской. Автомастерская предлагает определенный набор услуг («кузовные работы», «малярные работы» и т.д.). Сотрудник мастерской выполняет одну или несколько услуг автомастерской.

Вариант № 16

База данных аптек. Аптеки располагаются в различных районах города («центральный район», «южный» и т.д.). Каждый препарат, продаваемый в аптеке, обладает одним или несколькими свойствами («жаропонижающее», «болеутоляющее» и т.д.).

Заключение

Учебное пособие позволяет проводить лабораторные занятия по курсу «Базы данных» без привлечения коммерческих, платных программных средств. Кроме того, использование среды IBExpert позволяет сконцентрировать внимание обучающегося на изучении клиент-серверных баз данных, а не на программировании, так как не требуется разработка программ-клиентов на языках программирования. Приведенный комплекс лабораторных работ позволит закрепить теоретические знания по курсу у обучающегося.

Скрипт базы данных «Сотовые телефоны»

```
/******************************************************************/
/****                         Domains                          ****/
/******************************************************************/

/******************************************************************/
/****                        Generators                        ****/
/******************************************************************/

CREATE GENERATOR GEN_FIRMA_N;
CREATE GENERATOR GEN_FUNCTIONS_N;
CREATE GENERATOR GEN_MODEL_FUNCTIONS_N;
CREATE GENERATOR GEN_MODEL_N;

/******************************************************************/
/****                        Exceptions                        ****/
/******************************************************************/

/******************************************************************/
/****                        Procedures                        ****/
/******************************************************************/

SET TERM ^ ;

SET TERM ; ^

/******************************************************************/
/****                      Tables and Views                    ****/
/******************************************************************/

CREATE TABLE FIRMA (
    N INTEGER NOT NULL,
    NAME VARCHAR(15) CHARACTER SET WIN1251 COLLATE WIN1251,
    ABOUT VARCHAR(300) CHARACTER SET WIN1251 COLLATE
WIN1251);

CREATE TABLE FUNCTIONS (
    N INTEGER NOT NULL,
    NAME VARCHAR(50) CHARACTER SET WIN1251 COLLATE WIN1251);

CREATE TABLE MODEL (
    N INTEGER NOT NULL,
    N_FIRMA INTEGER,
```

```
   NAME VARCHAR(20) CHARACTER SET WIN1251 COLLATE WIN1251,
   PYEAR SMALLINT,
   WEIGHT SMALLINT,
   SIZES VARCHAR(30) CHARACTER SET WIN1251 COLLATE WIN1251);

CREATE TABLE MODEL_FUNCTIONS (
   N INTEGER NOT NULL,
   N_MODEL INTEGER,
   N_FUNCTIONS INTEGER,
   NOTES VARCHAR(30) CHARACTER SET WIN1251 COLLATE WIN1251);

/******************************************************************/
/****                         Primary keys                     ****/
/******************************************************************/

ALTER TABLE FIRMA ADD CONSTRAINT PK_FIRMA PRIMARY KEY (N);
ALTER TABLE FUNCTIONS ADD CONSTRAINT PK_FUNCTIONS PRI-
MARY KEY (N);
ALTER TABLE MODEL ADD CONSTRAINT PK_MODEL PRIMARY KEY
(N);
ALTER TABLE MODEL_FUNCTIONS ADD CONSTRAINT
PK_MODEL_FUNCTIONS PRIMARY KEY (N);

/******************************************************************/
/****                       Unique constraints                 ****/
/******************************************************************/


/******************************************************************/
/****                         Foreign keys                     ****/
/******************************************************************/

ALTER TABLE MODEL ADD CONSTRAINT FK_FIRMA_MODEL FOREIGN
KEY (N_FIRMA) REFERENCES FIRMA (N);
ALTER TABLE MODEL_FUNCTIONS ADD CONSTRAINT
FK_FUNCTION_MF FOREIGN KEY (N_FUNCTIONS) REFERENCES
FUNCTIONS (N);
ALTER TABLE MODEL_FUNCTIONS ADD CONSTRAINT FK_MODEL_MF
FOREIGN KEY (N_MODEL) REFERENCES MODEL (N);

/******************************************************************/
/****                         Check constraints                ****/
/******************************************************************/
```

```
/******************************************************************/
/****                          Indices                         ****/
/******************************************************************/


/******************************************************************/
/****                          Triggers                        ****/
/******************************************************************/

SET TERM ^ ;

CREATE TRIGGER FIRMA_BI FOR FIRMA
ACTIVE BEFORE INSERT POSITION 0
as
begin
  if (new.n is null) then
    new.n = gen_id(gen_firma_n,1);
end
^

CREATE TRIGGER FUNCTIONS_BI FOR FUNCTIONS
ACTIVE BEFORE INSERT POSITION 0
as
begin
  if (new.n is null) then
    new.n = gen_id(gen_functions_N,1);
end
^

CREATE TRIGGER MODEL_BEFORE_DELETE FOR MODEL
ACTIVE BEFORE DELETE POSITION 0
AS
begin
  delete
  from model_functions mf
  where mf.n_model=old.n;
end
^

CREATE TRIGGER MODEL_BI FOR MODEL
ACTIVE BEFORE INSERT POSITION 0
as
begin
  if (new.n is null) then
    new.n = gen_id(gen_model_N,1);
```

```
end
^

CREATE TRIGGER MODEL_FUNCTIONS_BI FOR MODEL_FUNCTIONS
ACTIVE BEFORE INSERT POSITION 0
as
begin
  if (new.n is null) then
    new.n = gen_id(gen_model_functions_n,1);
end
^

SET TERM ; ^

/******************************************************************/
/****                         Procedures                       ****/
/******************************************************************/

SET TERM ^ ;

SET TERM ; ^

/******************************************************************/
/****                      Object descriptions                 ****/
/******************************************************************/

DESCRIBE TABLE FIRMA 'Фирма';

DESCRIBE FIELD N TABLE FIRMA 'Номер записи';

DESCRIBE FIELD NAME TABLE FIRMA 'Название';

DESCRIBE FIELD ABOUT TABLE FIRMA 'О фирме ...';

DESCRIBE TABLE FUNCTIONS 'Функции';

DESCRIBE FIELD N TABLE FUNCTIONS 'Номер записи';

DESCRIBE FIELD NAME TABLE FUNCTIONS 'Функция';

DESCRIBE TABLE MODEL 'Модель';

DESCRIBE FIELD N TABLE MODEL 'Номер записи';

DESCRIBE FIELD N_FIRMA TABLE MODEL 'Номер записи фирмы';
```

```
DESCRIBE FIELD NAME TABLE MODEL 'Название';

DESCRIBE FIELD PYEAR TABLE MODEL 'Год выпуска';

DESCRIBE FIELD WEIGHT TABLE MODEL 'Вес';

DESCRIBE FIELD SIZES TABLE MODEL 'Размеры';

DESCRIBE TABLE MODEL_FUNCTIONS 'Модель-функции';

DESCRIBE FIELD N TABLE MODEL_FUNCTIONS 'Номер записи';

DESCRIBE FIELD N_MODEL TABLE MODEL_FUNCTIONS 'Номер записи
модели';

DESCRIBE FIELD N_FUNCTIONS TABLE MODEL_FUNCTIONS 'Номер за-
писи функции';

DESCRIBE FIELD NOTES TABLE MODEL_FUNCTIONS 'Примечания';
```

Список использованных источников

1. Ковязин А. Н. Мир InterBase. Архитектура, администрирование и разработка приложений баз данных в InterBase/ Ковязин А. Н., Востриков С. М.. – СПб..: КУДИЦ-Образ, 2005, - 496 с.
2. Бори Х. Firebird: руководство разработчика баз данных: Пер. с англ. – СПб.: БХВ-Петербург, 2006. – 1104 с.
3. http://www.ibase.ru/
4. http://www.ibexpert.com/
5. Карпова Т.С. Базы данных. Модели, разработка, реализация. Учебник., Ст-П., 2001. - 304 с.
6. Коннолли Т. Базы данных: проектирование, реализация, сопровождение. Теория и практика, 2-е изд. : Пер. с англ. : Уч. пос./ Коннолли Т., Бегг К., Страчан А. – М.: Изд. дом "Вильямс", 2000. – 1120 с.
7. Корнеев В.В. Базы данных. Интеллектуальная обработка информации/ Корнеев В.В., Гареев А.Ф., Васютин С.В., Райх В.В. - М.: Нолидж, 2000. - 352 с.
8. Кренке. Д. Теория и практика построения база данных. 8-е изд. – СПб.: Питер, 2003. – 800 с.
9. Кузнецов С. Д. Базы данных. Модели и языки. – М.: Бином-Пресс, 2008. - 720 с.
10. Марков А.С. Базы данных. Введение в теорию и методологию/ Марков А.С., Лисовский К.Ю. – М.: Финансы и статистика, 2006, - 512 с.
11. Хансен Г. Базы данных: разработка и управление/ Хансен Г., Хансен Д. - М.: ЗАО «Издательство БИНОМ», 1999. - 704 с.
12. Хоменко Д.А., Цыганков В.М., Мальцев М.Г. Базы данных: Учебник для высших учебных заведений/ Под редакцией проф. А.Д.Хоменко. – Спб.: КОРОНА принт, 2000. – 416 с.